ATYPON

Java and DevOps Training

Decentralized Cluster-Based NoSQL DB System

Prepared by: Bashar Mohmmad Alsaid Ahmad

Table of Contents

1.0 **Introduction**:

In this report I will describe how I built decentralized cluster-based NoSQL database system. It was a challenging and useful project.

The system comprises a bootstrapper that initializes the database nodes and assigns them the initial configuration, a client that communicates with both the bootstrapper and the database nodes, and a demo to showcase how users can create their own projects using the database.

Given that the database is a disk database, fast read and write operations were employed in building the system to improve its performance. Additionally, the database is multithreaded to allow multiple users to access it concurrently.

The report also describes how load balancing was implemented in various parts of the system, including the bootstrapper, which distributes users to nodes, the node and affinity distributing mechanism.

2.0 Software Architecture

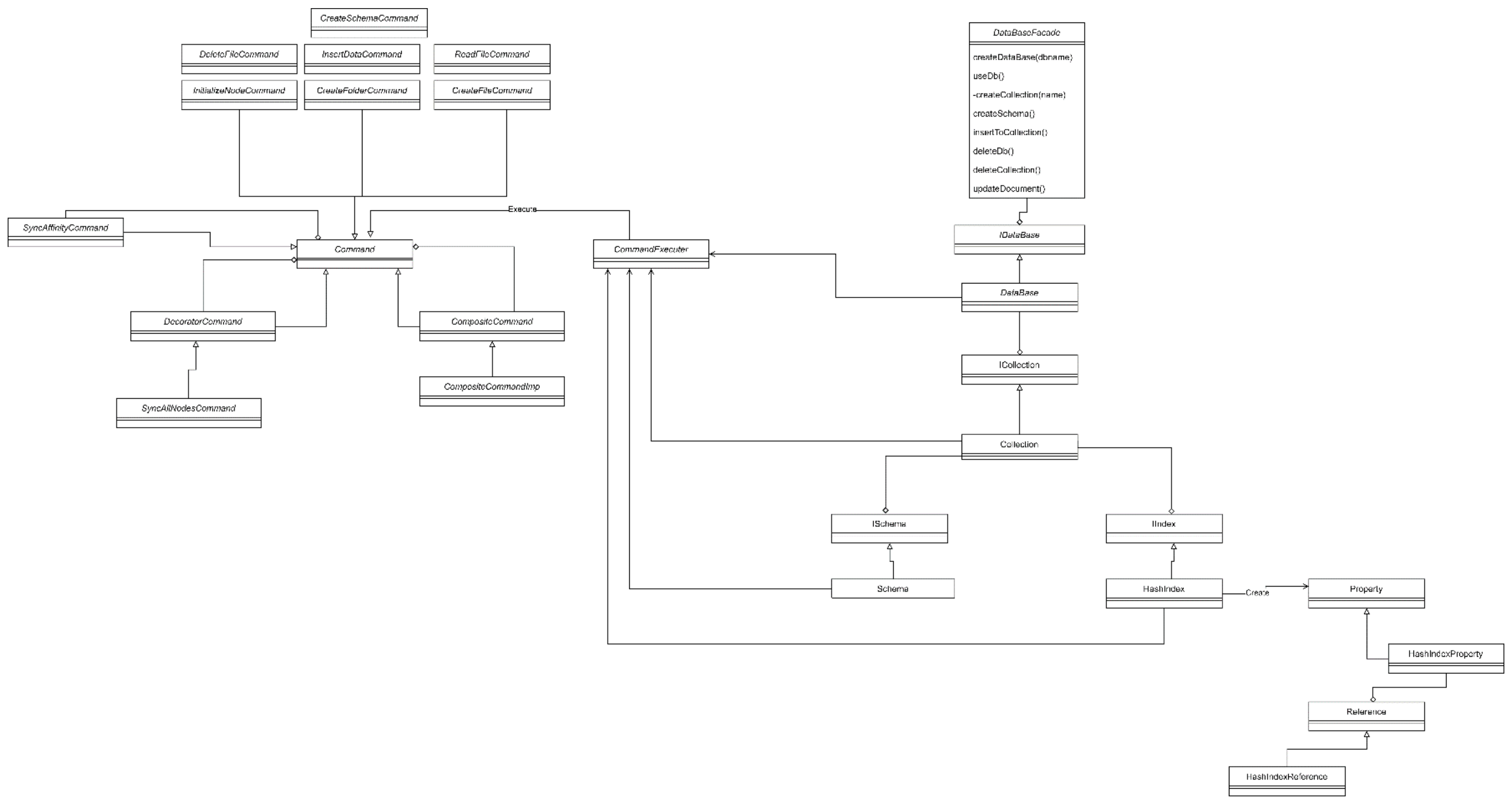
1. The bootstrapping node: The first component of this system is responsible for creating clusters and providing worker nodes with necessary information.
2. The Maven project contains all the database implementations. The project is built using Maven, and a JAR file is created and used in the Spring Boot application.
3. The third component is the Spring Boot application that uses the Maven project as a library. The Maven project is included in the pom.xml file of the Spring Boot application.

To interact with the system, users send requests to the REST API controller, which is handled by the Spring Boot application. The application processes the requests and returns the responses back to the user.

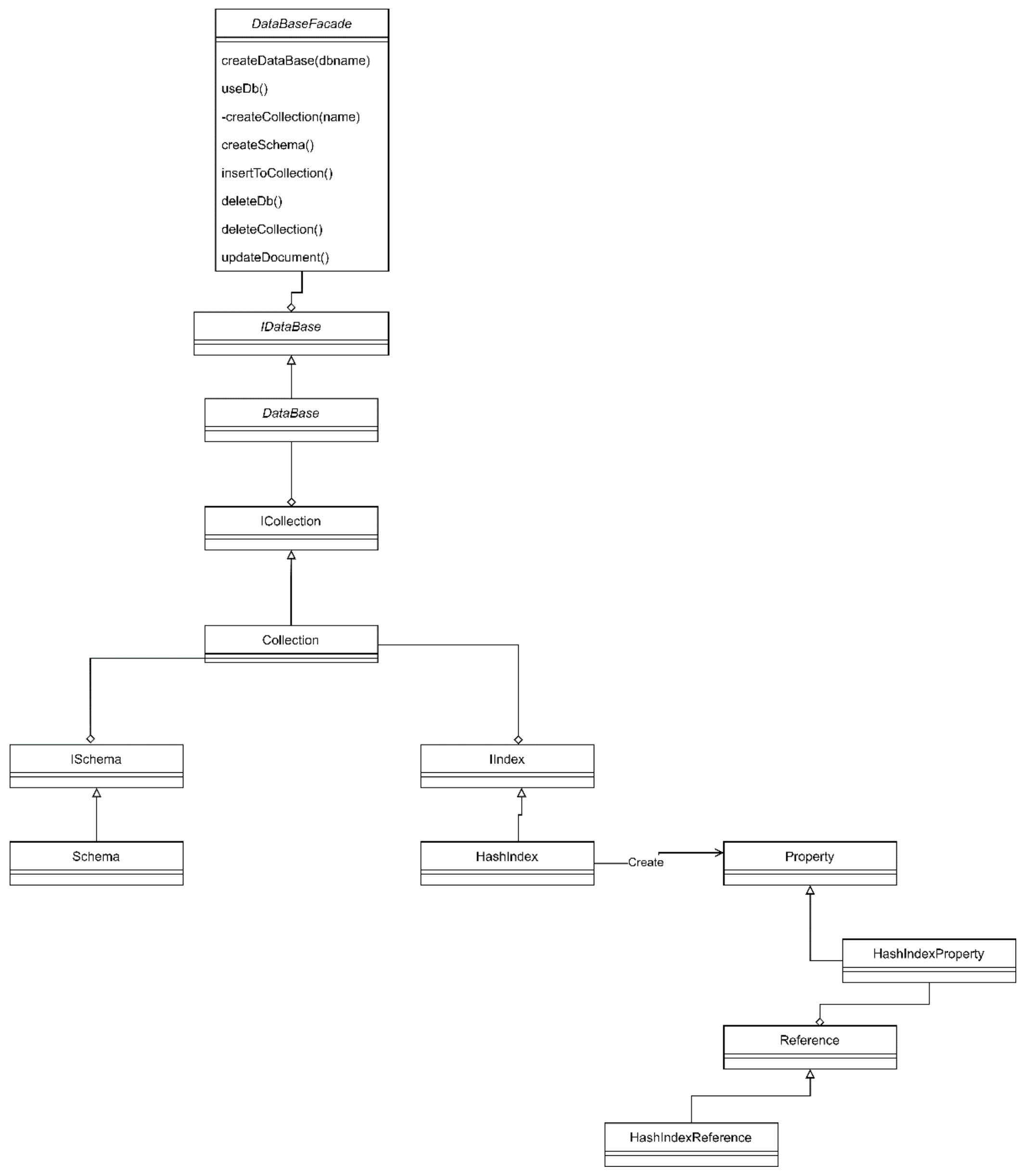
3.0 Database Implementation

In this chapter, I will describe my approach to implementing the system design, including the use of design patterns and data structures. Throughout the implementation process, my primary focus was to ensure that the database was easy to maintain, flexible, and adhered to the SOLID principles as much as possible. Additionally, I leveraged design patterns to support my implementation and make it open for extension while being closed for modification.

3.1 Class diagram:

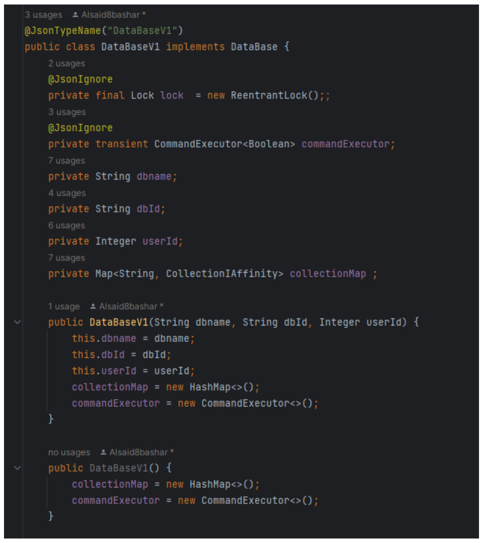


3.2 Database Component:



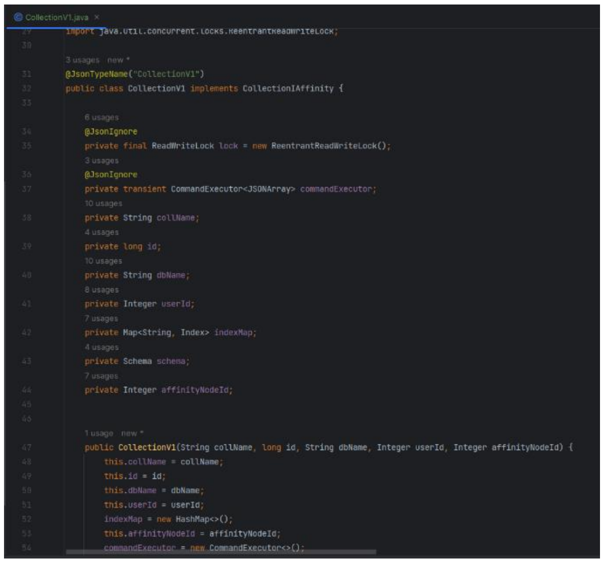
In a NoSQL database, there are several components, including a database, collections, schemas, and indexes.

1 - the Database class would typically store information about the database and be responsible for managing the collections within it, including operations like creating and deleting them.



```java
3 usages  Alsaid8bashar *
@JsonTypeName("DataBaseV1")
public class DataBaseV1 implements DataBase {
    2 usages
    @JsonIgnore
    private final Lock lock  = new ReentrantLock();;
    3 usages
    @JsonIgnore
    private transient CommandExecutor<Boolean> commandExecutor;
    7 usages
    private String dbname;
    4 usages
    private String dbId;
    6 usages
    private Integer userId;
    7 usages
    private Map<String, CollectionIAffinity> collectionMap ;

    1 usage  Alsaid8bashar *
    public DataBaseV1(String dbname, String dbId, Integer userId) {
        this.dbname = dbname;
        this.dbId = dbId;
        this.userId = userId;
        collectionMap = new HashMap<>();
        commandExecutor = new CommandExecutor<>();
    }

    no usages  Alsaid8bashar *
    public DataBaseV1() {
        collectionMap = new HashMap<>();
        commandExecutor = new CommandExecutor<>();
    }
```

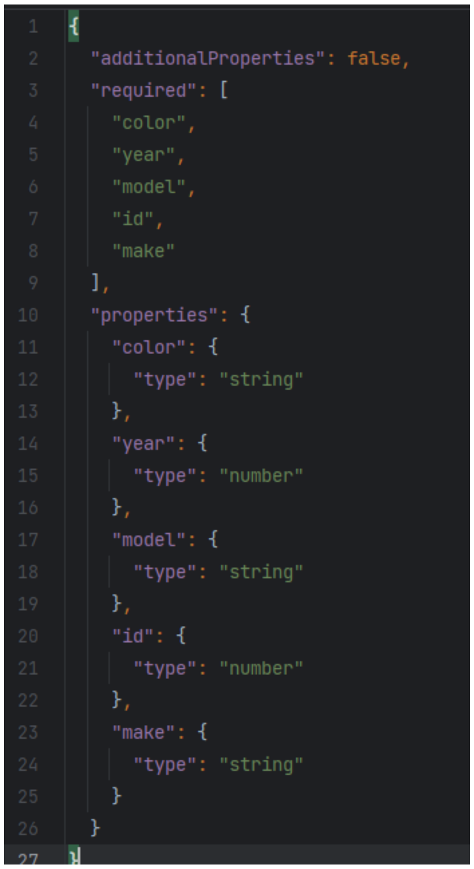
2 – In NoSQL databases, the Collection class is responsible for holding the schema for the collection itself. This means that each collection has its own schema. The Collection class also maintains the index map, which contains the indexes that have been created for the collection.



```java
CollectionV1.java
import java.util.concurrent.locks.ReentrantReadWriteLock;

3 usages  new *
@JsonTypeName("CollectionV1")
public class CollectionV1 implements CollectionIAffinity {

    6 usages
    @JsonIgnore
    private final ReadWriteLock lock = new ReentrantReadWriteLock();
    3 usages
    @JsonIgnore
    private transient CommandExecutor<JSONArray> commandExecutor;
    10 usages
    private String collName;
    4 usages
    private long id;
    10 usages
    private String dbName;
    8 usages
    private Integer userId;
    7 usages
    private Map<String, Index> indexMap;
    4 usages
    private Schema schema;
    7 usages
    private Integer affinityNodeId;


    1 usage  new *
    public CollectionV1(String collName, long id, String dbName, Integer userId, Integer affinityNodeId) {
        this.collName = collName;
        this.id = id;
        this.dbName = dbName;
        this.userId = userId;
        indexMap = new HashMap<>();
        this.affinityNodeId = affinityNodeId;
        commandExecutor = new CommandExecutor<>();
```

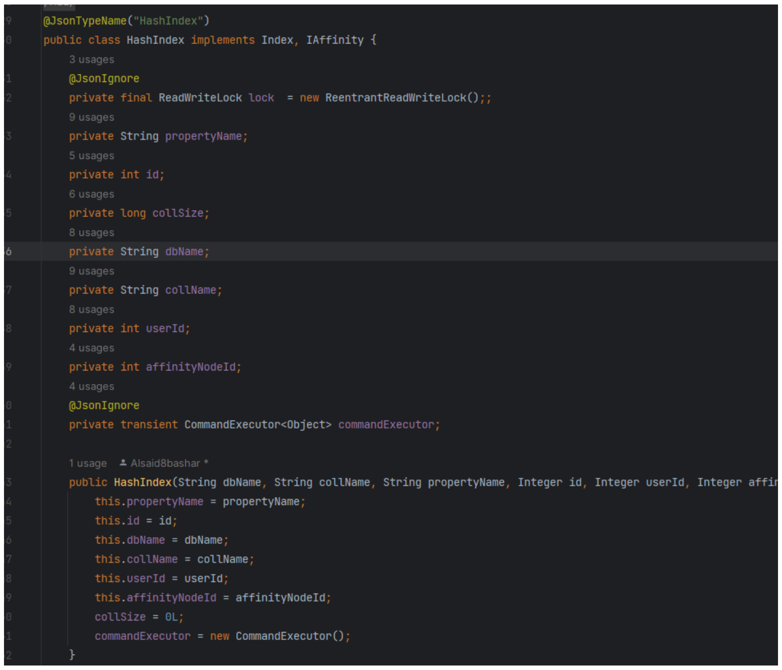
3-Shema class : The Shema class is responsible for creating the schema for the collection. I prompt the user to provide an object that represents the collection schema. To construct the schema, I utilize the builder design pattern and will provide more information on the builder details later.

This is how the Shema should appear on the disk:



```json
{
  "additionalProperties": false,
  "required": [
    "color",
    "year",
    "model",
    "id",
    "make"
  ],
  "properties": {
    "color": {
      "type": "string"
    },
    "year": {
      "type": "number"
    },
    "model": {
      "type": "string"
    },
    "id": {
      "type": "number"
    },
    "make": {
      "type": "string"
    }
  }
}
```

4– The index class stores information such as the properties contained in the index and the last document added. Therefore, in the event of any data updates, the index will persist and include the new files.

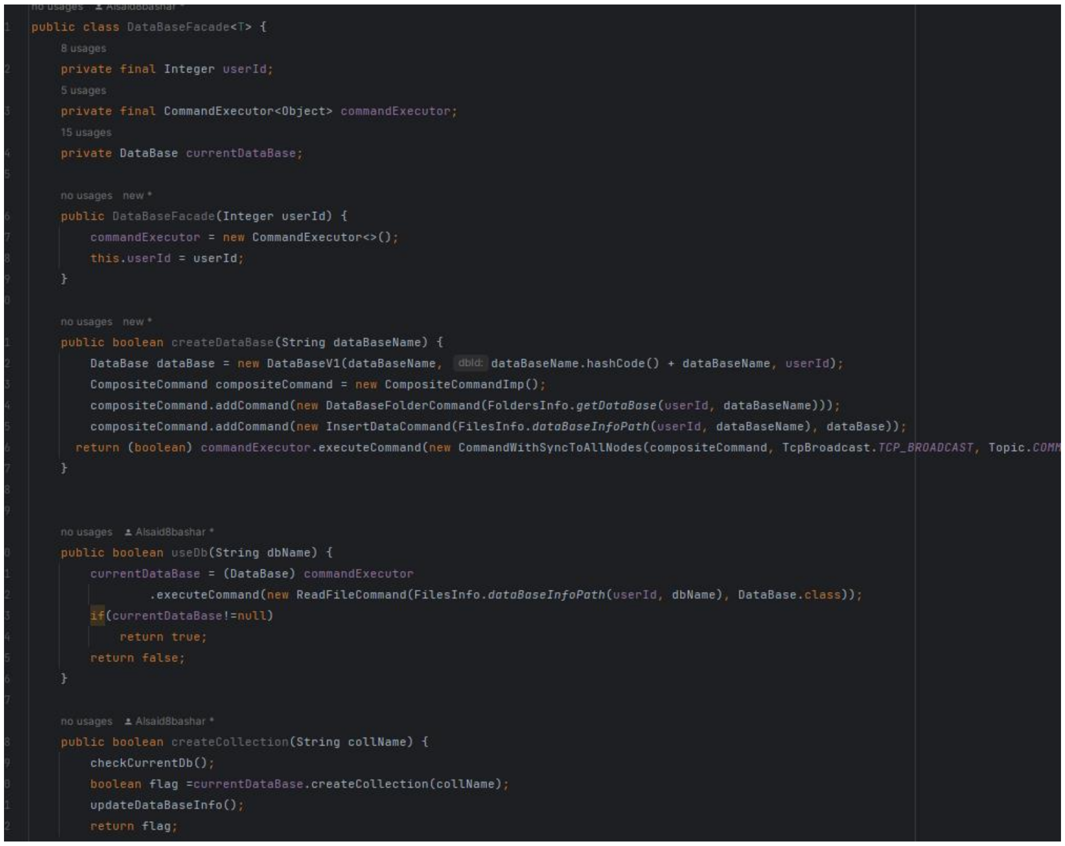


```java
@JsonTypeName("HashIndex")
public class HashIndex implements Index, IAffinity {
    3 usages
    @JsonIgnore
    private final ReadWriteLock lock  = new ReentrantReadWriteLock();;
    9 usages
    private String propertyName;
    5 usages
    private int id;
    6 usages
    private long collSize;
    8 usages
    private String dbName;
    9 usages
    private String collName;
    8 usages
    private int userId;
    4 usages
    private int affinityNodeId;
    4 usages
    @JsonIgnore
    private transient CommandExecutor<Object> commandExecutor;

    1 usage   Alsaid8bashar *
    public HashIndex(String dbName, String collName, String propertyName, Integer id, Integer userId, Integer affir
        this.propertyName = propertyName;
        this.id = id;
        this.dbName = dbName;
        this.collName = collName;
        this.userId = userId;
        this.affinityNodeId = affinityNodeId;
        collSize = 0L;
        commandExecutor = new CommandExecutor();
    }
```

I will provide detailed information about the index component in the upcoming chapters.

5- Façade class: To improve the readability and usability ,I have implemented the Facade Design Pattern as the System's Entry Point .

To utilize the database, the user needs to first log in to the system successfully. Once logged in, they can perform various operations such as creating a database, creating a collection, defining a schema for the collection, inserting data, deleting a collection or database, updating documents, and reading collection data. To create a database, the user can use the façade class. They can then choose which database to work on using the useDB function. Finally, they can perform operations such as creating a collection, defining a schema for the collection, inserting data, deleting a collection or database, updating documents, and finding data using the find function.



```
no usages  Alsaid8bashar *
public class DataBaseFacade<T> {
    8 usages
    private final Integer userId;
    5 usages
    private final CommandExecutor<Object> commandExecutor;
    15 usages
    private DataBase currentDataBase;

    no usages  new *
    public DataBaseFacade(Integer userId) {
        commandExecutor = new CommandExecutor<>();
        this.userId = userId;
    }

    no usages  new *
    public boolean createDataBase(String dataBaseName) {
        DataBase dataBase = new DataBaseV1(dataBaseName,  dbId: dataBaseName.hashCode() + dataBaseName, userId);
        CompositeCommand compositeCommand = new CompositeCommandImp();
        compositeCommand.addCommand(new DataBaseFolderCommand(FoldersInfo.getDataBase(userId, dataBaseName)));
        compositeCommand.addCommand(new InsertDataCommand(FilesInfo.dataBaseInfoPath(userId, dataBaseName), dataBase));
      return (boolean) commandExecutor.executeCommand(new CommandWithSyncToAllNodes(compositeCommand, TcpBroadcast.TCP_BROADCAST, Topic.COMM
    }


    no usages  Alsaid8bashar *
    public boolean useDb(String dbName) {
        currentDataBase = (DataBase) commandExecutor
                .executeCommand(new ReadFileCommand(FilesInfo.dataBaseInfoPath(userId, dbName), DataBase.class));
        if(currentDataBase!=null)
            return true;
        return false;
    }

    no usages  Alsaid8bashar *
    public boolean createCollection(String collName) {
        checkCurrentDb();
        boolean flag =currentDataBase.createCollection(collName);
        updateDataBaseInfo();
        return flag;
```

4.0 Design Patterns:

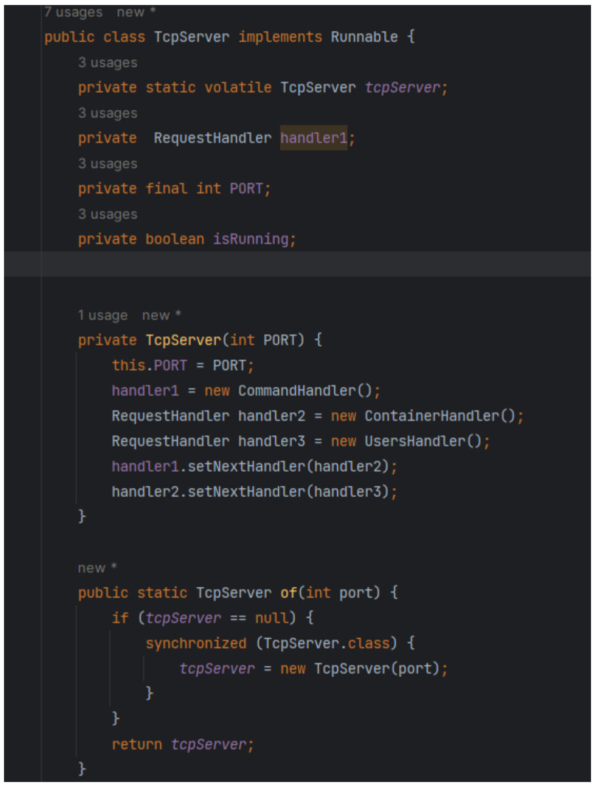
Design patterns made the design more flexible, more resilient to change, and easier to maintain. In this subsection ill talk about the design patterns I have implemented and why I chose them.

4.1 Creational

- Singleton Design Pattern

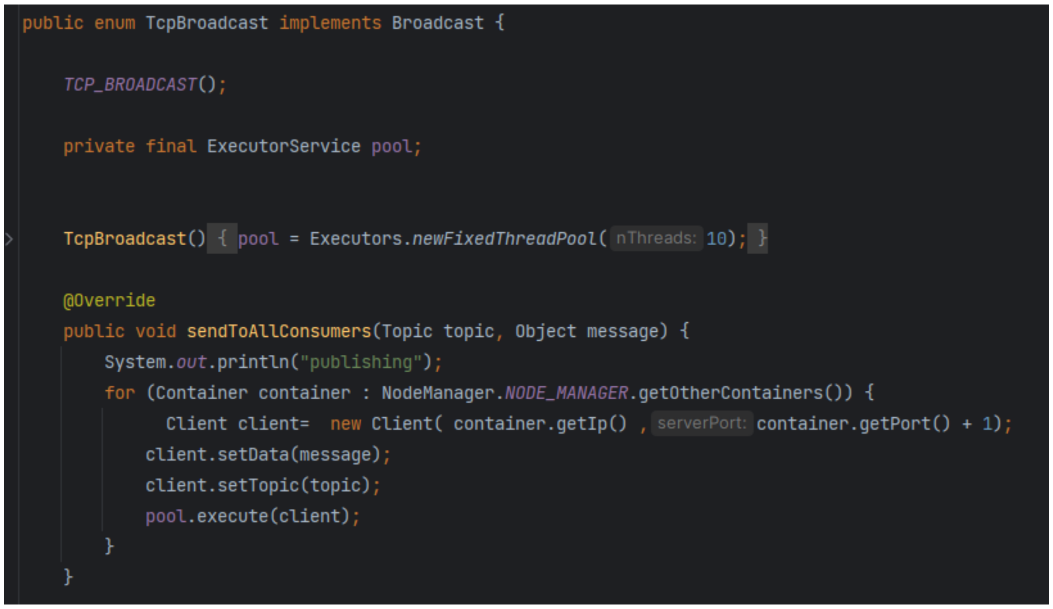
  I have utilized the Singleton Design Pattern extensively in my code as several objects are shared among multiple other objects.

  - The Singleton Design Pattern has been employed in the Server class, which has proven to be quite beneficial in this context, as we only require a single instance of the Server to listen for incoming network connections. Furthermore, by utilizing the Singleton Design Pattern, we can guarantee the existence of only one instance of the Server.



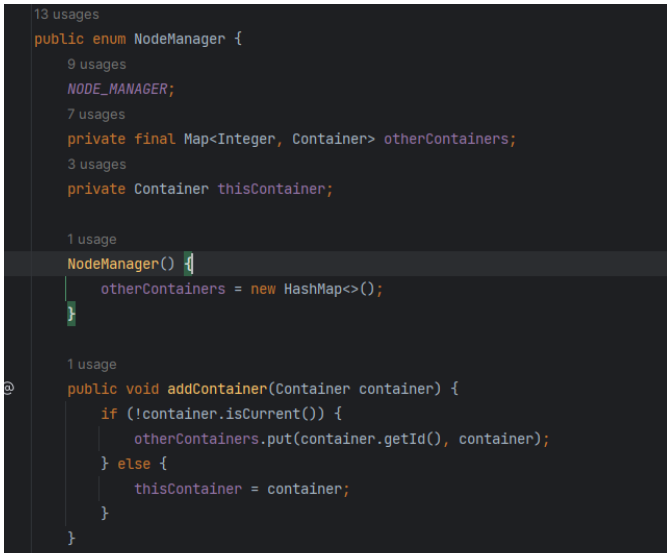
```
7 usages  new *
public class TcpServer implements Runnable {
    3 usages
    private static volatile TcpServer tcpServer;
    3 usages
    private  RequestHandler handler1;
    3 usages
    private final int PORT;
    3 usages
    private boolean isRunning;


    1 usage  new *
    private TcpServer(int PORT) {
        this.PORT = PORT;
        handler1 = new CommandHandler();
        RequestHandler handler2 = new ContainerHandler();
        RequestHandler handler3 = new UsersHandler();
        handler1.setNextHandler(handler2);
        handler2.setNextHandler(handler3);
    }

    new *
    public static TcpServer of(int port) {
        if (tcpServer == null) {
            synchronized (TcpServer.class) {
                tcpServer = new TcpServer(port);
            }
        }
        return tcpServer;
    }
```

- In the case of the TcpBroadcast class, I have implemented the Singleton Design Pattern to prevent unnecessary network overhead. By making TcpBroadcast a singleton, each node is limited to only one method of communication with the other nodes.



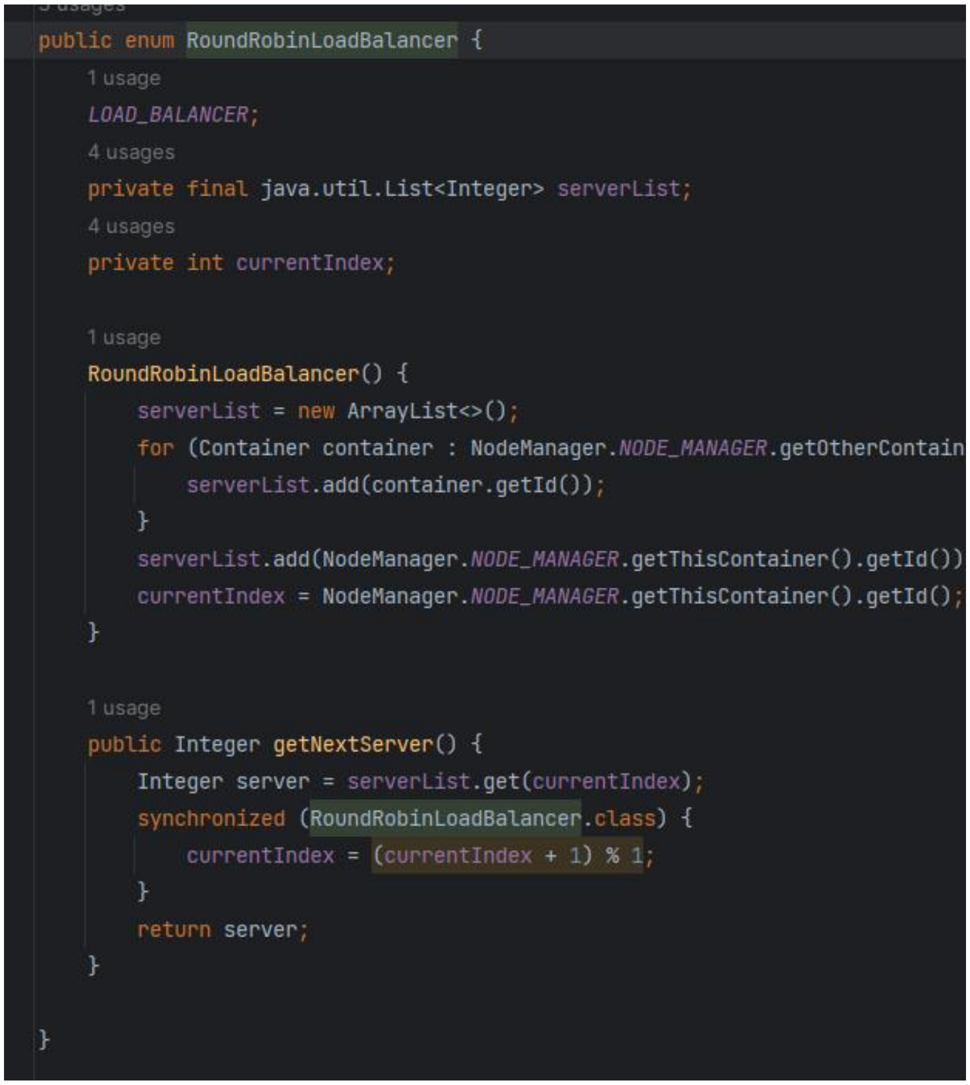
```java
public enum TcpBroadcast implements Broadcast {

    TCP_BROADCAST();

    private final ExecutorService pool;


    TcpBroadcast() { pool = Executors.newFixedThreadPool( nThreads: 10); }

    @Override
    public void sendToAllConsumers(Topic topic, Object message) {
        System.out.println("publishing");
        for (Container container : NodeManager.NODE_MANAGER.getOtherContainers()) {
              Client client=  new Client( container.getIp() , serverPort: container.getPort() + 1);
            client.setData(message);
            client.setTopic(topic);
            pool.execute(client);
        }
    }
```

- The NodeManager class stores information regarding the nodes as well as other relevant node-related data.



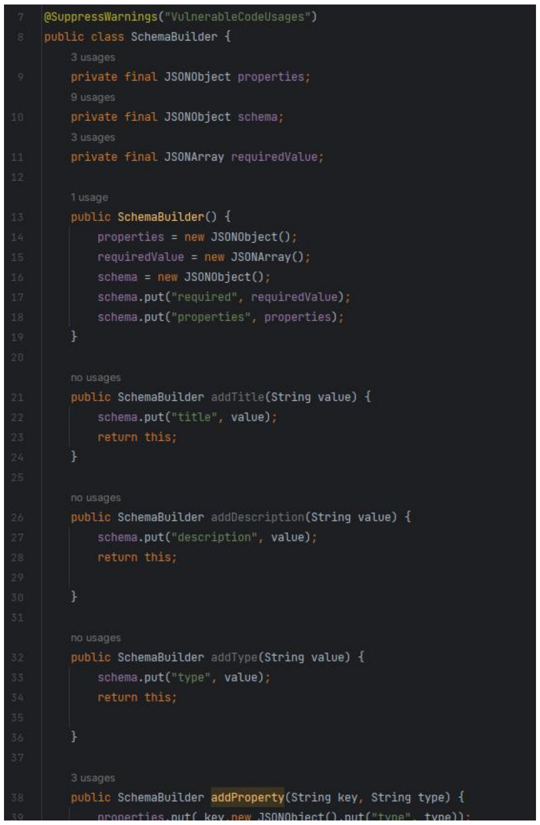
```java
13 usages
public enum NodeManager {
    9 usages
    NODE_MANAGER;
    7 usages
    private final Map<Integer, Container> otherContainers;
    3 usages
    private Container thisContainer;

    1 usage
    NodeManager() {
        otherContainers = new HashMap<>();
    }

    1 usage
    public void addContainer(Container container) {
        if (!container.isCurrent()) {
            otherContainers.put(container.getId(), container);
        } else {
            thisContainer = container;
        }
    }
```

- I utilized a Singleton pattern in the RoundRobinLoadBalancer class to ensure that all thread on the node share the same load balancer. I will provide more details about this in the upcoming chapters.



```java
public enum RoundRobinLoadBalancer {
    1 usage
    LOAD_BALANCER;
    4 usages
    private final java.util.List<Integer> serverList;
    4 usages
    private int currentIndex;

    1 usage
    RoundRobinLoadBalancer() {
        serverList = new ArrayList<>();
        for (Container container : NodeManager.NODE_MANAGER.getOtherContain
            serverList.add(container.getId());
        }
        serverList.add(NodeManager.NODE_MANAGER.getThisContainer().getId())
        currentIndex = NodeManager.NODE_MANAGER.getThisContainer().getId();
    }

    1 usage
    public Integer getNextServer() {
        Integer server = serverList.get(currentIndex);
        synchronized (RoundRobinLoadBalancer.class) {
            currentIndex = (currentIndex + 1) % 1;
        }
        return server;
    }

}
```

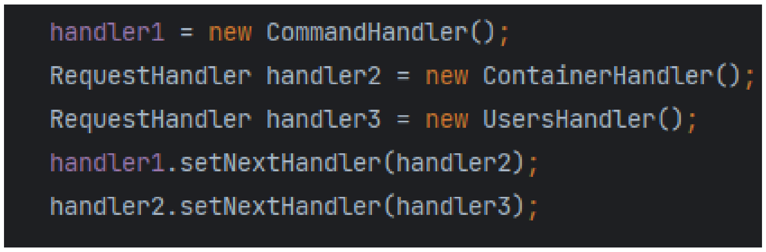
- Builder design pattern:
    - schema builder : I have employed the Builder Design Pattern in conjunction with a schema to overcome uncertainty regarding the number of properties contained within the schema. The Builder Design Pattern proved to be invaluable in this regard.



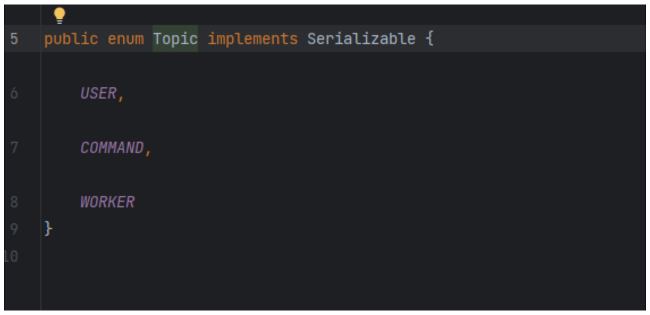
```
@SuppressWarnings("VulnerableCodeUsages")
public class SchemaBuilder {
    3 usages
    private final JSONObject properties;
    9 usages
    private final JSONObject schema;
    3 usages
    private final JSONArray requiredValue;

    1 usage
    public SchemaBuilder() {
        properties = new JSONObject();
        requiredValue = new JSONArray();
        schema = new JSONObject();
        schema.put("required", requiredValue);
        schema.put("properties", properties);
    }

    no usages
    public SchemaBuilder addTitle(String value) {
        schema.put("title", value);
        return this;
    }

    no usages
    public SchemaBuilder addDescription(String value) {
        schema.put("description", value);
        return this;

    }

    no usages
    public SchemaBuilder addType(String value) {
        schema.put("type", value);
        return this;

    }

    3 usages
    public SchemaBuilder addProperty(String key, String type) {
        properties.put( key,new JSONObject().put("type", type));
```

4.2 Behavioral design patterns:

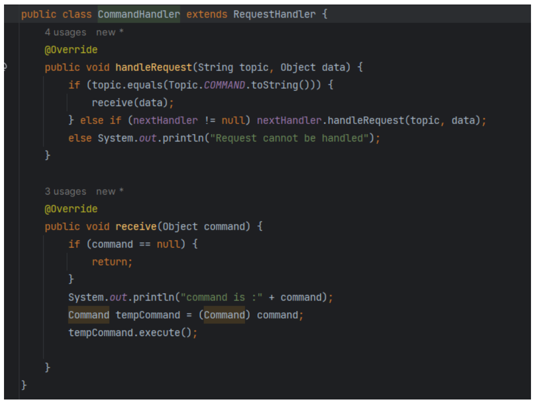
- chain of responsibility: I utilized the Chain of Responsibility design pattern with the server to enable it to handle multiple topics based on their respective types.



```java
handler1 = new CommandHandler();
RequestHandler handler2 = new ContainerHandler();
RequestHandler handler3 = new UsersHandler();
handler1.setNextHandler(handler2);
handler2.setNextHandler(handler3);
```



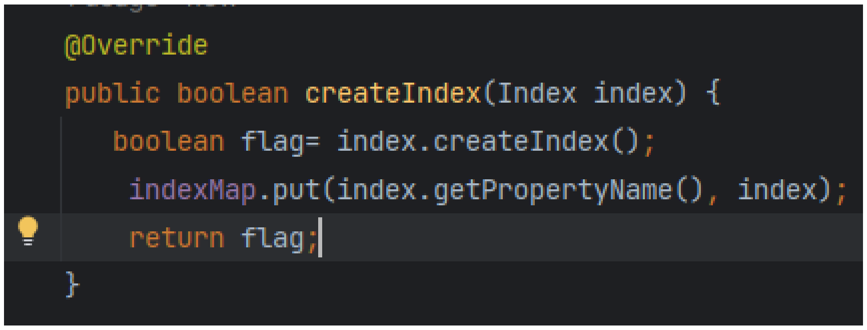
```java
public enum Topic implements Serializable {

    USER,

    COMMAND,

    WORKER
}
```



```java
public class CommandHandler extends RequestHandler {
    4 usages  new *
    @Override
    public void handleRequest(String topic, Object data) {
        if (topic.equals(Topic.COMMAND.toString())) {
            receive(data);
        } else if (nextHandler != null) nextHandler.handleRequest(topic, data);
        else System.out.println("Request cannot be handled");
    }

    3 usages  new *
    @Override
    public void receive(Object command) {
        if (command == null) {
            return;
        }
        System.out.println("command is :" + command);
        Command tempCommand = (Command) command;
        tempCommand.execute();

    }
}
```

- strategy design pattern : I employ the strategy design pattern in conjunction with broadcasting. This approach offers flexibility, as modifying the broadcast implementation is effortless. The broadcast itself serves as a supertype, ensuring compatibility regardless of the constructor's parameters.

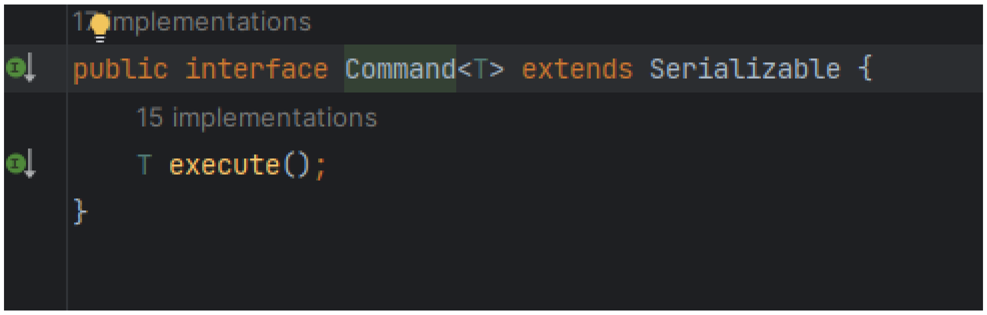
I have employed the Strategy design pattern with collection indices to ensure that the collection can work with any index class. The collection is not bound to a specific concrete index class, and any index passed to it should work seamlessly.



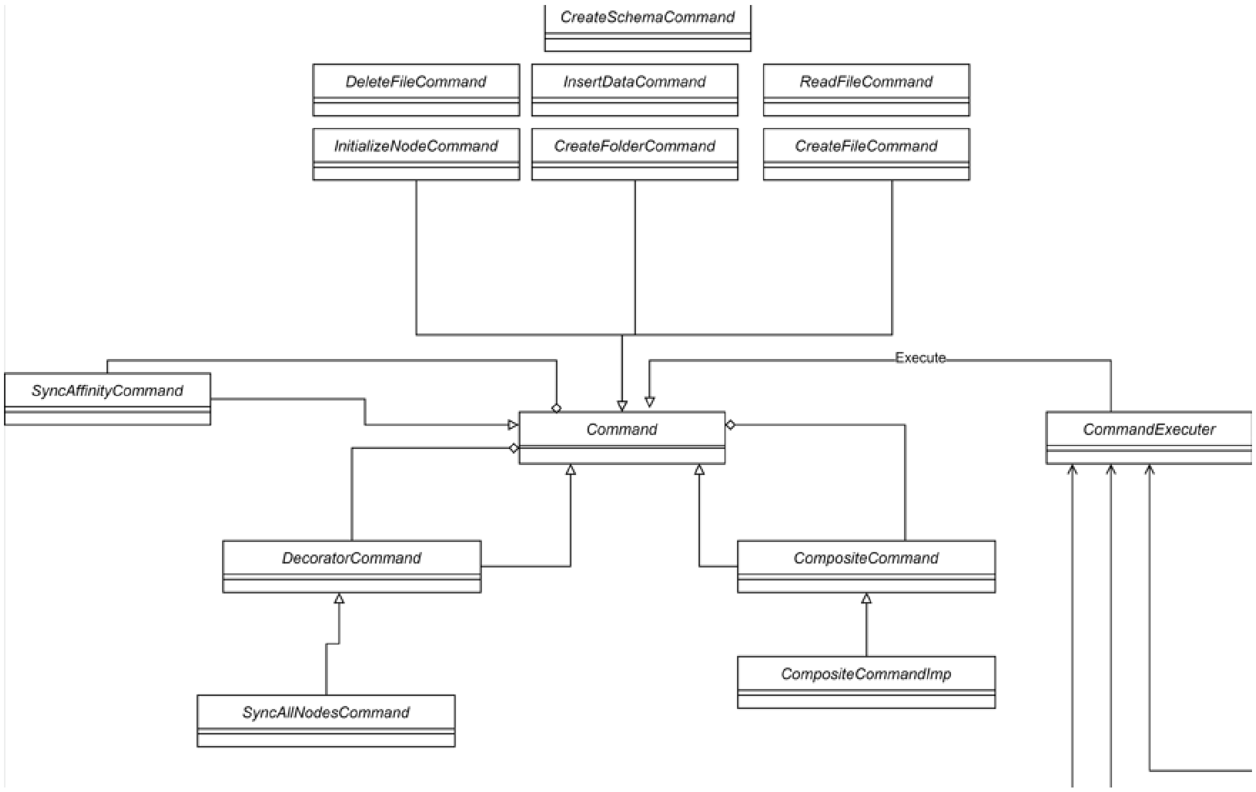
```java
@Override
public boolean createIndex(Index index) {
    boolean flag= index.createIndex();
     indexMap.put(index.getPropertyName(), index);
     return flag;
}
```

- Command design pattern:

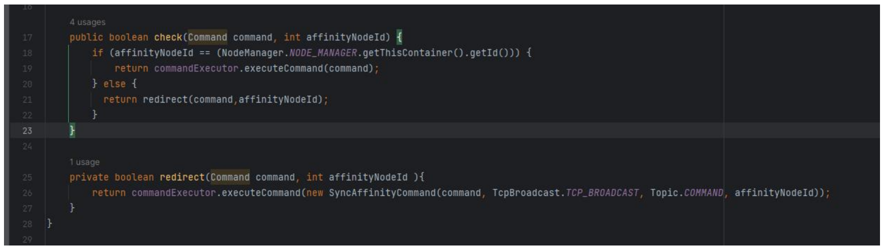
 the command provides flexibility to encapsulate the request, making it easier to maintain and send to other nodes. It is highly efficient to utilize.



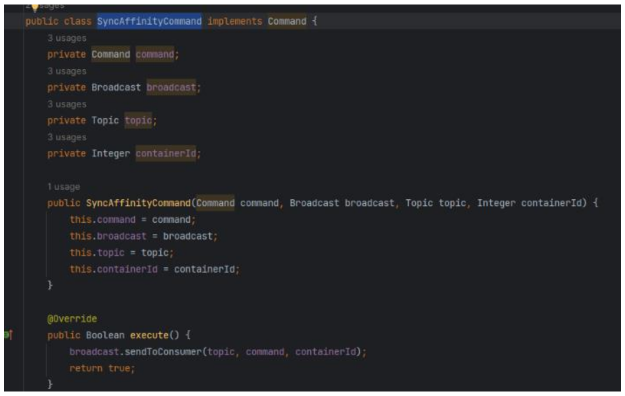
```java
17 implementations
public interface Command<T> extends Serializable {
    15 implementations
    T execute();
}
```



I would like to clarify the purpose of the "SyncAffinityCommand" feature, as it may be a bit confusing. Essentially, this command takes any given command and sends it to the appropriate affinity node for execution. I typically use it in conjunction with a decorator, particularly when I intend to execute a write query. Before executing the query, I check if the current node is the appropriate affinity for the associated collection, schema, or index. If it is not, then the command is simply passed to the affinity node for execution. The affinity node will then execute the command and broadcast it to the other nodes.



```
    4 usages
    public boolean check(Command command, int affinityNodeId) {
        if (affinityNodeId == (NodeManager.NODE_MANAGER.getThisContainer().getId())) {
            return commandExecutor.executeCommand(command);
        } else {
          return redirect(command,affinityNodeId);
        }
    }

    1 usage
    private boolean redirect(Command command, int affinityNodeId ){
        return commandExecutor.executeCommand(new SyncAffinityCommand(command, TcpBroadcast.TCP_BROADCAST, Topic.COMMAND, affinityNodeId));
    }
}
```

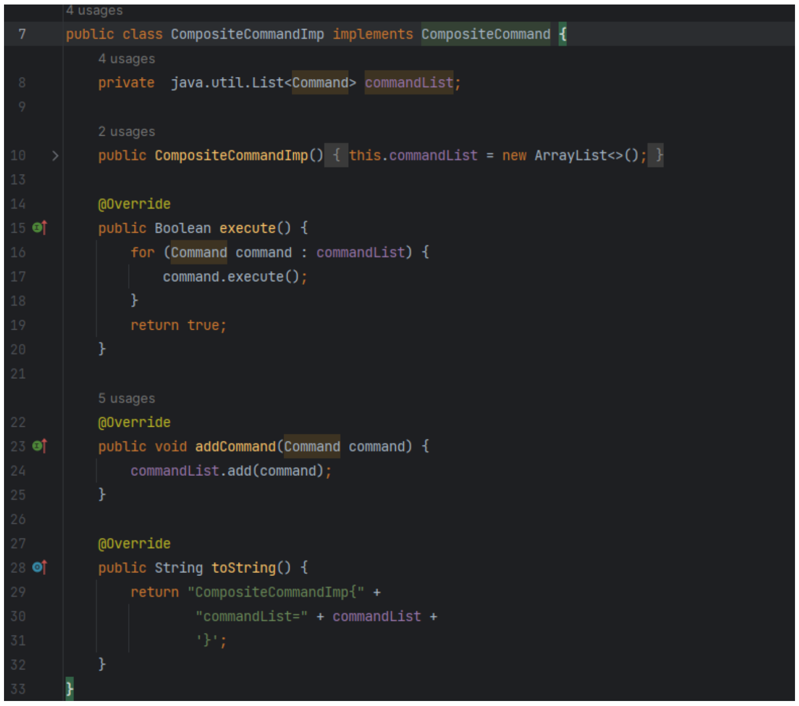


```
public class SyncAffinityCommand implements Command {
    3 usages
    private Command command;
    3 usages
    private Broadcast broadcast;
    3 usages
    private Topic topic;
    3 usages
    private Integer containerId;

    1 usage
    public SyncAffinityCommand(Command command, Broadcast broadcast, Topic topic, Integer containerId) {
        this.command = command;
        this.broadcast = broadcast;
        this.topic = topic;
        this.containerId = containerId;
    }

    @Override
    public Boolean execute() {
        broadcast.sendToConsumer(topic, command, containerId);
        return true;
    }
```

4.3 structural design patterns:

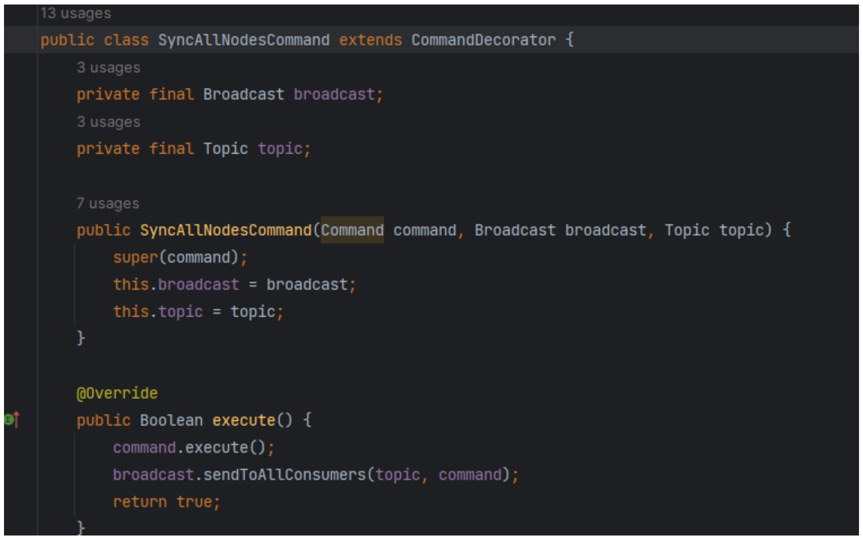
- Façade : As mentioned earlier, the Façade serves as the entry point for user interaction with the database. It employs the Façade pattern to conceal the intricacies of the system and furnish a user interface for the client to utilize the system.

- Composite: Utilizing the Composite design pattern in conjunction with the Command design pattern, I am able to add a group of commands and execute them simultaneously. The Composite design pattern enables me to accomplish this by combining multiple objects into a single object, allowing me to treat the group of commands as a cohesive unit.



```java
4 usages
public class CompositeCommandImp implements CompositeCommand {
    4 usages
    private  java.util.List<Command> commandList;

    2 usages
    public CompositeCommandImp() { this.commandList = new ArrayList<>(); }

    @Override
    public Boolean execute() {
        for (Command command : commandList) {
            command.execute();
        }
        return true;
    }

    5 usages
    @Override
    public void addCommand(Command command) {
        commandList.add(command);
    }

    @Override
    public String toString() {
        return "CompositeCommandImp{" +
                "commandList=" + commandList +
                '}';
    }
}
```

- Decorator design pattern : By employing the Decorator design pattern in conjunction with the Command design pattern, I am able to enhance the functionality of a command and use it for broadcasting. For instance, if an insert command needs to be broadcasted to other nodes, I can execute the command by adding it to the Decorator, which will then broadcast it to the other nodes.

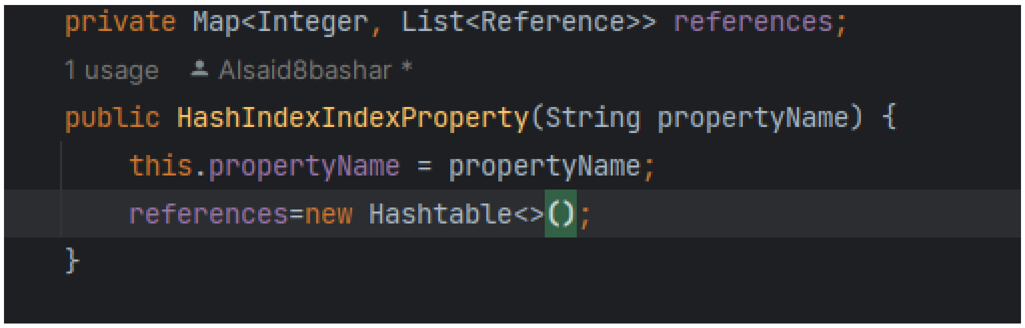


```
13 usages
public class SyncAllNodesCommand extends CommandDecorator {
    3 usages
    private final Broadcast broadcast;
    3 usages
    private final Topic topic;

    7 usages
    public SyncAllNodesCommand(Command command, Broadcast broadcast, Topic topic) {
        super(command);
        this.broadcast = broadcast;
        this.topic = topic;
    }

    @Override
    public Boolean execute() {
        command.execute();
        broadcast.sendToAllConsumers(topic, command);
        return true;
    }
```

5.0 The Data structures used:

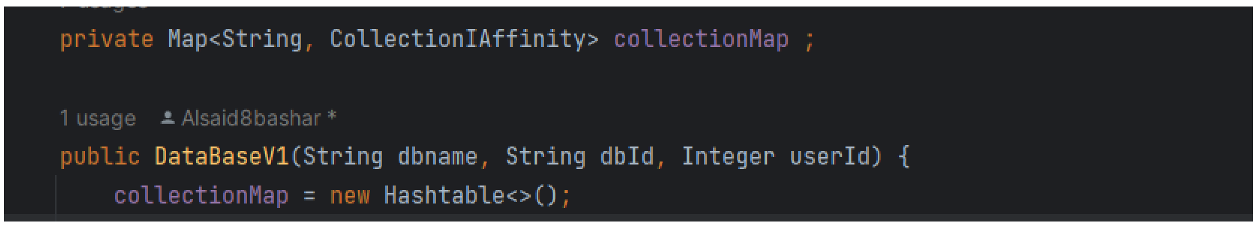
Data structures are fundamental components of computer science, used to store and organize data in a way that makes it easy to access, modify and manipulate. Each data structure has its own unique characteristics that make it suitable for different types of applications. In this chapter, we will focus on a specific data structure that I have used in my programming projects.

1- I opted for using a hash table with hash index due to its thread-safety feature, enabling it to be accessed by multiple threads simultaneously without risking data corruption. Additionally, the hash table disallows null values for both keys and values.



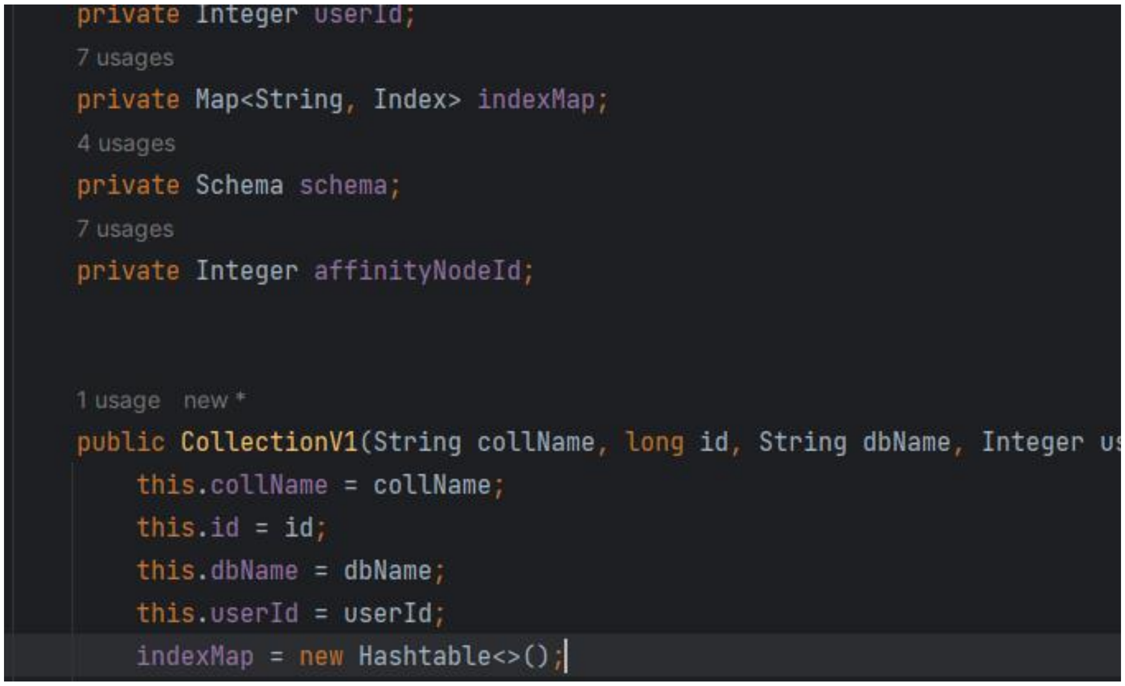
```
private Map<Integer, List<Reference>> references;
1 usage   Alsaid8bashar *
public HashIndexIndexProperty(String propertyName) {
    this.propertyName = propertyName;
    references=new Hashtable<>();
}
```

2- In the database class, I employed a hashtable with the same purpose, where it holds a map of collections.



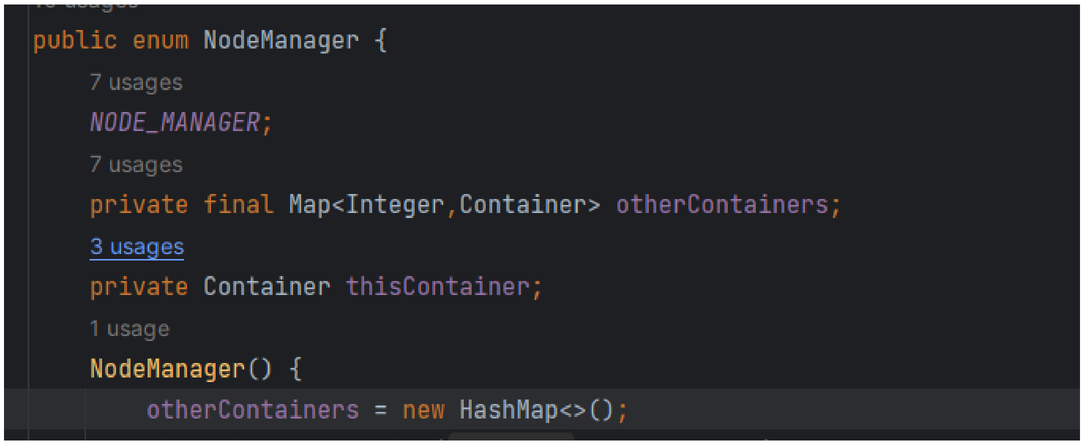
```
private Map<String, CollectionIAffinity> collectionMap ;

1 usage   Alsaid8bashar *
public DataBaseV1(String dbname, String dbId, Integer userId) {
    collectionMap = new Hashtable<>();
```

also on the Collcetion class I have hashTable



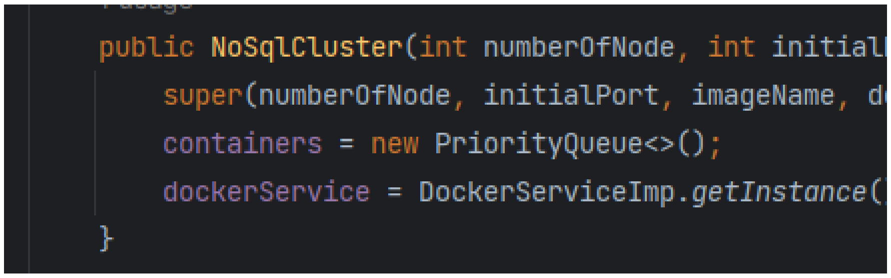
```
    private Integer userId;
    7 usages
    private Map<String, Index> indexMap;
    4 usages
    private Schema schema;
    7 usages
    private Integer affinityNodeId;


    1 usage   new *
    public CollectionV1(String collName, long id, String dbName, Integer us
        this.collName = collName;
        this.id = id;
        this.dbName = dbName;
        this.userId = userId;
        indexMap = new Hashtable<>();
```

4- I used a HashMap in the NodeManager class because it is already thread-safe as a Singleton using an Enum, and it is faster than a Hashtable. Since Hashtable is not thread-safe, I made the decision to use a HashMap instead.



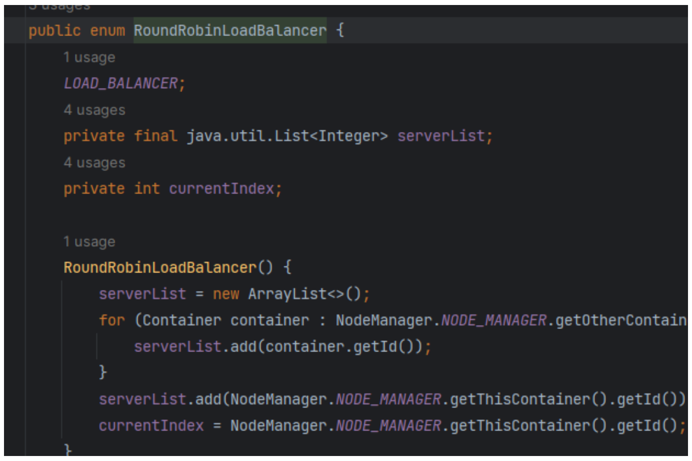
```
public enum NodeManager {
    7 usages
    NODE_MANAGER;
    7 usages
    private final Map<Integer,Container> otherContainers;
    3 usages
    private Container thisContainer;
    1 usage
    NodeManager() {
        otherContainers = new HashMap<>();
```

5-To ensure load balancing, I utilized a priority queue in the BootstrappingNode to store containers. I will explain this in more detail in the following chapters,



```java
public NoSqlCluster(int numberOfNode, int initialI
    super(numberOfNode, initialPort, imageName, d
    containers = new PriorityQueue<>();
    dockerService = DockerServiceImp.getInstance(
}
```

6- To efficiently maintain load balancing affinity with nodes, we utilized an Array List to store container IDs.



```java
public enum RoundRobinLoadBalancer {
    1 usage
    LOAD_BALANCER;
    4 usages
    private final java.util.List<Integer> serverList;
    4 usages
    private int currentIndex;

    1 usage
    RoundRobinLoadBalancer() {
        serverList = new ArrayList<>();
        for (Container container : NodeManager.NODE_MANAGER.getOtherContain
            serverList.add(container.getId());
        }
        serverList.add(NodeManager.NODE_MANAGER.getThisContainer().getId())
        currentIndex = NodeManager.NODE_MANAGER.getThisContainer().getId();
    }
```

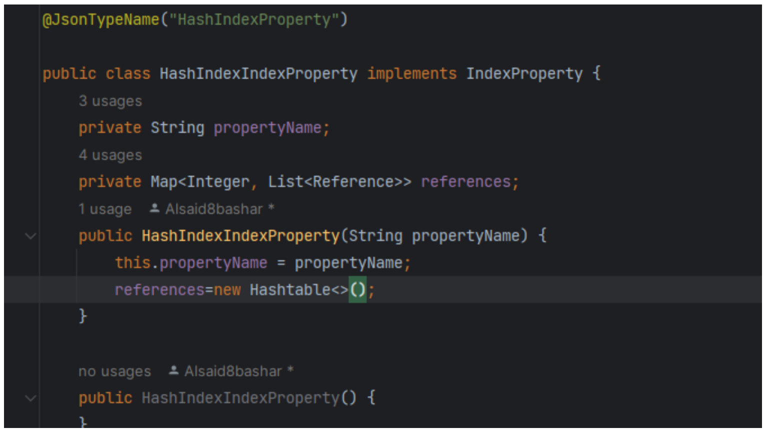
6.0 Indexing:

In this chapter, I will discuss how I implemented indexing in my project. Initially, I explored various options for indexing, but I ultimately decided to implement the index from scratch without using any existing source code. I researched several indexing techniques, such as B-trees, B+ trees, AVL trees, and hash indexes. Ultimately, I opted for the hash index because, as I mentioned earlier, I wanted to implement the index from scratch. Additionally, I found the hash index to be highly efficient for this project's requirements because it only needed to read and write specific JSON properties. It did not require complex logical operations such as (OR, AND, less than, greater than), which makes the hash index suitable for this use case. Overall, I believe that the hash index was a good choice for this type of project.

When the "createIndex" function is called , the index class takes charge of generating the index. It accomplishes this by utilizing the "HashIndexIndexProperty" and "HashIndexReference" classes and then persisting the result to the disk.

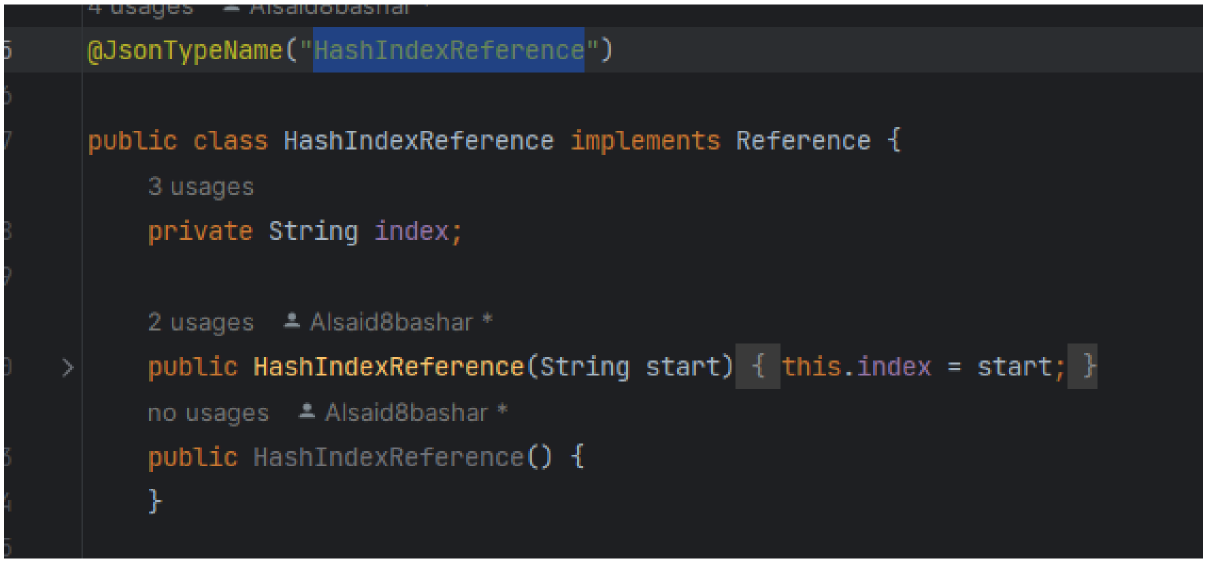
3.1-index component :

- The HashIndexIndexProperty class contains a map and a list of references for each key.



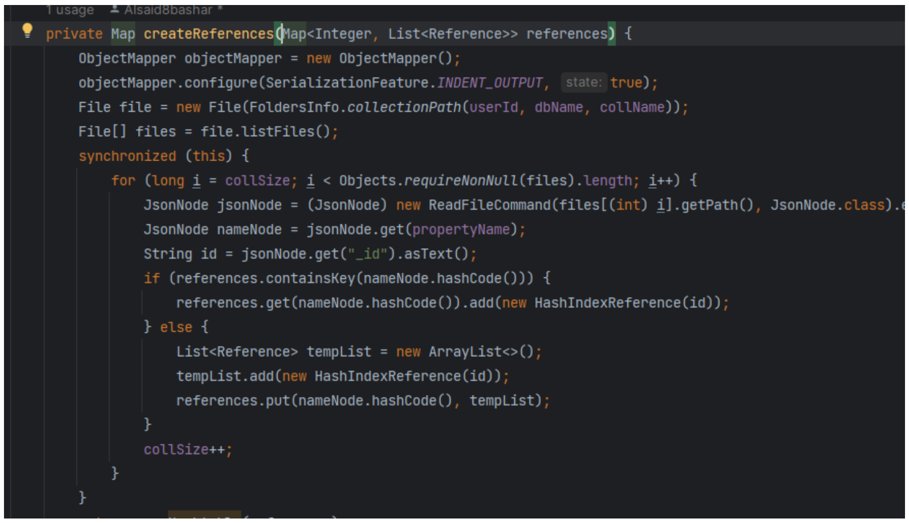
```
@JsonTypeName("HashIndexProperty")

public class HashIndexIndexProperty implements IndexProperty {
    3 usages
    private String propertyName;
    4 usages
    private Map<Integer, List<Reference>> references;
    1 usage   Alsaid8bashar *
    public HashIndexIndexProperty(String propertyName) {
        this.propertyName = propertyName;
        references=new Hashtable<>();
    }

    no usages   Alsaid8bashar *
    public HashIndexIndexProperty() {
    }
```

- The HashIndexReference class serves as a container for document references, which essentially function as file names on the disk.



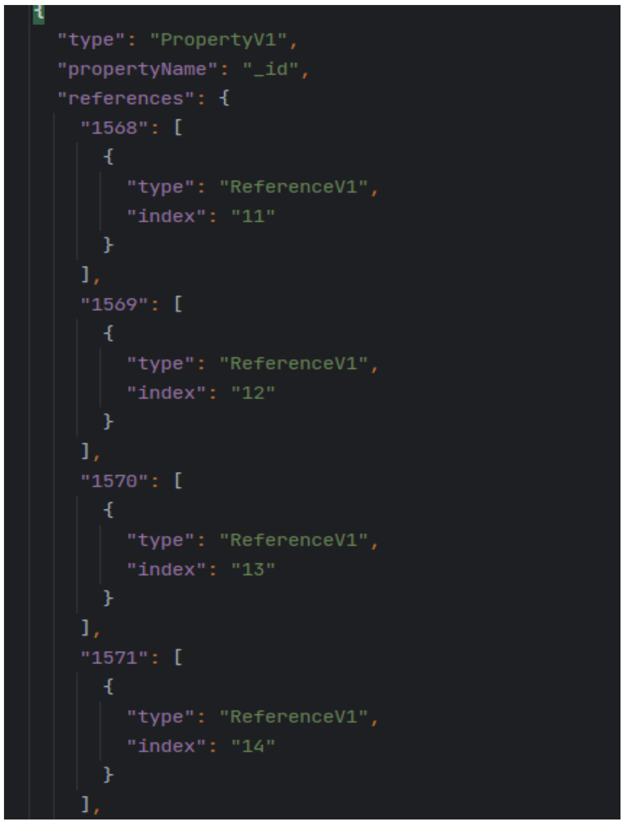
```java
4 usages   Alsaid8bashar *
@JsonTypeName("HashIndexReference")

public class HashIndexReference implements Reference {
    3 usages
    private String index;

    2 usages   Alsaid8bashar *
    public HashIndexReference(String start) { this.index = start; }
    no usages   Alsaid8bashar *
    public HashIndexReference() {
    }
```

The process of creating an index in the hash index class involves using a specific function. When this function is called, the createReference function is executed first. It searches for the folder where the collection of documents is stored and iterates through every file in the folder. During this iteration, the createReference function creates an index for the "_id" property of each JSON document, which serves as the field ID in the folder. Once the indexing is complete, another function is called to save the index in the folder and broadcast it to other nodes.



```java
1 usage   Alsaid8bashar *
private Map createReferences(Map<Integer, List<Reference>> references) {
    ObjectMapper objectMapper = new ObjectMapper();
    objectMapper.configure(SerializationFeature.INDENT_OUTPUT, state: true);
    File file = new File(FoldersInfo.collectionPath(userId, dbName, collName));
    File[] files = file.listFiles();
    synchronized (this) {
        for (long i = collSize; i < Objects.requireNonNull(files).length; i++) {
            JsonNode jsonNode = (JsonNode) new ReadFileCommand(files[(int) i].getPath(), JsonNode.class).
            JsonNode nameNode = jsonNode.get(propertyName);
            String id = jsonNode.get("_id").asText();
            if (references.containsKey(nameNode.hashCode())) {
                references.get(nameNode.hashCode()).add(new HashIndexReference(id));
            } else {
                List<Reference> tempList = new ArrayList<>();
                tempList.add(new HashIndexReference(id));
                references.put(nameNode.hashCode(), tempList);
            }
            collSize++;
        }
    }
```

This is how the index should appear on the disk.

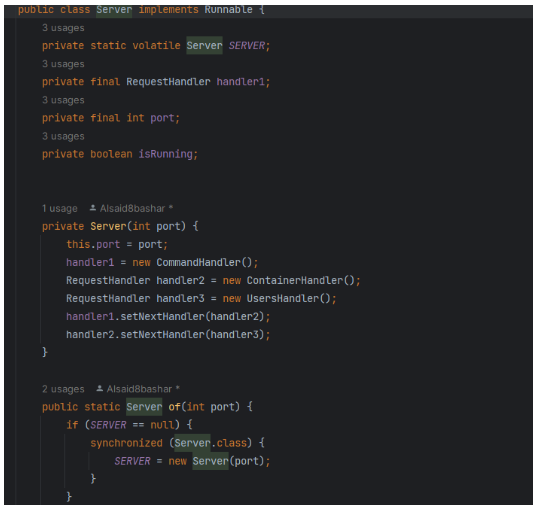


```
{
  "type": "PropertyV1",
  "propertyName": "_id",
  "references": {
    "1568": [
      {
        "type": "ReferenceV1",
        "index": "11"
      }
    ],
    "1569": [
      {
        "type": "ReferenceV1",
        "index": "12"
      }
    ],
    "1570": [
      {
        "type": "ReferenceV1",
        "index": "13"
      }
    ],
    "1571": [
      {
        "type": "ReferenceV1",
        "index": "14"
      }
    ],
```

7.0 Communication protocols between nodes

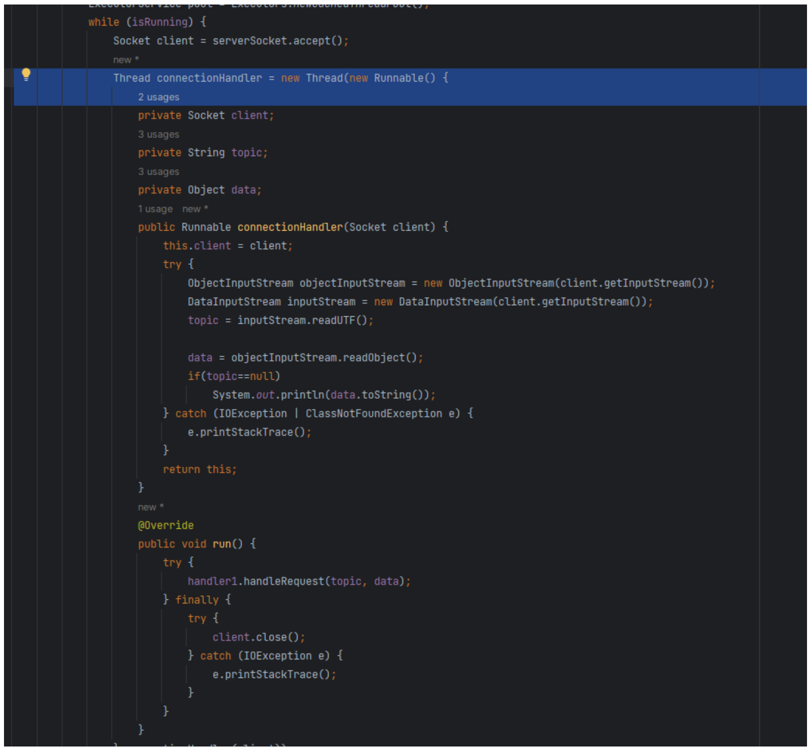
I have implemented Java socket and Java server socket to enable communication between nodes in my system. Each node has its own TCP server to communicate with other nodes. While TCP is a reliable choice, I believe that UDP could be a better option as it avoids the overhead of TCP on the network, especially when there is no multicasting. Nevertheless, TCP still does the job efficiently. As previously mentioned, each node has its own server to communicate with other nodes. When a node wants to communicate with other nodes, I create a client socket for each node on the network and send the topic and message with it.

1- The server in my system implements the Runnable interface to prevent it from blocking the main thread. Additionally, it has a cached thread pool, which creates a new thread for every client that connects to the server to handle their requests. As mentioned earlier, the server is a Singleton class that should only run once. In the Design Patterns section, I discussed using the Chain of Responsibility design pattern to handle client requests



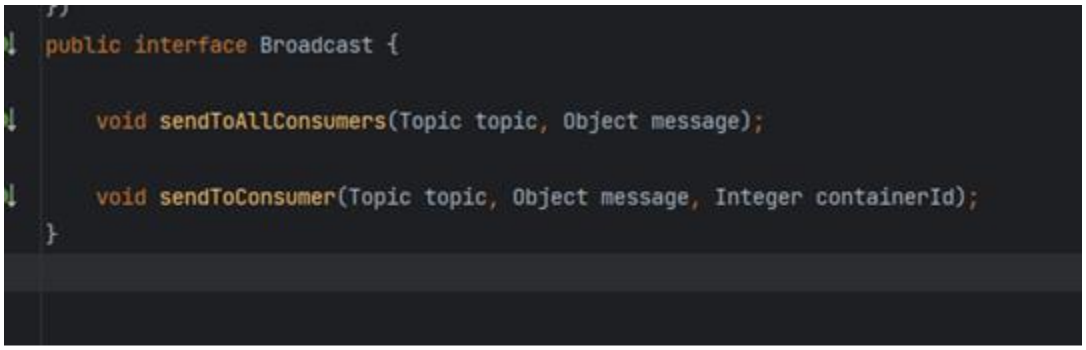
```java
public class Server implements Runnable {
    3 usages
    private static volatile Server SERVER;
    3 usages
    private final RequestHandler handler1;
    3 usages
    private final int port;
    3 usages
    private boolean isRunning;


    1 usage   Alsaid8bashar *
    private Server(int port) {
        this.port = port;
        handler1 = new CommandHandler();
        RequestHandler handler2 = new ContainerHandler();
        RequestHandler handler3 = new UsersHandler();
        handler1.setNextHandler(handler2);
        handler2.setNextHandler(handler3);
    }

    2 usages   Alsaid8bashar *
    public static Server of(int port) {
        if (SERVER == null) {
            synchronized (Server.class) {
                SERVER = new Server(port);
            }
        }
```

When a client attempts to connect, the server will create a thread to serve that client. I will discuss the beauty of anonymous classes in the thread section.



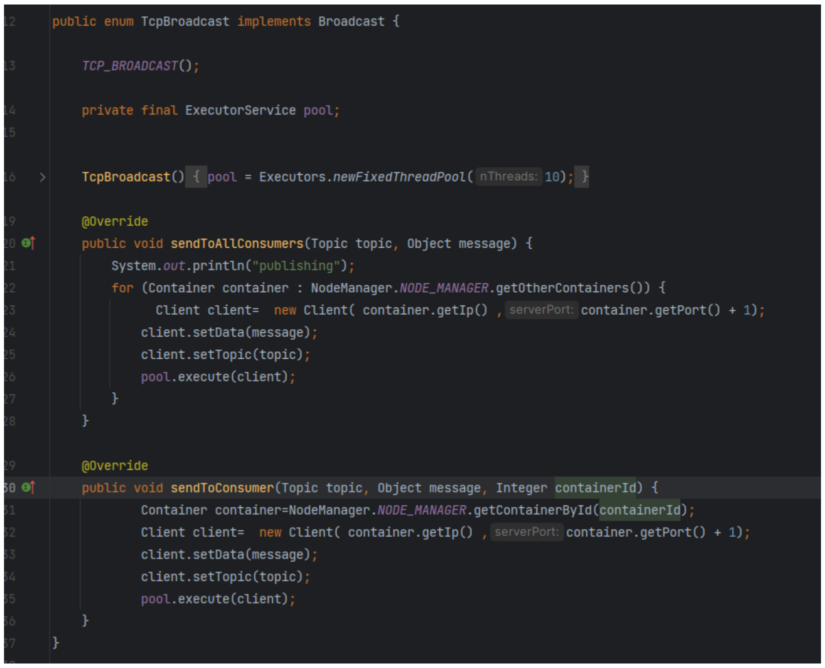
```java
        while (isRunning) {
            Socket client = serverSocket.accept();
            new *
            Thread connectionHandler = new Thread(new Runnable() {
                2 usages
                private Socket client;
                3 usages
                private String topic;
                3 usages
                private Object data;
                1 usage  new *
                public Runnable connectionHandler(Socket client) {
                    this.client = client;
                    try {
                        ObjectInputStream objectInputStream = new ObjectInputStream(client.getInputStream());
                        DataInputStream inputStream = new DataInputStream(client.getInputStream());
                        topic = inputStream.readUTF();

                        data = objectInputStream.readObject();
                        if(topic==null)
                            System.out.println(data.toString());
                    } catch (IOException | ClassNotFoundException e) {
                        e.printStackTrace();
                    }
                    return this;
                }
                new *
                @Override
                public void run() {
                    try {
                        handler1.handleRequest(topic, data);
                    } finally {
                        try {
                            client.close();
                        } catch (IOException e) {
                            e.printStackTrace();
                        }
                    }
                }
```

2- The Broadcast interface provides two functions:
  - sendToAllConsumers: Sends data to all nodes on the network.
  - sendToConsumer: Sends data to a specific node on the network.



```java
public interface Broadcast {

    void sendToAllConsumers(Topic topic, Object message);

    void sendToConsumer(Topic topic, Object message, Integer containerId);
}
```

A potential implementation of the concrete class is TcpBroadcast, which is a singleton class.



```java
public enum TcpBroadcast implements Broadcast {

    TCP_BROADCAST();

    private final ExecutorService pool;


    TcpBroadcast() { pool = Executors.newFixedThreadPool( nThreads: 10); }

    @Override
    public void sendToAllConsumers(Topic topic, Object message) {
        System.out.println("publishing");
        for (Container container : NodeManager.NODE_MANAGER.getOtherContainers()) {
              Client client=  new Client( container.getIp() , serverPort: container.getPort() + 1);
            client.setData(message);
            client.setTopic(topic);
            pool.execute(client);
        }
    }

    @Override
    public void sendToConsumer(Topic topic, Object message, Integer containerId) {
            Container container=NodeManager.NODE_MANAGER.getContainerById(containerId);
            Client client=  new Client( container.getIp() , serverPort: container.getPort() + 1);
            client.setData(message);
            client.setTopic(topic);
            pool.execute(client);
    }
}
```

8.0 Multithreading and locks

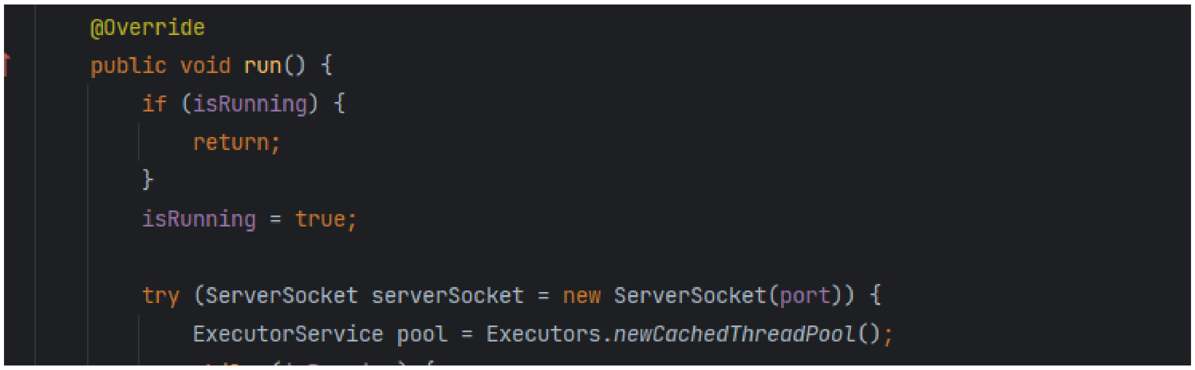
8.1 Multithreading:

As previously mentioned, I am using Spring as the communication layer in my project. Spring is inherently designed to support multithreading and generates a new thread for each incoming request. Also I have made some decisions to optimize the system further.

Thread pool : The implementation of a thread pool has been utilized in two key locations: the server and the TCP broadcast class. The design choices made for each of these implementations will be explained in detail below.

- Thread pool

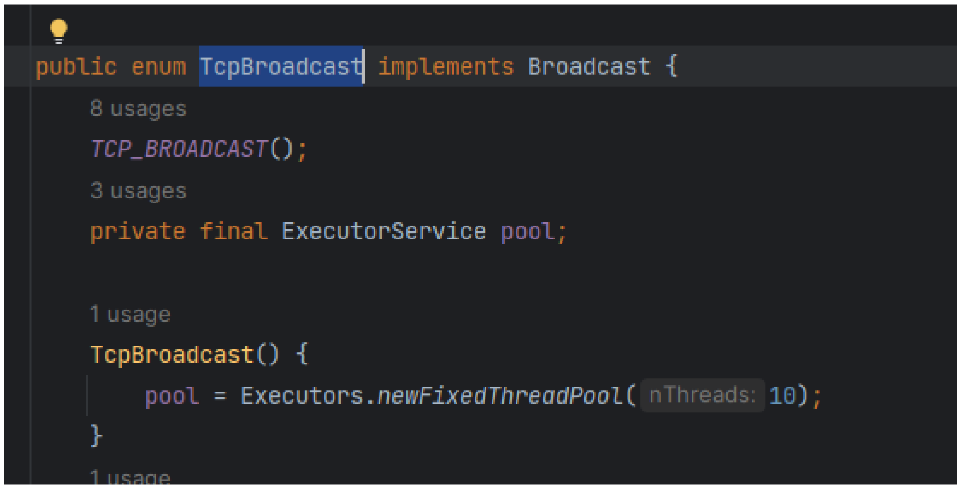
1- Server thread pool : on the server thread pool I create newCachedThreadPool , This was done to enable the server to handle multiple incoming client requests in a more efficient manner. The server creates a pool of worker threads, which can then be used to execute tasks in parallel. This allows the server to handle a larger number of requests simultaneously, reducing the overall response time and improving the system's performance.



```
@Override
public void run() {
    if (isRunning) {
        return;
    }
    isRunning = true;

    try (ServerSocket serverSocket = new ServerSocket(port)) {
        ExecutorService pool = Executors.newCachedThreadPool();
```

2- In addition to utilizing a thread pool in the server implementation, a fixed thread pool has also been implemented in the TCP class. The design decision was made to prevent the system from running an excessive number of threads and to optimize hardware resources.

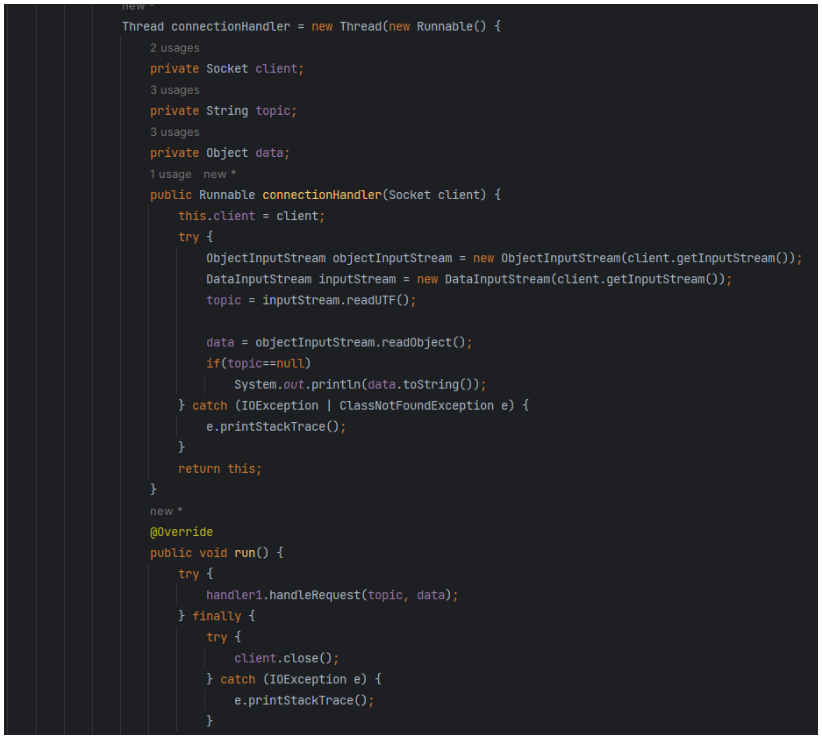
approach ensures that the number of threads used by the system remains constant, preventing the system from creating an unmanageable number of threads that could degrade the system's performance. This also ensures that hardware resources are used efficiently, preventing unnecessary strain on the system.



```java
public enum TcpBroadcast implements Broadcast {
    8 usages
    TCP_BROADCAST();
    3 usages
    private final ExecutorService pool;

    1 usage
    TcpBroadcast() {
        pool = Executors.newFixedThreadPool( nThreads: 10);
    }
    1 usage
```

- Anonymous class : In order to avoid the need for creating a class that can only be used once and in one specific location, I employed the concept of Anonymous classes to create an Anonymously named class that can handle the client.

By implementing this approach, the creation of a separate class is avoided, which can help to streamline the code and improve its readability. Additionally, it can help to minimize the risk of errors or conflicts that can arise from the use of multiple classes within the same codebase.

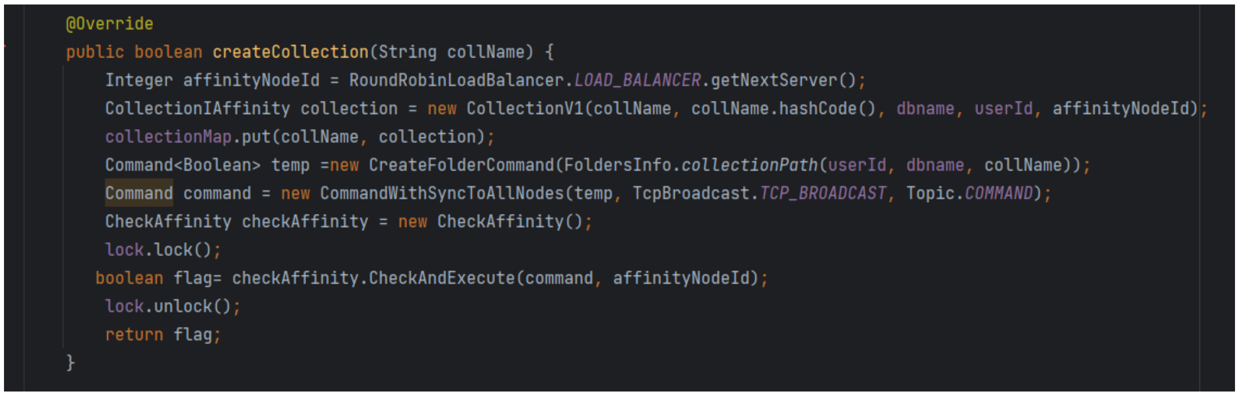


```java
Thread connectionHandler = new Thread(new Runnable() {
    2 usages
    private Socket client;
    3 usages
    private String topic;
    3 usages
    private Object data;
    1 usage  new *
    public Runnable connectionHandler(Socket client) {
        this.client = client;
        try {
            ObjectInputStream objectInputStream = new ObjectInputStream(client.getInputStream());
            DataInputStream inputStream = new DataInputStream(client.getInputStream());
            topic = inputStream.readUTF();

            data = objectInputStream.readObject();
            if(topic==null)
                System.out.println(data.toString());
        } catch (IOException | ClassNotFoundException e) {
            e.printStackTrace();
        }
        return this;
    }
    new *
    @Override
    public void run() {
        try {
            handler1.handleRequest(topic, data);
        } finally {
            try {
                client.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
```

8.2 Locks:

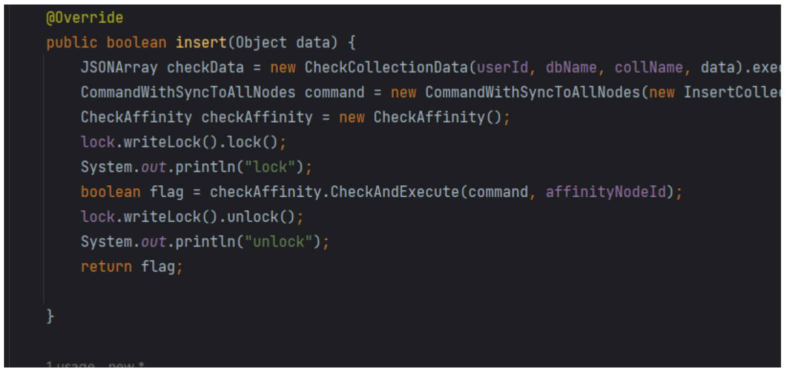
When designing a database, it is essential to ensure that both read and write operations are thread safe. This is because multiple threads may attempt to access the database simultaneously, which can result in conflicts and data inconsistencies. To address this issue, the ReadWriteLock feature in Java can be used to allow multiple threads to read from the database simultaneously, while ensuring that only one thread can write to the database at any given time. This approach ensures that the database remains consistent and prevents data corruption. By implementing ReadWriteLock in database design, developers can ensure that their applications are robust and efficient, even under heavy loads and concurrent access.

- To avoid race conditions when creating or deleting collections in the database, I utilized the ReentrantLock in my implementation. This allowed me to ensure that only one thread at a time could access the critical section of code that created or deleted the collection. By doing so, I was able to prevent data inconsistencies and conflicts that could arise from multiple threads attempting to access the database simultaneously. With the ReentrantLock in place, I could rest assured that my application was robust and could handle concurrent access without any issues.

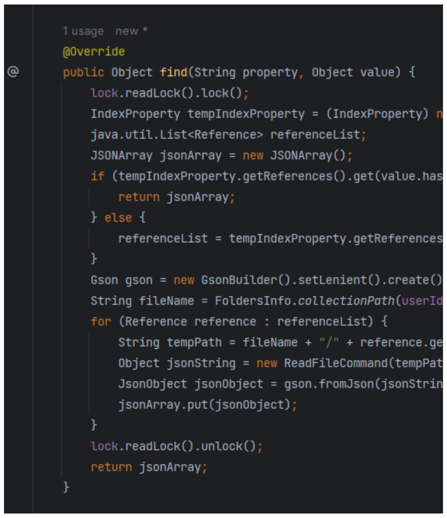


```java
@Override
public boolean createCollection(String collName) {
    Integer affinityNodeId = RoundRobinLoadBalancer.LOAD_BALANCER.getNextServer();
    CollectionIAffinity collection = new CollectionV1(collName, collName.hashCode(), dbname, userId, affinityNodeId);
    collectionMap.put(collName, collection);
    Command<Boolean> temp =new CreateFolderCommand(FoldersInfo.collectionPath(userId, dbname, collName));
    Command command = new CommandWithSyncToAllNodes(temp, TcpBroadcast.TCP_BROADCAST, Topic.COMMAND);
    CheckAffinity checkAffinity = new CheckAffinity();
    lock.lock();
   boolean flag= checkAffinity.CheckAndExecute(command, affinityNodeId);
    lock.unlock();
    return flag;
}
```

- Also on the collection side create lock on the read and insert data



```java
@Override
public boolean insert(Object data) {
    JSONArray checkData = new CheckCollectionData(userId, dbName, collName, data).exe
    CommandWithSyncToAllNodes command = new CommandWithSyncToAllNodes(new InsertColle
    CheckAffinity checkAffinity = new CheckAffinity();
    lock.writeLock().lock();
    System.out.println("lock");
    boolean flag = checkAffinity.CheckAndExecute(command, affinityNodeId);
    lock.writeLock().unlock();
    System.out.println("unlock");
    return flag;

}
```

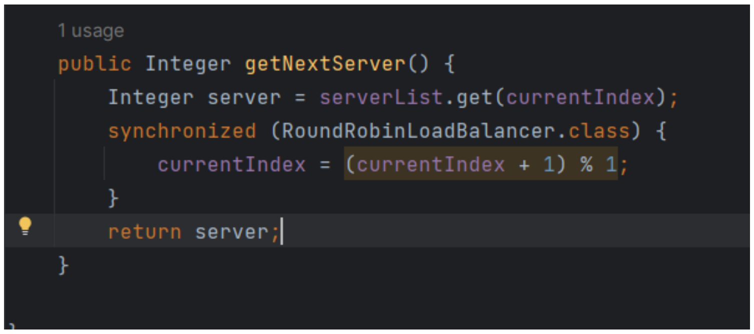


```java
1 usage  new *
@Override
public Object find(String property, Object value) {
    lock.readLock().lock();
    IndexProperty tempIndexProperty = (IndexProperty) n
    java.util.List<Reference> referenceList;
    JSONArray jsonArray = new JSONArray();
    if (tempIndexProperty.getReferences().get(value.has
        return jsonArray;
    } else {
        referenceList = tempIndexProperty.getReferences
    }
    Gson gson = new GsonBuilder().setLenient().create()
    String fileName = FoldersInfo.collectionPath(userId
    for (Reference reference : referenceList) {
        String tempPath = fileName + "/" + reference.ge
        Object jsonString = new ReadFileCommand(tempPat
        JsonObject jsonObject = gson.fromJson(jsonStrin
        jsonArray.put(jsonObject);
    }
    lock.readLock().unlock();
    return jsonArray;
}
```

Similar to how I implemented a lock on creating and deleting collections in the database, I also utilized a lock on both read and write operations on the index and schema. This allowed me to ensure that only one thread at a time could read or write to the same object, preventing any conflicts or data inconsistencies that could arise from concurrent access. By implementing this locking mechanism, I could ensure that my application was thread-safe and able to handle multiple requests in a robust and efficient manner

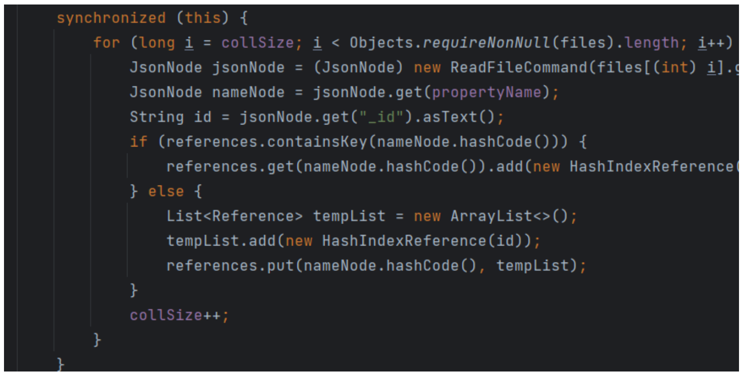
- Synchronized key word : I used the synchronized keyword in various places to ensure that the system operates correctly. This keyword helps to prevent multiple threads from concurrently accessing the same code block, thereby avoiding potential race conditions and ensuring thread safety.

    1 on the LoadBalancer class



```
1 usage
public Integer getNextServer() {
    Integer server = serverList.get(currentIndex);
    synchronized (RoundRobinLoadBalancer.class) {
        currentIndex = (currentIndex + 1) % 1;
    }
    return server;
}
```

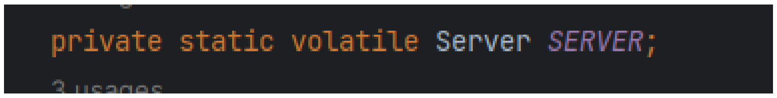
    2 When creating an index, it's important to take the collection size into account. This is because if new data is inserted, the index should continue from where it left off to avoid having to reindex the entire collection. To ensure thread safety, it's important to make sure that the increment of the collection size is done in a thread-safe manner, so that there are no conflicts between multiple threads trying to modify the size at the same time.



```
synchronized (this) {
    for (long i = collSize; i < Objects.requireNonNull(files).length; i++)
        JsonNode jsonNode = (JsonNode) new ReadFileCommand(files[(int) i].
        JsonNode nameNode = jsonNode.get(propertyName);
        String id = jsonNode.get("_id").asText();
        if (references.containsKey(nameNode.hashCode())) {
            references.get(nameNode.hashCode()).add(new HashIndexReference
        } else {
            List<Reference> tempList = new ArrayList<>();
            tempList.add(new HashIndexReference(id));
            references.put(nameNode.hashCode(), tempList);
        }
        collSize++;
    }
}
```

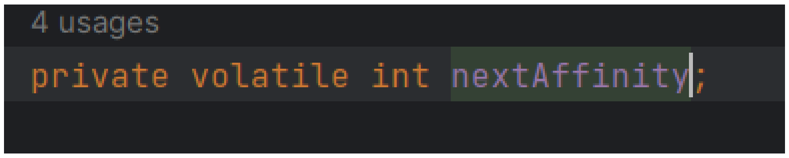
- Volatile key word :I added the volatile keyword to ensure that the value of the instance or variable is always up-to-date across all threads. This ensures that the system functions correctly and efficiently.

  1- on the server:

  ```
  private static volatile Server SERVER;
  ```

  2- on the RoundRobinLoadBalancer:

  ```
  4 usages
  private volatile int nextAffinity;
  ```

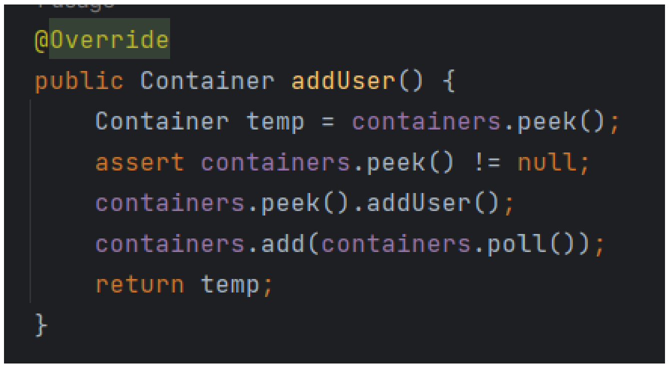
- One benefit of utilizing an enum with the singleton design pattern is that enums are inherently thread-safe, making it advisable to use enums in conjunction with singletons.

9.0 Data Consistency issues in the DB

During the testing process, I made an observation regarding the node redirecting requests to the affinity node. Specifically, when attempting to read data directly, there was a delay in performance. However, upon the second request, the data was accessed and performed correctly.

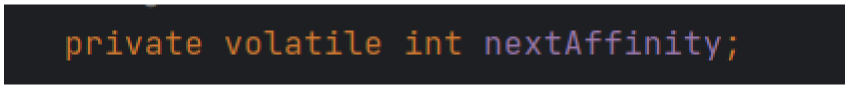
10.0 Node hashing and load balancing.

- I utilize the priority queue in the bootstrapping node, where I have overridden the "comparable" method within the Node class to compare nodes based on their current user count. This ensures that Java uses a min heap to prioritize the nodes. To keep the queue balanced, I remove the node each time a user is assigned to it, and then add it back in.



```
@Override
public Container addUser() {
    Container temp = containers.peek();
    assert containers.peek() != null;
    containers.peek().addUser();
    containers.add(containers.poll());
    return temp;
}
```

- Despite the fact that the load balancer may not always be 100% accurate, I still consider the round-robin technique to be a good approach for handling affinity node requests. This technique ensures that requests are distributed fairly among the nodes, which can improve the overall performance and efficiency of the system. when the node is initlize each node whill start the counter passed on his id .



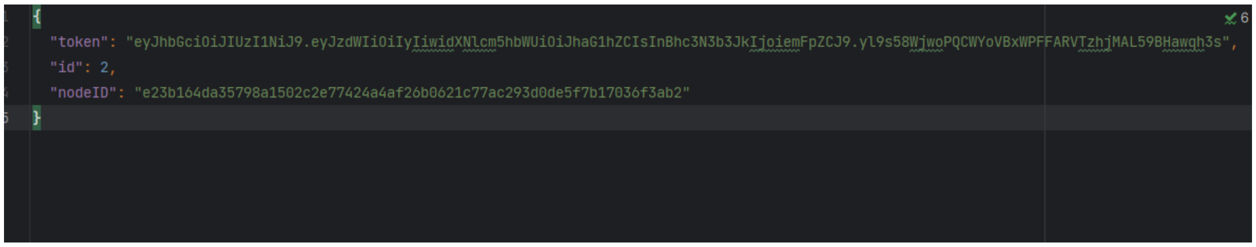
```java
private volatile int nextAffinity;
```

11.0 Security issues

In order to minimize security risks, I utilize JSON Web Tokens (JWTs) for user authentication. When a user attempts to register with the bootstrapping node, the node will provide the user with a token and information regarding their node.

When a user attempts to log in on a worker node the user will provide their token containing their user ID. The node will first check the validity of the token and then proceed to verify the user's authorization to access the node. This is done by comparing the current node ID with the user's file information, which includes their node information.

The user file information:



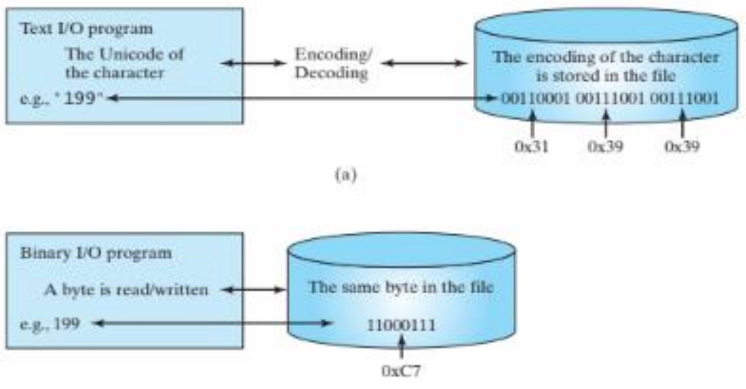
```json
{
  "token": "eyJhbGciOiJIUzI1NiJ9.eyJzdWIiOiIyIiwidXNlcm5hbWUiOiJhaG1hZCIsInBhc3N3b3JkIjoiemFpZCJ9.yl9s58WjwoPQCWYoVBxWPFFARVTzhjMAL59BHawqh3s",
  "id": 2,
  "nodeID": "e23b164da35798a1502c2e77424a4af26b0621c77ac293d0de5f7b17036f3ab2"
}
```

12.0 Process of Database Optimization:

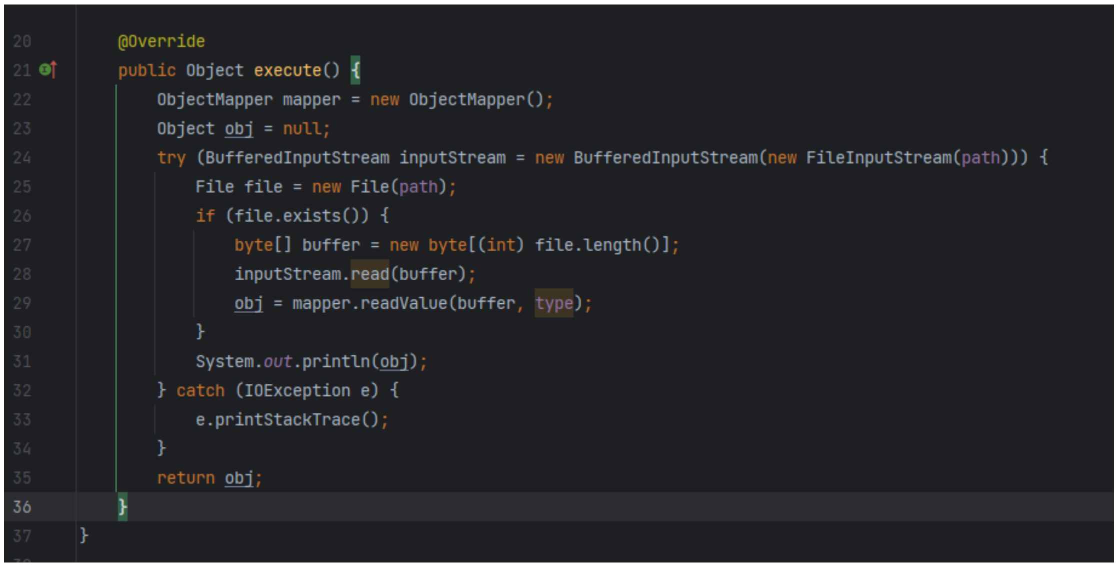
Database optimization is the process of improving the performance and efficiency of a database system by reducing the time it takes to access and manipulate data. This process involves analyzing the database schema, query patterns, hardware resources, and other factors that affect database performance.

The goal of database optimization is to improve the overall performance of a database system by reducing the response time for queries, minimizing the amount of disk space required for storage, and improving the reliability and availability of the system.

1- Using the Java Binary I/O : Binary I/O can be faster than text I/O because it involves less processing overhead. Since binary I/O reads and writes data in its raw binary form, it doesn't need to convert data into a human-readable format, like text I/O does.

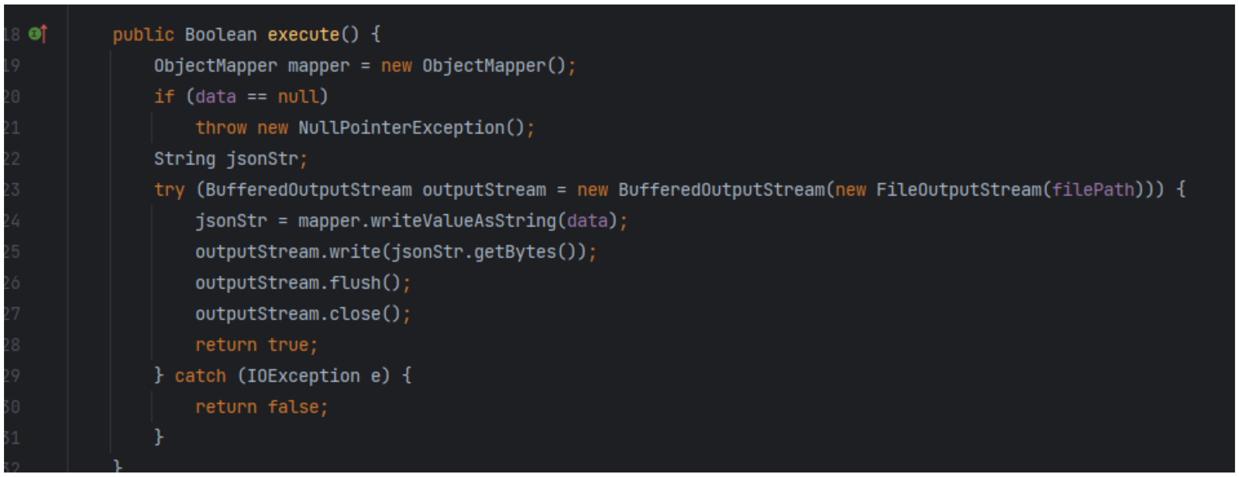


read operations in the "ReadFileCommand" class are performed using Binary I/O.



```java
    @Override
    public Object execute() {
        ObjectMapper mapper = new ObjectMapper();
        Object obj = null;
        try (BufferedInputStream inputStream = new BufferedInputStream(new FileInputStream(path))) {
            File file = new File(path);
            if (file.exists()) {
                byte[] buffer = new byte[(int) file.length()];
                inputStream.read(buffer);
                obj = mapper.readValue(buffer, type);
            }
            System.out.println(obj);
        } catch (IOException e) {
            e.printStackTrace();
        }
        return obj;
    }
}
```

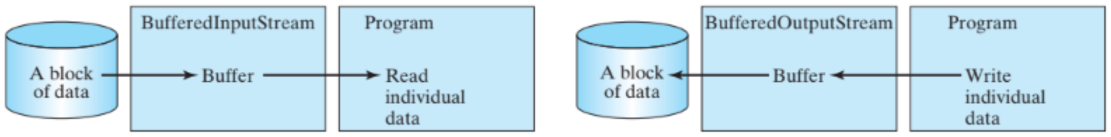
write operations in the " InsertDataCommand" class are performed using Binary I/O.



```java
    public Boolean execute() {
        ObjectMapper mapper = new ObjectMapper();
        if (data == null)
            throw new NullPointerException();
        String jsonStr;
        try (BufferedOutputStream outputStream = new BufferedOutputStream(new FileOutputStream(filePath))) {
            jsonStr = mapper.writeValueAsString(data);
            outputStream.write(jsonStr.getBytes());
            outputStream.flush();
            outputStream.close();
            return true;
        } catch (IOException e) {
            return false;
        }
    }
```

2- Using a buffer to read and write from files in Java: Reading and writing data from a file can be slow if done byte-by-byte. Using a buffer to read and write data in larger chunks can significantly improve performance.

Reduces the number of I/O operations required. Reading and writing data in larger chunks reduces the overhead associated with frequent I/O operations.

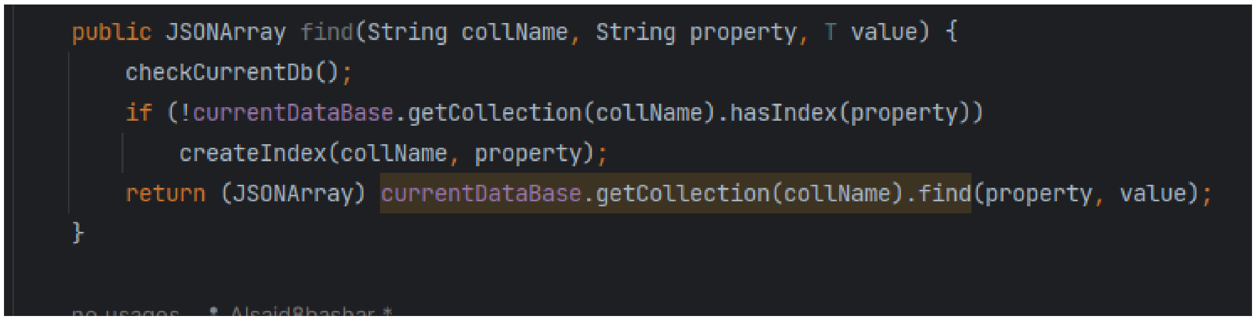


As the screenshots above I used the buffer to read and write from files .

3- The index creation process occurs only when the user invokes the "find" function and specifies the property to be read. Initially, the system checks if an index has already been created for that property. If an index does not exist, it creates one and then returns the query result to the user. As a result, the first execution may be slower, but subsequent reads will be faster due to the existence of the created index.

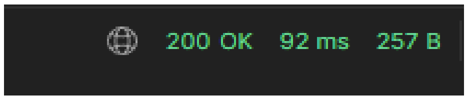
This approach reduces storage space by avoiding the creation of an index for every property in the collection, including those that may not be used at all.

In the façade :

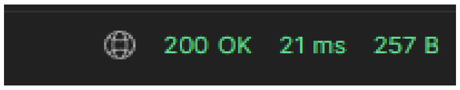


```
public JSONArray find(String collName, String property, T value) {
    checkCurrentDb();
    if (!currentDataBase.getCollection(collName).hasIndex(property))
        createIndex(collName, property);
    return (JSONArray) currentDataBase.getCollection(collName).find(property, value);
}
```

First time I do the read query VS Second time I do the same query and same data



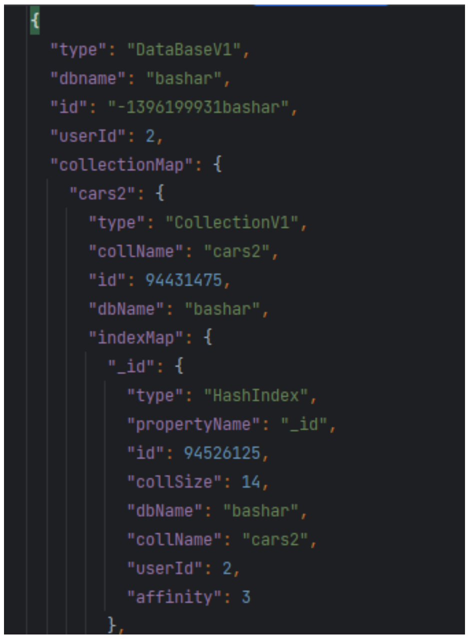
```
200 OK   92 ms   257 B
```



```
200 OK   21 ms   257 B
```

4- Putting Database Metadata in Memory Instead of the Entire Data:
I always separate the data from its metadata, enabling me to read directly from disk instead of memory. When the user calls the "useDb" function, I read the necessary database information, such as collections and schema, and store it in memory until the user logs out or changes the database. Whenever the user performs an operation, I retrieve the database information from memory to access the data based on its metadata.

This is how the database info should appear on the disk:



```
{
  "type": "DataBaseV1",
  "dbname": "bashar",
  "id": "-1396199931bashar",
  "userId": 2,
  "collectionMap": {
    "cars2": {
      "type": "CollectionV1",
      "collName": "cars2",
      "id": 94431475,
      "dbName": "bashar",
      "indexMap": {
        "_id": {
          "type": "HashIndex",
          "propertyName": "_id",
          "id": 94526125,
          "collSize": 14,
          "dbName": "bashar",
          "collName": "cars2",
          "userId": 2,
          "affinity": 3
        },
```

13.0 SOLID principles

13.1 Single Responsibility Principle (SRP):

The Single Responsibility Principle states that a class should have only one reason to change, meaning that it should have a single job to do. This is achieved by designing classes with a single responsibility:

- Command classes, where each command has a single responsibility.
- handler class, each handler is responsible for a single task or responsibility.
- the Broadcast class, which has the single responsibility of sending data to consumers.
- The UserAuthentication class has the single responsibility of verifying the user's credentials, as any changes or updates related to user authentication can be made within this class without impacting other parts of the system.
- The Index class has the singular responsibility of creating and updating the index.
- The SchemaValidator class has a single responsibility for defining the data against the schema.
- The CheckAffinity class has the sole responsibility of verifying whether the current node is the affinity node, and if not, it redirects the request to the affinity node.

13.2 Open-Closed Principle (OCP):

If we want to add a new database command to our system, we can achieve this by extending the Command interface and adding the new command without altering the existing code. By doing so, we can ensure that our system remains flexible and maintainable. Additionally, if we want to add new features to an existing command, we can utilize the CommandDecorator supertype. This approach provides us with the flexibility to add new functionality to a command without modifying the original command itself. By using this design pattern, we can avoid breaking the existing code. Furthermore, when using the chain of responsibility pattern in the server handler, we can add new handlers to the server without modifying other handler classes. This can be achieved by extending the RequestHandler supertype and implementing the required methods for the new handler. By doing so, we can ensure that the new handler is integrated seamlessly into the existing chain of responsibility, and the system remains open for extension without the need for significant modifications.

13.3 Liskov Substitution Principle (LSP):

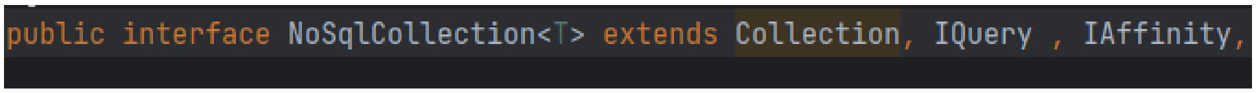
Objects are replaceable with their subtypes without affecting the correctness.

of the program, so if we have a parent class and a child class, both of them can.

be used interchangeably without getting incorrect results.

13.3 Interface Segregation Principle (ISP):

In order to adhere to the Interface Segregation Principle, I broke down the interfaces into smaller interfaces, each containing a method that can be added as needed. By doing this, I ensured that each interface has a clear and specific purpose, and clients only need to implement the methods they actually need,



```
public interface NoSqlCollection<T> extends Collection, IQuery , IAffinity,
```

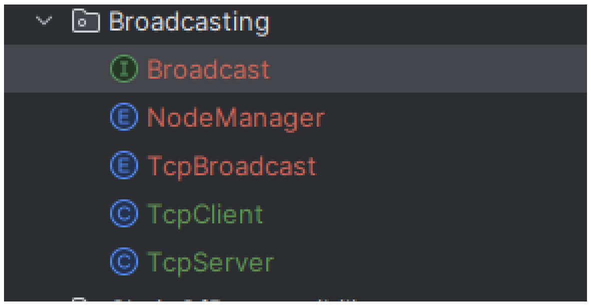
13.4 Dependency Inversion Principle (DIP):

Whenever I have classes that depend on each other, I implement a supertype for each of them to ensure that high-level modules do not depend on low-level modules, and both should depend on abstractions. By creating abstract classes or interfaces, we can define a common API that both the high-level and low-level modules can use to interact with each other. This promotes loose coupling between the modules and makes the system more modular and extensible.

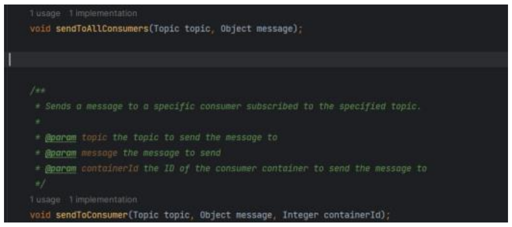
The Database component class has interfaces for each of its subcomponents, such as the Database interface, Collection interface, and Index interface. By utilizing interfaces, we can define a contract for each component, specifying the methods that each component must implement.

15 .0 Clean Code

1. Meaningful Names
    1. Classes names : All class names specific and descriptive.

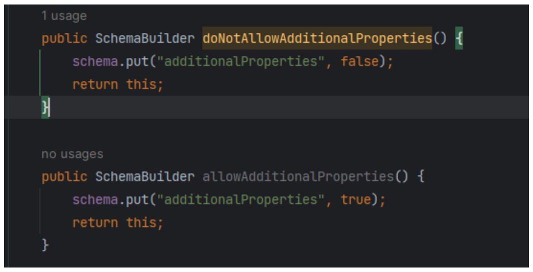


    2. Functions Names :fully understandable from the names (what it does)



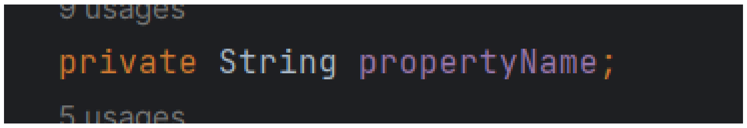
```
1 usage   1 implementation
void sendToAllConsumers(Topic topic, Object message);


/**
 * Sends a message to a specific consumer subscribed to the specified topic.
 *
 * @param topic the topic to send the message to
 * @param message the message to send
 * @param containerId the ID of the consumer container to send the message to
 */
1 usage   1 implementation
void sendToConsumer(Topic topic, Object message, Integer containerId);
```

        1. . When using the schema builder, I use a more human-readable language.



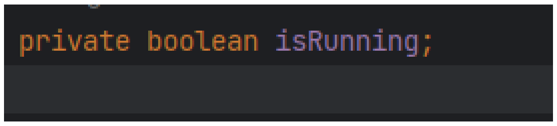
```
1 usage
public SchemaBuilder doNotAllowAdditionalProperties() {
    schema.put("additionalProperties", false);
    return this;
}

no usages
public SchemaBuilder allowAdditionalProperties() {
    schema.put("additionalProperties", true);
    return this;
}
```

    3. Variables names :always specific and following the camelCase



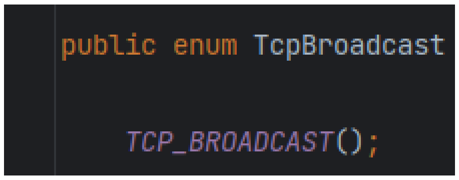
```
9 usages
private String propertyName;
5 usages
```

1. Boolean name:



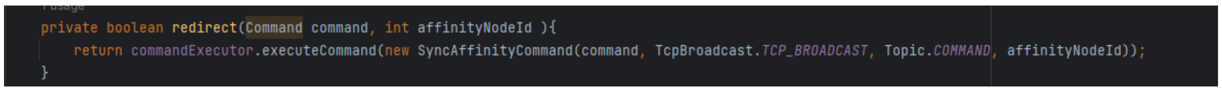
```
private boolean isRunning;
```

2. Constant name :



```
public enum TcpBroadcast

    TCP_BROADCAST();
```

2. Functions:
    1. Most functions in the implemented program are small and do one thing, one thing only, and they do it well. Flag arguments are avoided.

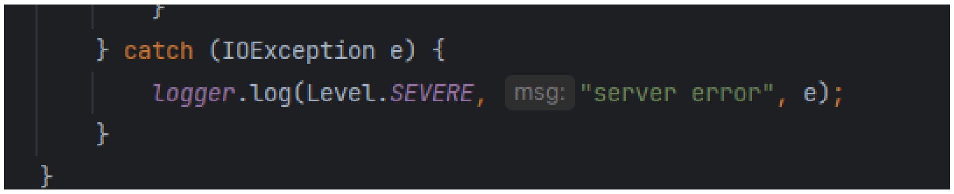


```
private boolean redirect(Command command, int affinityNodeId ){
    return commandExecutor.executeCommand(new SyncAffinityCommand(command, TcpBroadcast.TCP_BROADCAST, Topic.COMMAND, affinityNodeId));
}
```

    2. Function Arguments: I did my best to limit the amount of method arguments, because the less the function arguments the easier the testing process will be, most functions in the code have at most three arguments.
    3. Flag Arguments: Passing a Boolean into a function is considered a bad practice. it was avoided. the functions should do only one thing with flag arguments id will do more than one thing because the behavior of the function will change if the flag is true or
false.

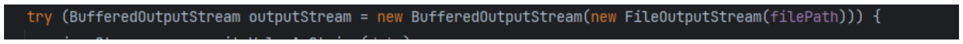
3. Try/Catch Blocks:

- It is recommended that the catch block always contains a log statement to record the exception, instead of printing the stack trace or leaving the block empty.



```java
    } catch (IOException e) {
        logger.log(Level.SEVERE, msg: "server error", e);
    }
}
```

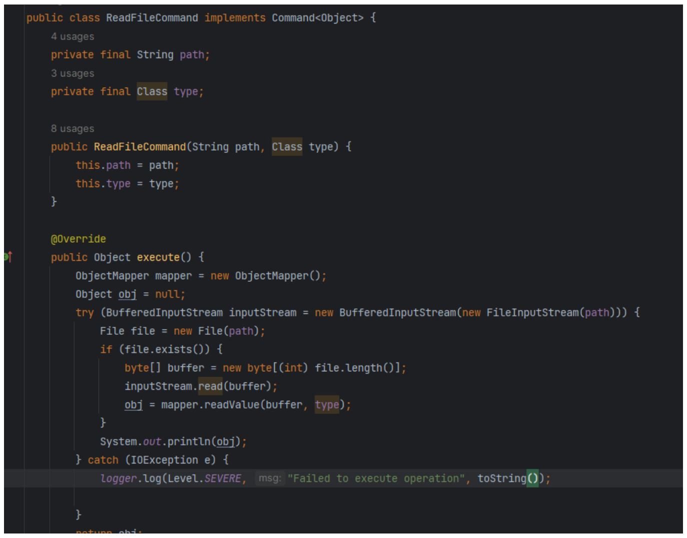
- I prefer to use the try-with-resources statement whenever possible.



```java
try (BufferedOutputStream outputStream = new BufferedOutputStream(new FileOutputStream(filePath))) {
```

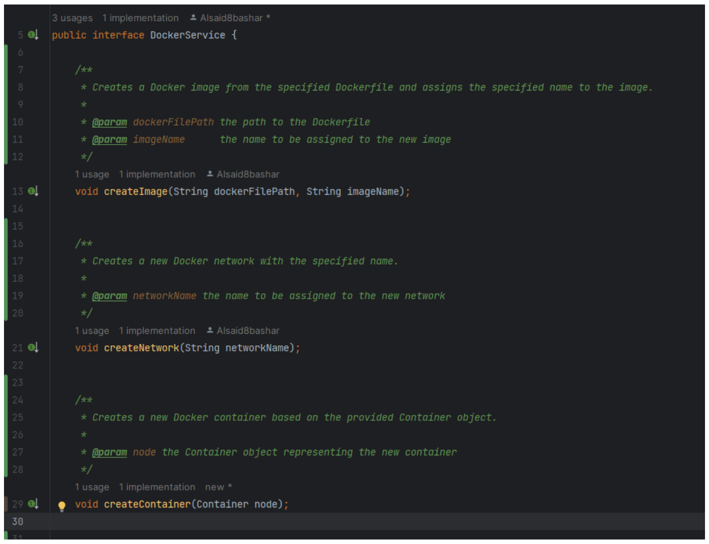
4 . DRY Principle:

To avoid duplicating code, I have created a separate command for reading from disk and returning the type specified by the provided class. This approach allows for code reuse across different parts of the application, as the same command can be used to read different types of objects from disk.



```java
public class ReadFileCommand implements Command<Object> {
    4 usages
    private final String path;
    3 usages
    private final Class type;

    8 usages
    public ReadFileCommand(String path, Class type) {
        this.path = path;
        this.type = type;
    }

    @Override
    public Object execute() {
        ObjectMapper mapper = new ObjectMapper();
        Object obj = null;
        try (BufferedInputStream inputStream = new BufferedInputStream(new FileInputStream(path))) {
            File file = new File(path);
            if (file.exists()) {
                byte[] buffer = new byte[(int) file.length()];
                inputStream.read(buffer);
                obj = mapper.readValue(buffer, type);
            }
            System.out.println(obj);
        } catch (IOException e) {
            logger.log(Level.SEVERE, msg: "Failed to execute operation", toString());

        }
```

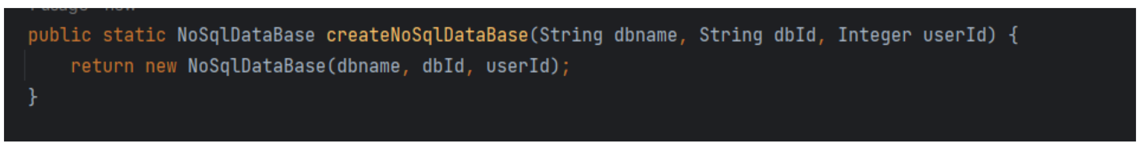
4. Comments: I have used comments to document the public API methods. By providing clear and concise descriptions of the methods, developers who use the API can better understand how to use the methods and what they do.



```java
3 usages   1 implementation   Alsaid8bashar *
public interface DockerService {

    /**
     * Creates a Docker image from the specified Dockerfile and assigns the specified name to the image.
     *
     * @param dockerFilePath the path to the Dockerfile
     * @param imageName      the name to be assigned to the new image
     */
    1 usage   1 implementation   Alsaid8bashar
    void createImage(String dockerFilePath, String imageName);


    /**
     * Creates a new Docker network with the specified name.
     *
     * @param networkName the name to be assigned to the new network
     */
    1 usage   1 implementation   Alsaid8bashar
    void createNetwork(String networkName);


    /**
     * Creates a new Docker container based on the provided Container object.
     *
     * @param node the Container object representing the new container
     */
    1 usage   1 implementation   new *
    void createContainer(Container node);

```

16.0 Effective Java

- Item 1: Consider static factory methods instead of constructors: Static factory is used to return an instance of the NoSqlDataBase

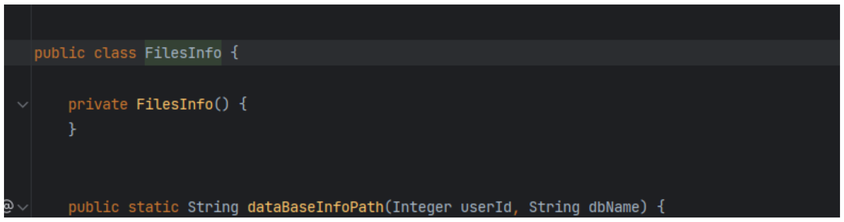


```
public static NoSqlDataBase createNoSqlDataBase(String dbname, String dbId, Integer userId) {
    return new NoSqlDataBase(dbname, dbId, userId);
}
```

- Item 2: Consider a builder when faced with many constructor parameters:
  I used the builder pattern with a schema that defines the fields and methods needed to create objects with a specific structure.

- Item 3: Enforce the singleton property with a private constructor or an Enum type:
  in my code, I used different approaches to implement the singleton pattern. For the TcpServer class, I used the traditional approach, where I created a private constructor and a public static method to access the singleton instance.
  For the NodeManager and LoadBalancer classes, however, I used the enum way of implementing the singleton pattern .This approach is thread-safe by default

- Item 4: Enforce noninstantiability with a private constructor:
  I used a class that is just a grouping of static methods and static fields such as the “FilesInfo “, “Folderinfo” ,”UserAuthentication “classes .



```
public class FilesInfo {

    private FilesInfo() {
    }


    public static String dataBaseInfoPath(Integer userId, String dbName) {
```

- Item 5: Prefer dependency injection to hardwiring resources: I always prefer using dependency injection instead of hardwiring resources in my code , As an example, when working on the SyncAllNodesCommand, I always utilize dependency injection to pass the necessary broadcast to the command via a constructor parameter. This approach allows me to modify the broadcast functionality separately without having to modify the code directly in the SyncAllNodesCommand class.

- Item 6: Avoid creating unnecessary objects: No new objects are created when we can reuse an existing one. Common use of one method to increase code reusability.

- Item 8: Avoid finalizers. Since using finalizers may cause severe performance issues and security problems, using them was avoided.

- Item 9: Prefer try-with-resources to try-finally: It's the best way to close resources and its more readable, since we are dealing with files, all methods that deal with resources use try-with-resources, to ensure the resources are closed.

- Item 12: Always override toString: All classes override the toString method.

- Item 13: Minimize the accessibility of classes and members:

  Information Hiding and Encapsulation is very important, since classes with

  public mutable fields are not thread safe I Made sure that each class or member

  is as inaccessible as possible.

- Item 14. In public classes, use accessor methods, not public fields: Encapsulation of data using getters/setters. Class members cannot be accessed directly, a getter should be used.

- Item 15. Minimize Mutability :Most Variables are final and private to provide data protection, most entity classes are immutable so they are thread-safe and require no synchronization, unless there is a good reason

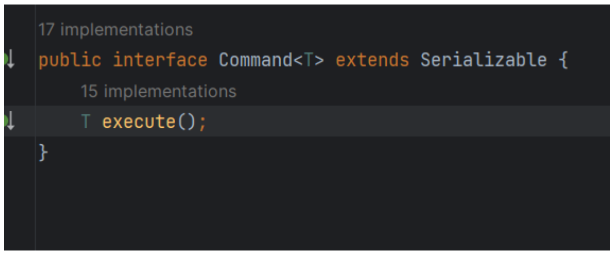
- Item   16. Prefer interfaces to abstract classes : I avoid the overuse of abstract classes and prefer to use interfaces whenever possible

- Item 29: Use interfaces only to define types: Using constant interface pattern is a poor use of interfaces, having interfaces to hold constants is wrong and should be avoided, no final values were declared in the interface.

- Item 23: Don't use raw types in new code : With raw types we will get compile-time exception, so using them was avoided.

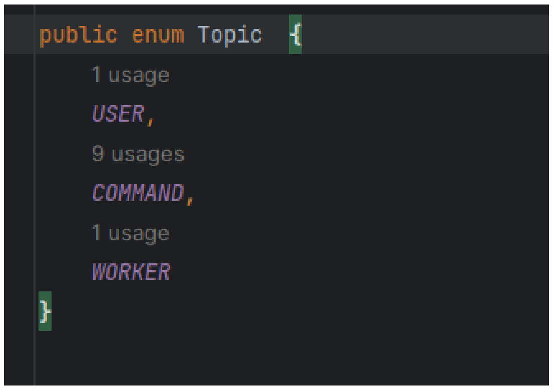
- Item 24. Eliminate unchecked warnings: Unchecked warnings are too important, I Eliminated every unchecked warning possible
- Item 25. Prefer lists to arrays: I always choose to use lists over arrays.

- Item 27: Favor generic methods: I utilized generic functions for the read query.



```
17 implementations
public interface Command<T> extends Serializable {
    15 implementations
    T execute();
}
```

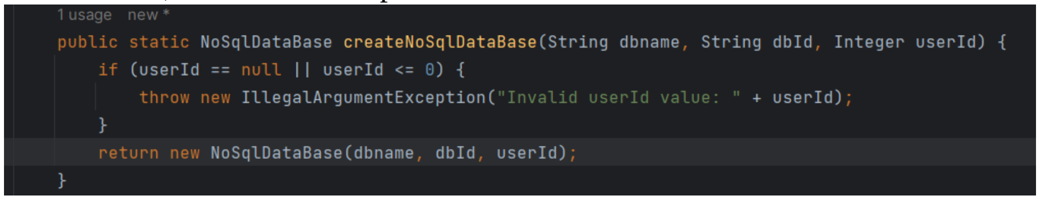
- Item 30: Use Enums instead of int constants: As topics are fixed



```
public enum Topic {
    1 usage
    USER,
    9 usages
    COMMAND,
    1 usage
    WORKER
}
```

- Item 36: Consistently use the Override annotation: the @Override annotation was used on
method declarations that override declarations from interfaces as well as
classes.

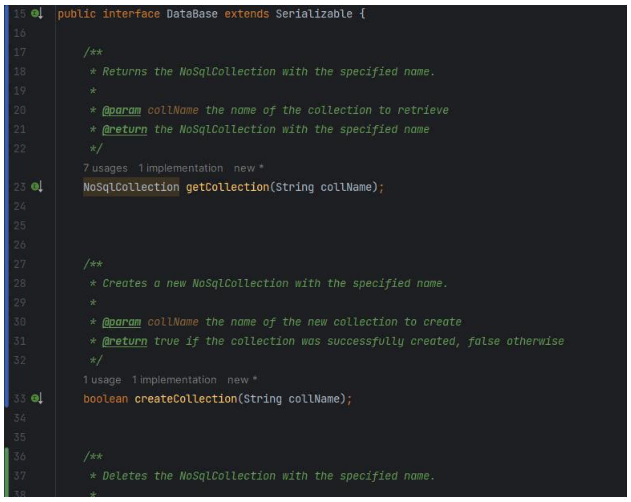
- Item 38. Check parameters for validity : Since we are using static methods instead of constructors, I can check the parameter values.



```
1 usage   new *
public static NoSqlDataBase createNoSqlDataBase(String dbname, String dbId, Integer userId) {
    if (userId == null || userId <= 0) {
        throw new IllegalArgumentException("Invalid userId value: " + userId);
    }
    return new NoSqlDataBase(dbname, dbId, userId);
}
```

- Item 43. Return empty arrays or collections, not nulls: i If the user tries to read data that does not exist, I will return an empty JSON Array.

- Item 44. Write doc comments for all exposed API elements: For every interface, I write comments to explain the purpose of the interface functions.


7 usages 1 implementation new *
1 usage 1 implementation new *

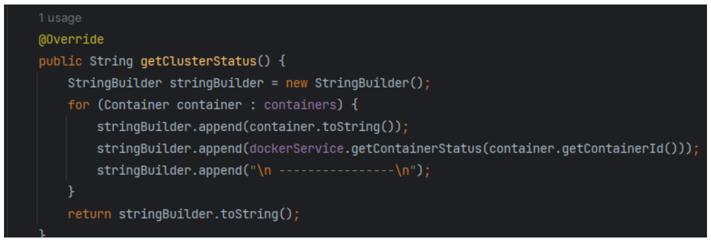
```java
public interface DataBase extends Serializable {

    /**
     * Returns the NoSqlCollection with the specified name.
     *
     * @param collName the name of the collection to retrieve
     * @return the NoSqlCollection with the specified name
     */
    NoSqlCollection getCollection(String collName);



    /**
     * Creates a new NoSqlCollection with the specified name.
     *
     * @param collName the name of the new collection to create
     * @return true if the collection was successfully created, false otherwise
     */
    boolean createCollection(String collName);


    /**
     * Deletes the NoSqlCollection with the specified name.
     *
```

- Item 46 :Prefer for-each loops to traditional for loops. I always prefer to use foreach instead of traditional loops whenever possible.

- Item 47. Know and use libraries: I have used several libraries, including:
  - Docker Java API
  - JWTs for handling tokens
  - Jackson Java library.

- Item 49. Prefer primitive types to boxed primitives: I always prefer primitive types over boxed primitives.

- Item 50. Avoid Strings where other types are more appropriate: To avoid creating new objects when appending strings, I opted to use StringBuilder instead of String when returning the cluster status. This allows me to efficiently manipulate strings without incurring the overhead of creating new objects.



```java
1 usage
@Override
public String getClusterStatus() {
    StringBuilder stringBuilder = new StringBuilder();
    for (Container container : containers) {
        stringBuilder.append(container.toString());
        stringBuilder.append(dockerService.getContainerStatus(container.getContainerId()));
        stringBuilder.append("\n ---------------\n");
    }
    return stringBuilder.toString();
}
```

- Item 52. Refer to objects by their interface: When I use a class that implements an interface, I refer to the concrete object by its interface.

- Item 60. Favor the use of standard exceptions : I always prefer using existing exceptions rather than custom ones.

- Item 65. Don't ignore exceptions: I never ignore exceptions and always use a logger to handle them.

- Item 66. Synchronize access to shared mutable data: I always synchronize with shared data, such as the singleton instance and shared variables within the same class.

- Item 67. Avoid excessive synchronization I always try my best to synchronize in the right places and share data between threads.

- Item 68. Prefer executors and tasks to threads :In the TcpServer class, to handle multiple users, I use newCachedThreadPool. Additionally, in the TcpBroadcast class, I use fixedThreadPool.

17.0 DevOps practice

1- Maven: I have utilized Maven as a build tool, which is an incredibly powerful tool that allows me to easily add library dependencies from the Maven repository, such as "Docker-java-API" and numerous other libraries that I have incorporated into my system. Additionally, I separated my Maven project from my Spring project, and use Maven to both install and package the JAR file for use in the Spring application.

2- Spring : To construct a language-independent database that anyone can use, I utilized Spring-Boot as the transaction layer. To achieve this, I developed my REST API project and incorporated the Maven project as a library, then used Spring-Boot's Restful controller to handle user requests. I selected Spring-Boot because I am more familiar with it than other Restful frameworks such as java Webservice.

3- Git and GitHub: I have utilized Git and GitHub to upload and save my project's progress.

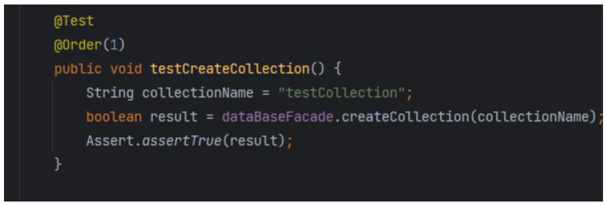
4- Docker: To fulfill the requirement of representing each node as a virtual machine, I utilized Docker and its network features to establish communication among the nodes. By leveraging Docker's containerization technology, I was able to create separate virtual environments for each node, and Docker's networking capabilities allowed these nodes to communicate with each other seamlessly.

5- Junit: For testing my code, I utilize the JUnit framework to ensure that database operations function correctly and that the system operates as intended. This helps me verify the functionality of the code and identify any issues that need to be addressed.

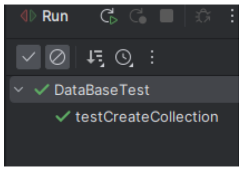
18.0 Code Testing:

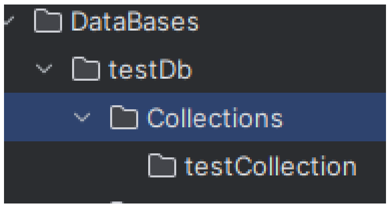
Initially, I will share some test cases that I developed using the JUnit framework. These test cases primarily focus on the database operations such as creating a collection, creating a schema, inserting valid data that conforms to the schema, inserting invalid data, searching for an object in the collection (which I expect to be present), and finally searching for an object that does not exist.

1- Create Collection test case.

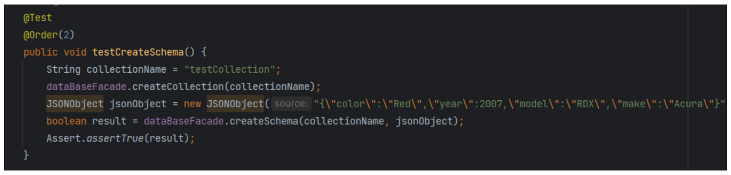


```java
@Test
@Order(1)
public void testCreateCollection() {
    String collectionName = "testCollection";
    boolean result = dataBaseFacade.createCollection(collectionName);
    Assert.assertTrue(result);
}
```

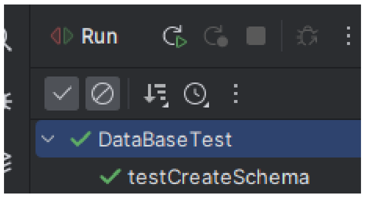


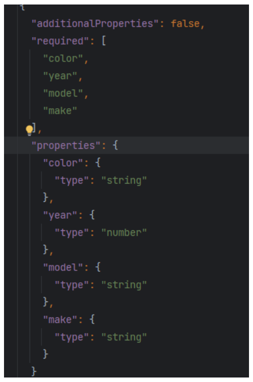


2- Create schema test case:



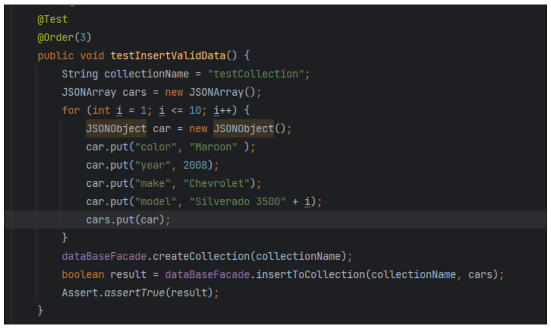
```java
@Test
@Order(2)
public void testCreateSchema() {
    String collectionName = "testCollection";
    dataBaseFacade.createCollection(collectionName);
    JSONObject jsonObject = new JSONObject( source: "{\"color\":\"Red\",\"year\":2007,\"model\":\"RDX\",\"make\":\"Acura\"}"
    boolean result = dataBaseFacade.createSchema(collectionName, jsonObject);
    Assert.assertTrue(result);
}
```



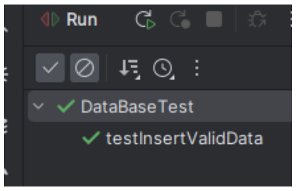


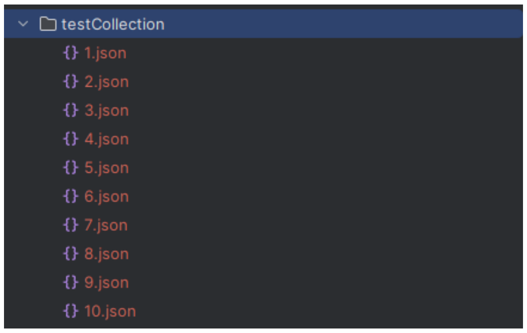
```json
{
  "additionalProperties": false,
  "required": [
    "color",
    "year",
    "model",
    "make"
  ],
  "properties": {
    "color": {
      "type": "string"
    },
    "year": {
      "type": "number"
    },
    "model": {
      "type": "string"
    },
    "make": {
      "type": "string"
    }
  }
```

3- Insert valid data to the collection.

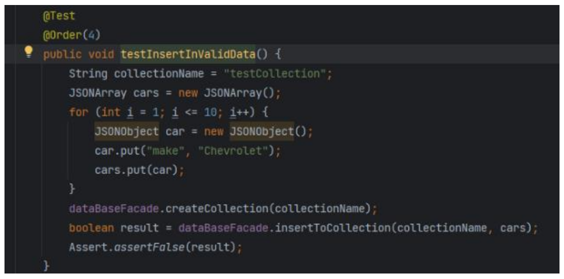


```java
@Test
@Order(3)
public void testInsertValidData() {
    String collectionName = "testCollection";
    JSONArray cars = new JSONArray();
    for (int i = 1; i <= 10; i++) {
        JSONObject car = new JSONObject();
        car.put("color", "Maroon" );
        car.put("year", 2008);
        car.put("make", "Chevrolet");
        car.put("model", "Silverado 3500" + i);
        cars.put(car);
    }
    dataBaseFacade.createCollection(collectionName);
    boolean result = dataBaseFacade.insertToCollection(collectionName, cars);
    Assert.assertTrue(result);
}
```

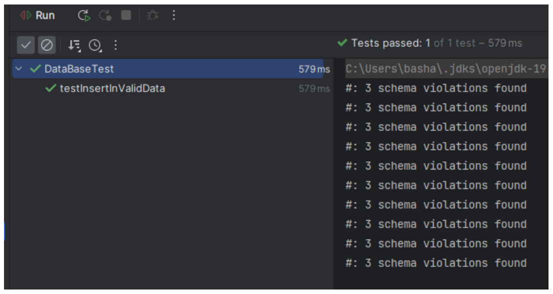




4- insert Invalid data to the collection

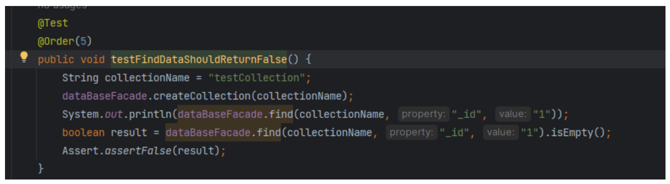


```
@Test
@Order(4)
public void testInsertInValidData() {
    String collectionName = "testCollection";
    JSONArray cars = new JSONArray();
    for (int i = 1; i <= 10; i++) {
        JSONObject car = new JSONObject();
        car.put("make", "Chevrolet");
        cars.put(car);
    }
    dataBaseFacade.createCollection(collectionName);
    boolean result = dataBaseFacade.insertToCollection(collectionName, cars);
    Assert.assertFalse(result);
}
```

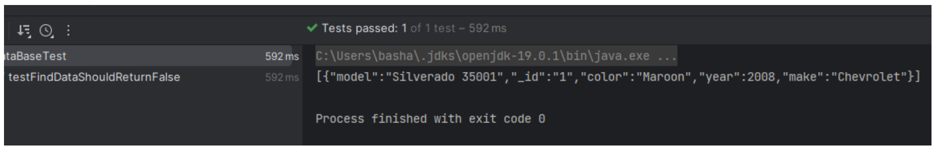

Run
Tests passed: 1 of 1 test – 579 ms
DataBaseTest 579 ms
testInsertInValidData 579 ms

```
C:\Users\basha\.jdks\openjdk-19
#: 3 schema violations found
#: 3 schema violations found
#: 3 schema violations found
#: 3 schema violations found
#: 3 schema violations found
#: 3 schema violations found
#: 3 schema violations found
#: 3 schema violations found
#: 3 schema violations found
#: 3 schema violations found
```

5- Find data (which I expect to be present):

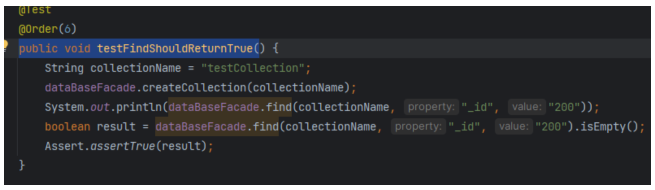


```
@Test
@Order(5)
public void testFindDataShouldReturnFalse() {
    String collectionName = "testCollection";
    dataBaseFacade.createCollection(collectionName);
    System.out.println(dataBaseFacade.find(collectionName, property: "_id", value: "1"));
    boolean result = dataBaseFacade.find(collectionName, property: "_id", value: "1").isEmpty();
    Assert.assertFalse(result);
}
```

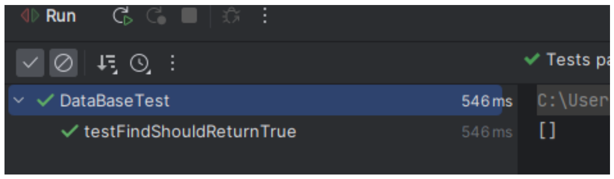

Tests passed: 1 of 1 test – 592 ms
taBaseTest 592 ms
testFindDataShouldReturnFalse 592 ms

```
C:\Users\basha\.jdks\openjdk-19.0.1\bin\java.exe ...
[{"model":"Silverado 35001","_id":"1","color":"Maroon","year":2008,"make":"Chevrolet"}]

Process finished with exit code 0
```

6- Find data does not exist:



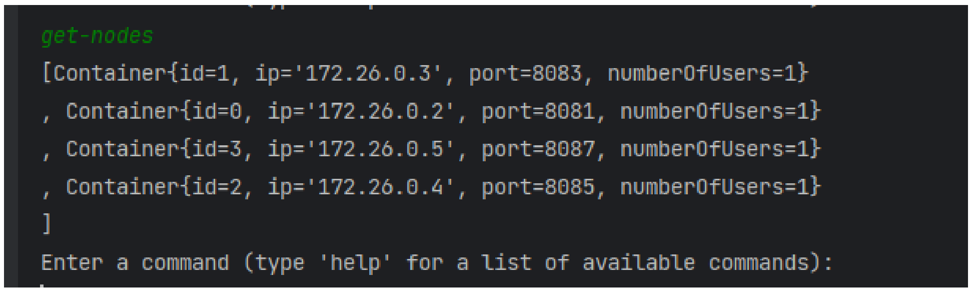
```
@Test
@Order(6)
public void testFindShouldReturnTrue() {
    String collectionName = "testCollection";
    dataBaseFacade.createCollection(collectionName);
    System.out.println(dataBaseFacade.find(collectionName, property: "_id", value: "200"));
    boolean result = dataBaseFacade.find(collectionName, property: "_id", value: "200").isEmpty();
    Assert.assertTrue(result);
}
```



```
C:\User
[]
```

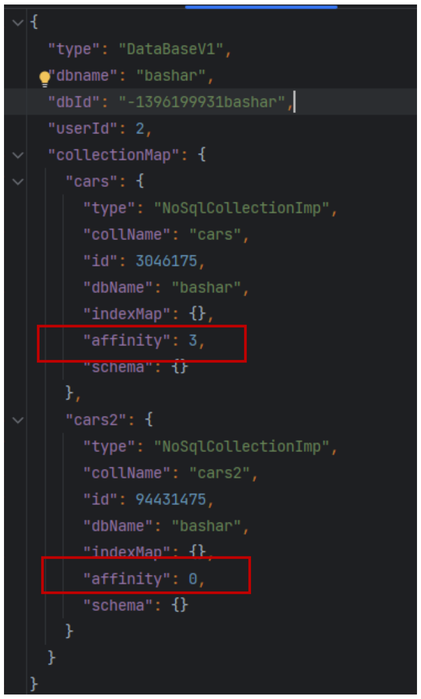
Load balance evidences:

1- To balance the load across nodes, I deployed four nodes and assigned one user to each container. This resulted in four users in total. The screenshot showcases the worker nodes and the number of users assigned to each individual node.



```
get-nodes
[Container{id=1, ip='172.26.0.3', port=8083, numberOfUsers=1}
, Container{id=0, ip='172.26.0.2', port=8081, numberOfUsers=1}
, Container{id=3, ip='172.26.0.5', port=8087, numberOfUsers=1}
, Container{id=2, ip='172.26.0.4', port=8085, numberOfUsers=1}
]
Enter a command (type 'help' for a list of available commands):
```

2- Collection to affinity load balance : I will create two collections on a single node and then access their files to display the collection affinity.

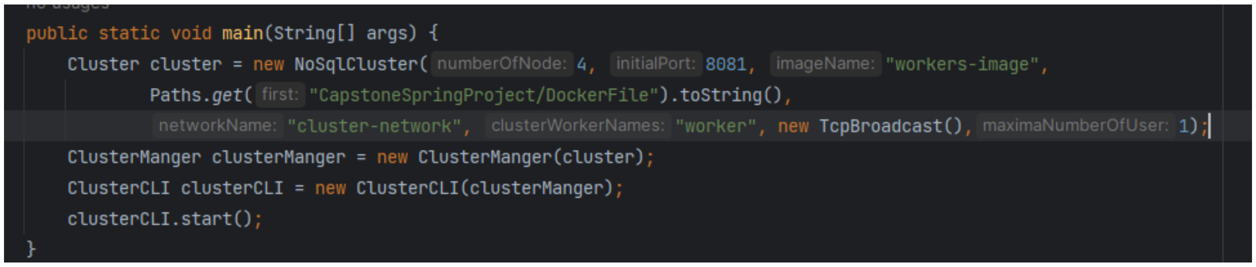


```
{
  "type": "DataBaseV1",
  "dbname": "bashar",
  "dbId": "-1396199931bashar",
  "userId": 2,
  "collectionMap": {
    "cars": {
      "type": "NoSqlCollectionImp",
      "collName": "cars",
      "id": 3046175,
      "dbName": "bashar",
      "indexMap": {},
      "affinity": 3,
      "schema": {}
    },
    "cars2": {
      "type": "NoSqlCollectionImp",
      "collName": "cars2",
      "id": 94431475,
      "dbName": "bashar",
      "indexMap": {},
      "affinity": 0,
      "schema": {}
    }
  }
}
```

19.0 Bootstrapping Node

The bootstrapper is designed to run on a host machine or virtual machine, rather than inside a container. To implement Docker functionality, I am utilizing the Docker-Java API, which provides exceptional features for my project. I have created a class to represent the Docker service on the bootstrapping node, which includes functionality to create Docker images, containers, start containers, and more.

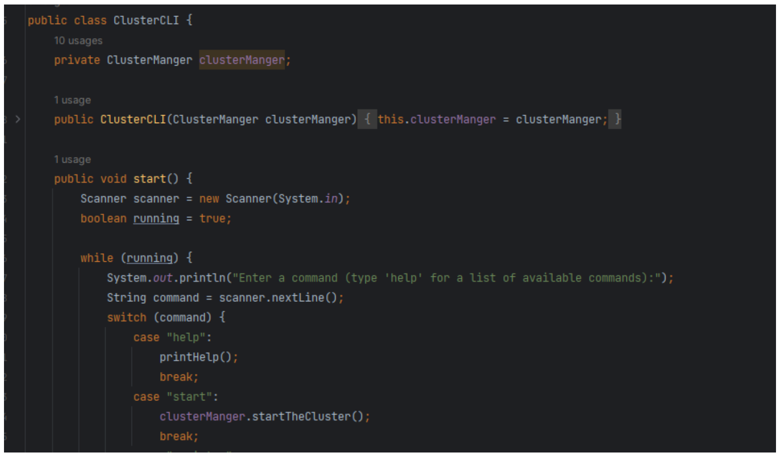
To commence the creation of the Docker infrastructure and worker nodes, the startCluster function is called after the administrator has provided essential information like network name and number of nodes in the cluster. To communicate with the Docker desktop, the dockerService class is employed, which contains all the necessary logic required for the process.



```java
public static void main(String[] args) {
    Cluster cluster = new NoSqlCluster( numberOfNode: 4,  initialPort: 8081,  imageName: "workers-image",
            Paths.get( first: "CapstoneSpringProject/DockerFile").toString(),
             networkName: "cluster-network",  clusterWorkerNames: "worker", new TcpBroadcast(), maximaNumberOfUser: 1);
    ClusterManger clusterManger = new ClusterManger(cluster);
    ClusterCLI clusterCLI = new ClusterCLI(clusterManger);
    clusterCLI.start();
}
```

The bootstrapping node includes a broadcast mechanism, to send the users information and the cluster node information.

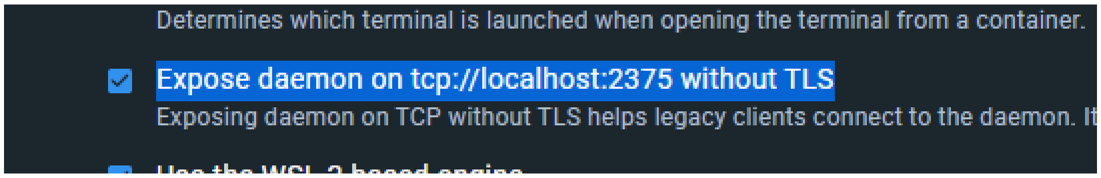
Managing and controlling the operation of a cluster requires a crucial component known as a cluster manager. It plays an essential role in ensuring the cluster functions effectively. The administrator can utilize a command-line interface (CLI) that offers a range of features such as scaling up, retrieving and displaying the cluster's status, deleting the cluster, and registering new users to control the cluster effectively.



```java
public class ClusterCLI {
    10 usages
    private ClusterManger clusterManger;

    1 usage
    public ClusterCLI(ClusterManger clusterManger) { this.clusterManger = clusterManger; }

    1 usage
    public void start() {
        Scanner scanner = new Scanner(System.in);
        boolean running = true;

        while (running) {
            System.out.println("Enter a command (type 'help' for a list of available commands):");
            String command = scanner.nextLine();
            switch (command) {
                case "help":
                    printHelp();
                    break;
                case "start":
                    clusterManger.startTheCluster();
                    break;
```

I implemented a simple auto-scaling mechanism by requesting the administrator to input the maximum number of users per node. If the head node in the queue reaches the maximum number of users, the cluster scales up, and bootstrapping distributes the new container to other nodes in the cluster.

Important note: If you intend to run the bootstrapping node on your machine, it is crucial to ensure that the "Expose daemon on tcp://localhost:2375 without TLS" option is enabled in the Docker desktop.



Determines which terminal is launched when opening the terminal from a container.

✓ Expose daemon on tcp://localhost:2375 without TLS

Exposing daemon on TCP without TLS helps legacy clients connect to the daemon. It

✓ Use the WSL 2 based engine