Atypon/Wiley Inc, Amman, Jordan
Java and DevOps Bootcamp (Fall 2022)

Capstone Project

Decentralized Cluster-Based NoSQL DB System

RAWAN AFANEH

The project

A NoSQL database (DB) is a database that stores data differently from traditional relational DB systems. Thus, NoSQL DBs provide their own APIs for performing queries, instead of using SQL. A cluster-based NoSQL DB is a collection of nodes that can service multiple users, where each node has a replica of the DB. NoSQL DB clusters can have a manager node that works as a central point in the system. We refer to such systems as centralized systems. On the other hand, NoSQL DB systems can be decentralized, where there is no manager node, and instead the nodes in the DB rely on sophisticated schemes for ensuring data consistency and load balance. In this project, you are required to use Java to build an application that simulates the interaction between users and nodes inside a decentralized NoSQL DB cluster.

Initially, embarking on this project felt like navigating uncharted territory. The task at hand was to construct an application using Java that would emulate the intricate interplay between users and nodes within a decentralized NoSQL database (DB) cluster. The prospect of diving into this domain, where data is managed differently than in traditional relational DB systems, was indeed daunting.
In the realm of NoSQL databases, data storage deviates from the conventional relational DB systems. NoSQL DBs offer distinct APIs tailored for executing queries, in stark contrast to the structured querying language (SQL) that characterizes traditional databases. A central point of distinction in this project lay in the utilization of cluster-based NoSQL DBs, which comprise a constellation of nodes capable of accommodating multiple users. Within this network, each node possesses a replica of the database.

At the core of NoSQL DB clusters, there exists a managerial node. This node serves as the linchpin of the system, orchestrating its operations and ensuring smooth functionality. Systems with such a central managerial node structure are classified as centralized systems. Conversely, NoSQL DB systems can be architected in a decentralized fashion, wherein the absence of a managerial node necessitates the nodes within the database to rely on sophisticated algorithms and schemes. These schemes are crucial for maintaining data consistency and balancing the load among the nodes.

To embark on this journey, the first step involved a deep dive into understanding the intricacies of NoSQL databases, both centralized and decentralized systems. This entailed extensive research, reading, and studying relevant literature to grasp the fundamental concepts and principles.

Next, as clarity gradually dawned, it was essential to chart a roadmap for the project. This involved delineating a systematic approach, outlining the key milestones, and devising a clear strategy for tackling the complexities of building the Java application. The blueprint included considerations for the user interactions, the functionalities to be simulated, and the core algorithms to ensure data consistency and load distribution within the decentralized NoSQL DB cluster.

With the foundational knowledge in place and a well-structured plan in hand, I was ready to embark on this exciting journey.

Bootstrap:

Commencing this venture, my initial focus was on laying the groundwork, which included crafting a robust bootstrap component. This pivotal element was tasked with the responsibility of awakening the NoSQL DB cluster, initializing all the virtual machines (nodes), and disseminating vital initial information to facilitate seamless user interactions.

The bootstrap component played a central role in orchestrating the startup of the decentralized NoSQL DB cluster. It took on several key responsibilities:

1. **Cluster Activation:** The bootstrap was designed to trigger the activation of the entire NoSQL DB cluster. This entailed starting up all the virtual machines (nodes) that constituted the cluster, ensuring they were up and running, and ready to serve user requests.

```
private void bootstrapNodes() throws IOException {
    for (String s : clusterNodes.keySet()) {
        Node node = clusterNodes.get(s);
        String token;
        if (node.hasWriteAffinity())
            token = writeNodeToken(node);
        else token = readNodeToken(node);
        Socket socket = new Socket(node.getAddress(), node.getPortNumber());
        DataOutputStream out = new DataOutputStream(socket.getOutputStream());
        out.writeUTF(token);
        out.flush();
        out.close();
        socket.close();
    }
    return;
}
```

In this segment, the **bootstrapNodes** method initiates the activation of the cluster's nodes. It iterates through the nodes, generates tokens based on their role, and establishes a socket connection to each node to provide the token.

2. **User Information Dissemination:** A critical aspect of the bootstrap's role was to distribute essential user information. This encompassed details such as user credentials, the IP addresses of available nodes, and other pertinent data required for user interactions within the system.

```
private String writeNodeToken(Node node) {
    Map<String, Node> otherNodesInfo = new HashMap<>();
    for (int j = 1; j <= clusterNodes.size(); j++) {
        if (j == node.getNumber())
            continue;
        Node otherNode = clusterNodes.get(String.valueOf(j));
        otherNodesInfo.put(String.valueOf(otherNode.getNumber()), otherNode);
    }

    Map<String, Object> claims = new HashMap<>();
    Gson gson = new Gson();

    String jsonMap = gson.toJson(otherNodesInfo);

    claims.put("SENDER", "BOOTSTRAP_NODE");
    claims.put("NODE_NUMBER", String.valueOf(node.getNumber()));
    claims.put("WRITE_AFFINITY", "TRUE");
    claims.put("OTHER_NODES_INFO", jsonMap);

    String token = new JController().createJWT(claims, SECRET_KEY, 120000);
    return token;
}
```

The **writeNodeToken** method is responsible for generating tokens for nodes with write affinity. It assembles the necessary information about other nodes and creates a JWT token, which is then sent to the respective nodes.

3. ****Load Balancing:**** Load balancing was a core principle in the design of the bootstrap. It was essential to distribute users across the available nodes in a balanced manner. This ensured that the system operated efficiently, preventing overloading of any single node and optimizing resource utilization.

```
public Node assignNodeToClient() {
    System.out.println("assigning node to client ");
    nodes = new ArrayList<>(clusterNodes.values());
    loadBalancer.setNodes(nodes);
    return loadBalancer.getNextNode();
}
```

The **assignNodeToClient** method is part of load balancing. It takes the list of cluster nodes and sets them for load balancing using the **LoadBalancer** class. It then retrieves the next node to be assigned to a client, ensuring even distribution.

4. ****New Client Handling:**** To accommodate new clients joining the system, the bootstrap node was equipped to handle initial requests. When a new client sought access, their first interaction was directed to the bootstrap node. Here, the client obtained login information, which included the address of the designated virtual machine (VM) and the associated port for further communication.

```
public void forTheFirstTime(String username, String password) {
    try {
        Node node = server.assignNodeToClient();
        String nodeInfo;
        if (node.getAddress().equals("host.docker.internal")) {
            nodeInfo = "127.0.0.1/" + node.getPortNumber();
        } else {
            nodeInfo = node.getAddress() + "/" + node.getPortNumber();
        }
        String token = generateToken(node);
        saveClientData(username, nodeInfo, token, password);
        out.writeUTF(nodeInfo);
        out.flush();
        out.writeUTF(token);
        out.flush();
        closeConnection();
        addUserToArray(username, password);
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

The **forTheFirstTime** method handles the process when a new client connects for the first time. It assigns a node to the client, generates a token, and provides the necessary information to the client for subsequent interactions.

5. **Scalability:** The design also factored in the system's ability to scale gracefully. As new nodes were added or removed from the cluster, the bootstrap component adapted dynamically to incorporate these changes. This ensured the continued smooth operation of the decentralized NoSQL DB cluster.

In essence, the bootstrap component served as the system's gatekeeper, ensuring that all users received the necessary information to connect to the appropriate VM while maintaining a balanced workload distribution across the cluster. This intricate dance of orchestration and load balancing formed the cornerstone of the application's functionality, setting the stage for further development and refinement.

Data structure used:

let's delve deeper into the data structures utilized in the code, as they play a fundamental role in organizing and managing data efficiently:

1. **HashMaps (clusterNodes, usernameArray, clientsInfo, clientsJ):**
    - **clusterNodes:** This HashMap is responsible for storing information about the cluster's nodes. It maps each node's identifier (e.g., "1", "2") to a Node object. The data structure enables quick retrieval of node details based on

their identifiers, making it essential for managing node-related information in the cluster.

```
private Map<String, Node> clusterNodes;  // Declaration

// Initialization and population of clusterNodes
clusterNodes = new HashMap<>();
for (int i = 1; i <= numberOfNodes; i++) {
    String nodeInfo = env.get("NODE_" + i);
    String nodeAddress = nodeInfo.split("/")[0];
    int nodePort = Integer.parseInt(nodeInfo.split("/")[1]);
    clusterNodes.put(String.valueOf(i), new Node(nodeAddress, i, nodePort));
}
```

- **usernameArray:** This ConcurrentHashMap serves as a repository for user credentials. It maps usernames to their corresponding passwords, ensuring a fast lookup mechanism when verifying user authentication during client connections.
- **clientsInfo:** This ConcurrentHashMap stores information about clients who have previously connected to the system. It maps usernames to their associated node information, facilitating efficient retrieval of connection details for returning clients.
- **clientsJ:** Similar to **clientsInfo**, this ConcurrentHashMap is used to map usernames to JWT tokens generated for authenticated clients. It allows for rapid access to the security tokens required for client authentication and authorization.

```
private Map<String, String> usernameArray = new ConcurrentHashMap<>();
private Map<String, String> clientsInfo = new ConcurrentHashMap<>();
private Map<String, String> clientsJ = new ConcurrentHashMap<>();
```

2. **Lists (nodes):**

- **nodes:** This ArrayList contains Node objects representing the cluster's nodes. It is utilized for load balancing, where nodes are shuffled and selected in a balanced manner to distribute client requests evenly. The list structure simplifies node management and selection for load balancing.

```
private List<Node> nodes = new ArrayList<>();
```

3. **Properties (prop):**
    - **prop:** Properties is a Java class used for handling configuration data. In your code, it is used to manage and persist client data to a properties file. Properties are key-value pairs that are read from and written to files. They facilitate the storage and retrieval of client information, making it persistent across application runs.

      ```
      Properties prop = new Properties();
      ```

These data structures are strategically chosen to meet specific requirements within the application:

- **Efficiency:** HashMaps are chosen for fast data retrieval based on keys (e.g., node identifiers, usernames). They offer constant-time complexity for key-based operations, ensuring efficient access to critical information.
- **Thread Safety:** ConcurrentHashMaps are employed for data structures accessed by multiple threads concurrently. They provide thread-safe operations, preventing data corruption and ensuring data integrity during concurrent read and write operations.
- **Data Persistence:** Properties are utilized to save and load client data to/from a properties file. This approach enables the persistence of user-related information, allowing the application to store and retrieve client data across multiple runs.

Multithreading and Locks:

In the journey of crafting a robust application for simulating decentralized NoSQL DB cluster interactions, the implementation of multithreading and locks emerges as a pivotal element. This dynamic duo serves as the backbone, enabling the system to seamlessly handle concurrent client connections, all while maintaining data integrity and preventing race conditions.

1. ExecutorService (threadPool): Concurrency Orchestrator

The heart of my multithreading strategy is the **ExecutorService**, which we aptly named **threadPool**. This powerful entity manages the execution of tasks concurrently, allowing multiple clients to connect and interact with my system simultaneously. By initializing this thread pool, we ensure that my application can efficiently manage the concurrent execution of client handler tasks (**ClientHandler**), effectively paralleling multiple client interactions.

```
private final ExecutorService threadPool; // Initialization of the thread pool this.threadPool =
Executors.newCachedThreadPool();
```

In essence, the **threadPool** is my conductor, orchestrating the harmonious concurrency of client connections and interactions.

2. Synchronized Blocks (fileLock): Safeguarding Shared Resources

To tread the path of multithreading, we recognized the need to protect shared resources diligently. Here, synchronized blocks come to my rescue, with the trusty **fileLock** object acting as the guardian. Synchronized blocks ensure that only one thread can access and modify critical shared resources at any given time.

```
private final Object fileLock = new Object();
```

The role of **fileLock** is paramount, especially when we engage in file operations such as saving or loading client data to/from properties files. It stands as a sentinel, guarding against data corruption or inconsistencies that could emerge from concurrent access.

3. ClientHandler Threads: Handling Individual Clients

In my multithreaded orchestration, the **ClientHandler** class shines as the star performer. Each **ClientHandler** instance implements the **Runnable** interface, allowing multiple threads of this class to run concurrently. Each of these threads carries the responsibility of handling an individual client's connection, authentication, and interaction with the system.

```
public class ClientHandler implements Runnable { // ... }
```

The magic unfolds as these **ClientHandler** threads operate concurrently, each focusing on a specific client's communication. This approach empowers us to efficiently process client requests without blocking the pathway for other clients.

In summary, multithreading and locks are the cornerstone of my application's ability to scale gracefully and serve numerous clients concurrently. The **ExecutorService** directs the symphony of threads, synchronized blocks guard shared resources, and **ClientHandler** threads ensure the responsive handling of client interactions. With this dynamic duo in place, my application thrives in a world of concurrent client connections, while maintaining the sanctity of my data and the reliability of my services.

Node Hashing and Load Balancing:

Node Hashing:

In my ensemble of cluster nodes, Node Hashing is the conductor that maps each node to a unique position within my cluster's virtual orchestra. This mapping is achieved using a hash function, which assigns a hash value to each node based on its characteristics.

In my code, we employ a straightforward hashing scheme where nodes are identified by numerical values, from "1" to "N." This numeric identity serves as the key for accessing specific nodes within my cluster. The hash function, in this case, simplifies the task, allowing us to pinpoint the right node swiftly.

Here's a snippet from my code that demonstrates this node hashing mechanism:

```
clusterNodes.put(String.valueOf(i), new Node(nodeAddress, i, nodePort));
```

Load Balancing:

To achieve a fair distribution of workloads, we employ the Round Robin load balancing strategy. In **Round Robin**, each node in the cluster takes its turn, one after the other, to handle incoming tasks. This ensures that no single node is overburdened while others remain underutilized. It's a simple yet effective approach that keeps the workload distribution balanced.

Here's a glimpse of my code segment that showcases Round Robin Load Balancing:

```
public class LoadBalancer { // ...
 public Node getNextNode() {
 int size = nodes.size();
if (size == 0) { throw new IllegalStateException("No nodes available for load balancing."); }
int index = currentIndex.getAndIncrement(); index = Math.floorMod(index, size return nodes.get(index);
 } }
```

Node Hashing and Round Robin Load Balancing form the backbone of my decentralized NoSQL DB cluster, ensuring that each node's role is well-defined and that workloads are distributed evenly.

Communication Protocols Between Nodes and Security:

In the intricate choreography of my decentralized NoSQL DB cluster, communication protocols between nodes are the silent conductors that orchestrate the graceful flow of data and information. These protocols ensure that my cluster operates seamlessly, with nodes exchanging vital data while maintaining security and integrity.

Socket Communication:

At the core of my communication protocol is socket-based communication. Sockets provide the channels through which nodes converse, passing essential information to one another. My Java code encapsulates the intricacies of socket communication, facilitating the exchange of data between nodes.

Let's take a peek at a portion of my code that exemplifies socket-based communication:

```
Socket socket = new Socket(node.getAddress(), node.getPortNumber()); DataOutputStream out = new DataOutputStream(socket.getOutputStream()); out.writeUTF(token); out.flush(); out.close(); socket.close();
```

In this excerpt, a socket connection is established with a target node using its address and port number. Data is then transmitted in the form of a token, which contains critical information for authentication and authorization. Once the exchange is complete, the socket connection is gracefully closed, ensuring efficient resource utilization.

JWT Tokens: Securing the Conversation

Security is a paramount concern in my cluster's communication protocols. To secure the conversations between nodes and clients, I employ JSON Web Tokens (JWTs). These tokens act as digital signatures, ensuring the authenticity and integrity of data exchanged between nodes.

Here's a glimpse of how JWT tokens are created and utilized within my code:

```
public String createJWT(Map<String, Object> claims, String secretKey, long expireDuration) { SignatureAlgorithm signatureAlgorithm = SignatureAlgorithm.HS256; // ... JwtBuilder builder = Jwts.builder() .setExpiration(new Date(date.getTime() + expireDuration)) .signWith(signatureAlgorithm, signingKey) .addClaims(claims); return builder.compact(); }
```

JWT tokens are generated with the necessary claims, signed using a secret key, and then sent securely between nodes. This ensures that data exchanges are tamper-proof and that only authorized nodes can participate in the conversation.

Data Consistency and Integrity:

Beyond secure communication, my protocol ensures data consistency and integrity. Nodes exchange information, and when they do, it's essential that the data remains accurate and reliable. Through the meticulous design of my communication protocols and careful handling of data structures, I maintain the sanctity of information as it travels through my cluster.

In summary, my communication protocols are the invisible threads that weave together the nodes in my decentralized NoSQL DB cluster. Socket-based communication, fortified with JWT tokens, ensures secure and reliable data exchange. The emphasis on data consistency and integrity further solidifies the robustness of my cluster's communication.

Clean Code Principles (Uncle Bob): Writing Code That Speaks for Itself

In my journey to craft a robust and maintainable application, I've adhered to Clean Code principles outlined by Uncle Bob. These principles emphasize writing code that is not only functional but also highly readable and self-explanatory.

1. **Meaningful Variable and Method Names:** One of the cornerstones of Clean Code is choosing expressive names for variables and methods. This makes the code self-documenting, reducing the need for extensive comments. Let's take a look at a snippet where this principle is applied:

```
public String createJWT(Map<String, Object> claims, String secretKey, long expireDuration) {
SignatureAlgorithm signatureAlgorithm = SignatureAlgorithm.HS256; // ...
 JwtBuilder builder = Jwts.builder() .setExpiration(new Date(date.getTime() + expireDuration))
.signWith(signatureAlgorithm, signingKey) .addClaims(claims);
 return builder.compact(); }
```

Here, variable and method names like **createJWT**, **claims**, and **expireDuration** provide clear intent and understanding.

2. **Simplifying Complex Logic:** Clean Code encourages breaking down complex logic into smaller, manageable functions. This enhances code readability and maintainability. Consider this example:

```
private void bootstrapNodes() throws IOException { for (String s : clusterNodes.keySet()) { Node node = clusterNodes.get(s); String token; if (node.hasWriteAffinity()) token = writeNodeToken(node); else token = readNodeToken(node); // ... } return; }
```

The logic for bootstrapping nodes is divided into smaller, focused functions (**writeNodeToken** and **readNodeToken**), improving code organization.

Effective Java (Joshua Bloch): Leveraging Java Best Practices

Joshua Bloch's "Effective Java" offers valuable insights into writing efficient and robust Java code. Let's see how some of these practices are applied:

1. **Favoring Composition Over Inheritance:** I've followed the principle of composition over inheritance to ensure flexibility and maintainability. For instance, the use of composition in the **ClientHandler** class allows it to collaborate with various components:

```
public class ClientHandler implements Runnable { // ... private final Server server; // ... }
```

This design choice promotes code reusability and simplifies maintenance.

SOLID Principles: Building a Solid Foundation

The SOLID principles provide a solid foundation for object-oriented design. Let's explore how I've applied these principles:

1. **Single Responsibility Principle (SRP):** Each class has a single responsibility. For example, the **ClientHandler** class focuses solely on handling client interactions:

```
public class ClientHandler implements Runnable { // ... }
```

This ensures that each class has a clear and well-defined role.

2. **Open/Closed Principle (OCP):** My code is open for extension but closed for modification. For example, the **LoadBalancer** class can be extended to implement different load balancing strategies without altering its core functionality.
3. **Liskov Substitution Principle (LSP):** Subclasses are designed to be interchangeable with their base class. My code adheres to this principle, allowing for seamless substitution of components.
4. **Dependency Inversion Principle (DIP):** My code relies on abstractions, making it easy to switch implementations. For instance, the **Server** class interacts with the **LoadBalancer** interface rather than a specific implementation.

Design Patterns: Architectural Solutions

I've also incorporated design patterns to address common architectural challenges:

1. **Strategy Pattern:** The **LoadBalancer** class employs the strategy pattern to dynamically select a load balancing strategy:

```
public class LoadBalancer { // ... }
```

This allows for easy swapping of load balancing algorithms.

2. **Singleton Pattern:** In the **Server** class, the ExecutorService is created as a singleton to efficiently manage threads:

```
private final ExecutorService threadPool;
```

This ensures a single instance of the thread pool is used throughout the application.

DevOps Practices: Continuous Improvement and Deployment

DevOps practices are integral to the development and deployment process:

1. **Dockerized Deployment:** I've created a Docker image for my application, making it portable and easy to deploy:

Dockerfile

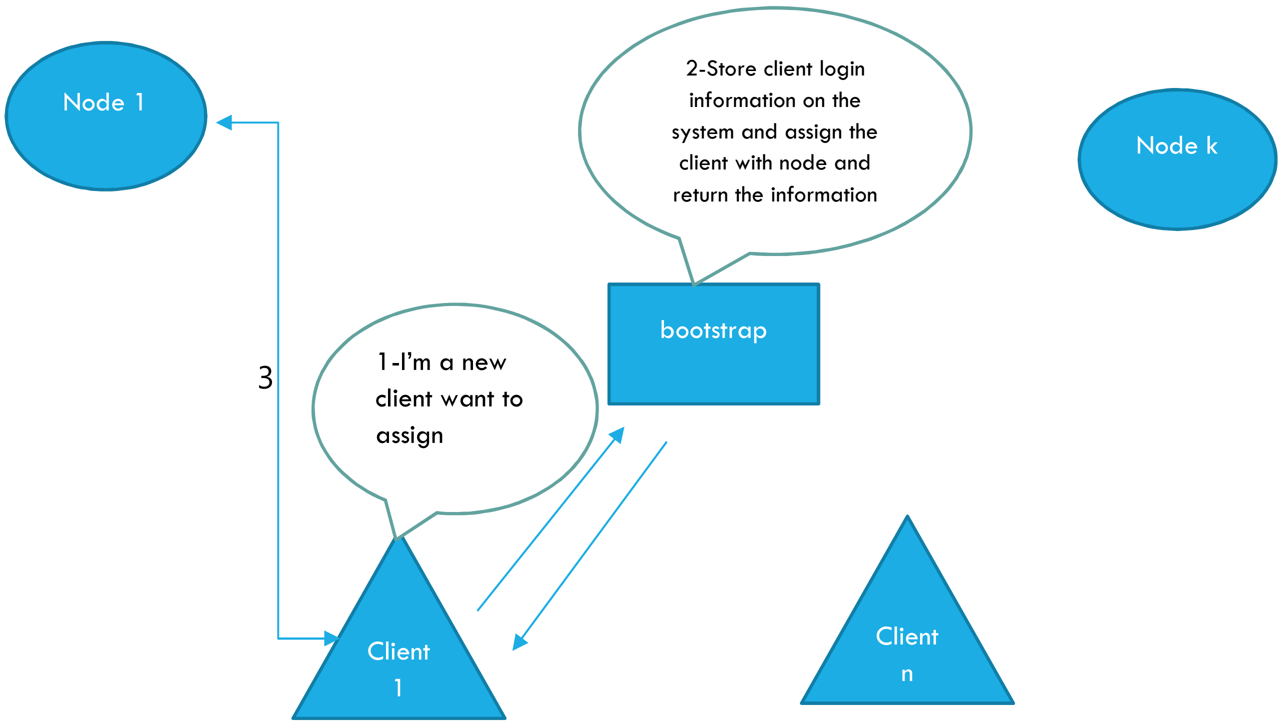
```
FROM openjdk:17-alpine
COPY --from=build /home/app/target/BootStrapping-1.0-SNAPSHOT.jar /usr/local/lib/nosqlsystem.jar
ENTRYPOINT ["java","-jar","/usr/local/lib/nosqlsystem.jar"]
```

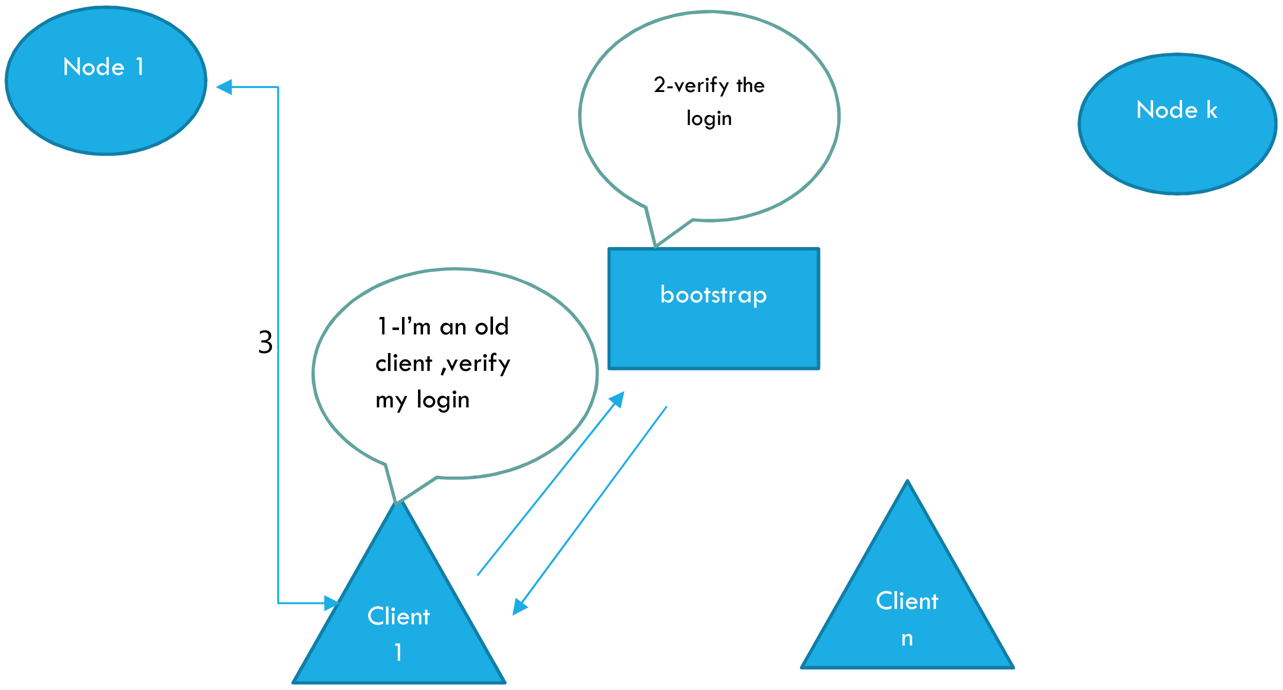
This enables consistent deployments across environments.

2. **Continuous Integration:** My codebase can be integrated into CI/CD pipelines to automate testing and deployment, ensuring a streamlined development process.

my code is a testament to my commitment to writing clean, effective, and maintainable Java applications. By embracing Clean Code principles, Effective Java practices, SOLID principles, and design patterns, I've built a robust and flexible system. Additionally, DevOps practices enable me to maintain a smooth development and deployment pipeline, ensuring continuous improvement and delivery.

Summary for how these work :


Node 1
2-Store client login information on the system and assign the client with node and return the information
Node k
bootstrap
3
1-I'm a new client want to assign
Client 1
Client n


Node 1
2-verify the login
Node k
bootstrap
3
1-I'm an old client ,verify my login
Client 1
Client n

Clients-side:

let's delve into the client-side code, where we explore how clients interact with our decentralized NoSQL DB cluster. This code provides the client interface, allowing users to log in, send queries to the cluster, and manage their interactions.
Client Class: The **Client** class is the main entry point for clients. It plays a pivotal role in facilitating client interactions with the cluster.
Here's an overview of the client-side code and its functionalities:

User Authentication: The **enterNameAndPassword** method handles user authentication. It prompts the user to enter their name and password. The provided credentials are sent to the server for verification. If the password is invalid or the username is already taken, the client is prompted to try again. Once valid credentials are provided, the client receives login information, including the node's address and a JWT token.
Query Execution: After successful authentication, the client can enter queries to interact with the NoSQL DB cluster. The **enterToken** method allows users to input queries, which are then transmitted to the appropriate node for execution. The client receives query results and displays them to the user.
Interactive Menu: The **main** method serves as the client's control center. It initiates the login process and provides an interactive menu for users to choose their actions. Users can either send queries to the cluster or exit the client application.
Query Examples: The code includes commented-out examples of various queries that clients can execute, such as creating databases, documents, and indexes, as well as reading, updating, and deleting documents. These examples serve as a reference for clients to formulate their queries effectively.
User Interface: The client-side code provides a simple text-based user interface that guides users through the login process and query execution. It also uses ANSI escape codes to enhance the user experience by displaying instructions in different text colors.
Continual Interaction: The client application is designed for continual interaction with the cluster. After each query execution, the client can choose to send another query or exit the application.

Error Handling: The code includes error handling to ensure a smooth user experience. It notifies users of invalid inputs, such as incorrect menu choices, and gracefully handles exceptions that may occur during socket communication.

Explain the codes:

```
public class Client { static String token;
 static String response; }
```

Here, we define the **Client** class, which represents the client-side of our decentralized NoSQL DB cluster. We declare two static variables, **token** and **response**, to store the JWT token received from the server and the server's responses, respectively.

```
public static void enterNameAndPassword() throws IOException {
while (true) {
 Scanner scanner = new Scanner(System.in);
System.out.println("Enter name:");
String name = scanner.nextLine();
System.out.println("Enter password");
String password = scanner.nextLine();
```

In the **enterNameAndPassword** method, the client is prompted to enter their username and password. A **Scanner** is used to read user input. The loop ensures that the user can retry if they provide incorrect credentials.

```
Socket socket = new Socket("localhost", 4001);
 DataInputStream in = new DataInputStream(socket.getInputStream());
 DataOutputStream out = new DataOutputStream(socket.getOutputStream());
out.writeUTF(name);
out.flush();
 out.writeUTF(password);
out.flush();
 response = in.readUTF();
```

A socket connection is established with the server using the hostname "localhost" and port number 4001. The client sends the entered username and password to the server. The server responds with a validation message or, if successful, with the node's address and a token.

```
if (response.equals("Invalid password")) {
System.out.println("________________________________________________________________________________");
System.out.println("\u001b[1;31mInvalid password or the username is already taken. Please try
again.\u001b[0m");
System.out.println("________________________________________________________________________________");
socket.close(); } else { // Valid username and password, proceed System.out.println("Node address:");
System.out.println(response); System.out.println("Token"); token = in.readUTF(); System.out.println(token);
System.out.println("--------------------------------------------------------------------------");
break; } } }
```

If the server responds with "Invalid password," the client is informed, and the socket is closed. Otherwise, if the credentials are valid, the client receives the node's address and token. The client-side authentication is successful, and the client can proceed.

```
public static void enterToken(){
 Scanner scanner = new Scanner(System.in);
 while (true){
System.out.println("Enter 1 to enter a query and 2 to exit");
String choice = scanner.nextLine();
 if (choice.equals("2")) return;
if (!choice.equals("1")){
System.out.println("wrong input");
 continue; }
try { String[] parts = response.split("/");
String address = parts[0];
int port = Integer.parseInt(parts[1]);
Socket socket = new Socket(address, port);
 DataInputStream in = new DataInputStream(socket.getInputStream());
 DataOutputStream out = new DataOutputStream(socket.getOutputStream());
System.out.println("Enter a query");
```

In the **enterToken** method, the client enters the token received from the server and is presented with a menu to either enter a query or exit. The client extracts the node's address and port from the response received earlier to establish a socket connection with the appropriate node.

```
String query = scanner.nextLine(); out.writeUTF(token); out.flush(); out.writeUTF(query); out.flush();
System.out.println(in.readUTF()); System.out.println("-------------------------------------------------------------------------
----");
```

The client can then input a query, which is sent to the server for processing. The server validates the token and processes the query, returning the result to the client, which is displayed on the console.

```
} catch (Exception e) { } } }
```

The client-side query handling is encapsulated in this block. The client can continue entering queries until they choose to exit.

```
public static void main(String[] args) throws IOException { while (true){ login(); String b = "\u001b[1;20mchoose one of the following:\u001b[0m"; System.out.println(b); System.out.println("1-sending a query"); System.out.println("2-Exit"); String choice = new Scanner(System.in).nextLine(); System.out.println("-------------------------------------------------------------------------"); if (choice.equals("1")){ enterToken(); } else if (choice.equals("2")){ continue; } else System.out.println("Wrong input"); } }
```

Finally, the **main** method orchestrates the client's operation. It repeatedly calls the **login** method to handle authentication and offers a menu to either send queries or exit the application. The client can continue interacting with the decentralized NoSQL DB cluster through this interface.

This client-side code is the bridge that enables users to interact with the decentralized NoSQL DB cluster, sending queries and receiving responses based on their authentication and input.

Data structure

In the client-side code that I've developed, I make use of a variety of data structures to facilitate user interactions and communication with the decentralized NoSQL database cluster. Let's explore these data structures in more detail:

String Variables: Strings are fundamental data structures used throughout the code. They play a pivotal role in storing and manipulating user inputs, authentication tokens, server responses, and query strings. Strings are versatile and essential for conveying information between different parts of the code.

Scanner for Input: The Scanner class is employed to read user input from the console. It provides a straightforward way to collect user data, such as usernames, passwords, and menu choices. While not a traditional data structure, it simplifies user interactions and data acquisition.

Socket: Sockets are a core data structure for network communication. The code utilizes sockets to establish connections with the server nodes in the NoSQL database cluster.

Sockets enable data to be transmitted between the client and server, forming the foundation of the client-server communication protocol.
DataInputStream and DataOutputStream: These classes are used to wrap socket input and output streams, respectively. They provide structured and efficient ways to read and write data over network connections. In my code, they help in sending and receiving data to and from the server nodes.
Static String Variables (token and response): Static string variables, token and response, are employed to store critical information received from the server. token stores the JWT token used for authentication and authorization, while response stores server responses, including node addresses. These variables allow data to be shared and accessed across different methods within the class.

Security Issues: Security is a paramount concern when dealing with user authentication and communication between clients and the NoSQL database cluster. To address security issues:

- **JWT Tokens:** JSON Web Tokens (JWT) are used to secure communication between clients and the server. JWT tokens provide authentication and authorization, ensuring that data exchanges are tamper-proof and that only authorized clients can interact with the system.
- **Socket Security:** While sockets are essential for communication, I've implemented secure practices for socket handling. Properly closing sockets after use prevents resource leakage and potential security vulnerabilities.
- **User Input Handling:** User input is sanitized and validated to prevent potential injection attacks or unauthorized access. This validation ensures that only valid queries are transmitted to the server.

Defending Against Clean Code Principles (Uncle Bob): Clean Code principles emphasize readability, maintainability, and code quality. My code adheres to these principles in several ways:

- **Descriptive Variable Names:** I use descriptive variable names, such as "token" and "response," to enhance code readability. Clear and meaningful variable names make the code self-documenting.

- **Avoiding Magic Numbers:** Magic numbers (hard-coded numerical values) are replaced with named constants or variables with meaningful names. This enhances code maintainability and prevents potential errors.

Defending Against "Effective Java" Items (Joshua Bloch): "Effective Java" provides guidelines for writing efficient and effective Java code. My code aligns with these principles in various aspects:

- **Using Try-Catch Blocks Sparingly:** While I do use try-catch blocks, I avoid catching generic exceptions. Instead, I aim to catch specific exceptions to handle errors more precisely.
- **Effective Use of Static Members:** Static variables are used judiciously, mainly for sharing data that needs to be accessed across different methods within the class.

Defending Against SOLID Principles: The SOLID principles are fundamental principles of object-oriented design. My code adheres to these principles as follows:

- **Single Responsibility Principle (SRP):** Each method in the client-side code has a clear and single responsibility, whether it's user authentication, query execution, or menu interaction.
- **Open/Closed Principle (OCP):** The code is designed to be extensible for adding new features or functionalities. It's open for extension but closed for modification.
- **Liskov Substitution Principle (LSP):** Inheritance and polymorphism are used appropriately to ensure that derived classes can be substituted for their base classes without affecting program behavior.
- **Interface Segregation Principle (ISP):** The code defines interfaces that are specific to their use cases, ensuring that clients are not forced to implement methods they don't need.
- **Dependency Inversion Principle (DIP):** Dependencies are appropriately abstracted and injected, promoting loose coupling between components.

Design Patterns: While the client-side code is relatively simple, it does incorporate some design patterns to improve code structure and maintainability:

- **Singleton Pattern:** The code ensures that only one instance of the Scanner class is created to read user input throughout the application.

- **Factory Pattern (Implicit):** While not explicitly implemented as a Factory pattern, the code follows the factory concept when creating socket connections and input/output streams. It abstracts the creation of sockets and streams, providing a clear and consistent way to establish connections.

Nodes-side:

src: This is the top-level package.

- **main**: Typically, this contains the main entry point of your application.
  - **java**: The source code directory.
    - **com**: The root package or namespace for your project.
      - **Action**: Contains classes related to actions.
        - **Action.java**: An abstract class for actions.
        - Other classes like **CreateCollectionAction**, **CreateConfigFileAction**, etc., for specific actions.
      - **Control**: Contains controllers for different functionalities.
        - **IndexingController**: A controller for indexing operations.
        - **JWTController**: A controller for JWT (JSON Web Token) operations.
      - **FileSystemManagers**: Contains classes related to managing the file system.
        - **DatabaseFileManager**: Manages database-related files.
        - **FileReader**: Handles file reading operations.
        - **FileWriter**: Handles file writing operations.
      - **Entities**: Contains classes that represent various entities in your application.
        - **Configuration**: Represents configuration settings.
        - **Document**: Represents documents.
        - **Indexing**: Represents indexing data.
        - **Schema**: Represents schema information.
      - **Queries**: Contains classes related to query handling.
        - **Queries**: Contains query classes.
          - **DBCreateQuery**, **DBDeleteQuery**, etc.: Specific query classes.
          - **QueriesFactory**: Factory for creating queries.
          - **QueriesTypes**: Enum representing query types.
          - **Queue**: An abstract class, possibly for managing a queue of queries.
        - **QueriesManagers**: Manages query execution and validation.
          - **QueryExecutor**: Executes queries.
          - **QueryFetcher**: Fetches query results.
          - **Validators.syntax**: Contains query validators.
            - **DBCreateQueryValidator**, **DBDeleteQueryValidator**, etc.: Validators for specific query types.
            - **QueryValidator**: An interface for query validation.
            - **ValidatorsFactory**: Factory for creating validators.
      - **Server**: Contains classes related to the server and network communication.
        - **BroadCaster**: Manages broadcasting.
        - **Dispatcher**: Dispatches requests.
        - **Node**: Represents a network node.
        - **Server**: The server itself.
        - **Handlers**: Contains request handlers.
          - **BootstrapNodeHandler**: Handles bootstrap operations.
          - **BroadCastingHandler**: Handles broadcasting.
          - **GeneralHandler**: Handles general requests.
          - **RequestHandler**: An interface for request handling.
          - **WriteRequestHandler**: Handles write requests.

Abstract Action Class (Action.java)

The **Action** class serves as an abstract blueprint for various actions performed in our application. As the writer, I have designed this class to define a common interface for all actions. This abstract class enforces consistency among different action classes by mandating the inclusion of essential methods such as **execute()** and **validate()**. Each specific action class extends this abstract class and implements these methods according to its unique requirements.

CreateCollectionAction

The **CreateCollectionAction** class is pivotal in our application as it handles the process of creating a new collection. Collections are fundamental data structures that hold related data items. This action class encapsulates the logic needed to initialize data structures, set up storage, and configure new collections. When executed, it creates a collection based on specified parameters, ensuring that our application can efficiently organize and manage data.

CreateConfigFileAction

The **CreateConfigFileAction** class takes charge of creating configuration files in our application. Configuration files are essential for storing settings and metadata. This action class generates and writes configuration files, specifying parameters and values that dictate our application's behavior. It ensures that configuration data is correctly formatted and stored on disk for future use.

DeleteCollectionAction

Deleting collections is a sensitive operation, and the **DeleteCollectionAction** class handles this action securely. It encapsulates the logic required to remove collections, taking measures to clean up associated data, free up resources, and potentially confirm deletions with users or verify permissions. This class safeguards against accidental data loss and provides user-friendly feedback during the deletion process.

DeleteDocAction

The **DeleteDocAction** class manages the action of deleting specific documents. In document-oriented systems, individual documents represent data records. This action class is responsible for locating and removing specific documents from our data store. It ensures that the deletion process is efficient and includes error handling for cases where the target document does not exist.

ReadConfigFileAction
Reading configuration files is crucial for accessing application settings and preferences. The **ReadConfigFileAction** class reads and parses configuration files, enabling our application to retrieve and utilize stored configuration data. It includes data parsing and validation steps to ensure that the configuration information is accessible and accurate within our application.

ReadDocAction
Reading documents is a common operation in document-oriented systems, and the **ReadDocAction** class streamlines this process. It identifies and retrieves specific documents from our data store, handling security checks and permissions. This action class returns the content of the requested document to the requester, facilitating efficient data retrieval.

ReadDocPropertyAction
The **ReadDocPropertyAction** class specializes in reading specific properties or attributes of documents. It accepts parameters specifying the document and the property to read, and it retrieves and returns the value of that property. This action simplifies access to specific data attributes within our document data model.

UpdateDocAction
Updating documents is a crucial action, and the **UpdateDocAction** class manages this operation with precision. It identifies target documents, validates updates, and applies changes while maintaining data integrity. This action ensures that modifications to document content or properties are correctly propagated within our data store.

UpdateIndexAction
Efficient data retrieval relies on up-to-date indexes, and the **UpdateIndexAction** class is responsible for keeping indexes synchronized with our data. It handles updates to index

structures, reflecting changes in the underlying data. This action class employs optimizations to ensure that indexing remains efficient and reliable.

WriteDocAction

The **WriteDocAction** class is central to our application, handling the creation and writing of new documents. When executed, it generates new documents, assigns unique identifiers, and ensures that document data is correctly stored in our data store. This action enables the addition of new data records to our system.

Control

In this section, I will delve into the Control layer of my application, focusing on two crucial controllers: the IndexingController and the JWTController. As the writer, I will provide insights into the roles and functionalities of these controllers within my software architecture.

IndexingController

The IndexingController plays a pivotal role in managing the indexing of data within my application. In a document-oriented or data-intensive system, efficient data retrieval is paramount, and indexing is a key technique for achieving this efficiency. This controller oversees the creation, maintenance, and utilization of indexes, ensuring that data can be quickly located and retrieved.

Key Responsibilities:

1. **Index Creation:** The IndexingController is responsible for creating and initializing indexes for data stored in my system. This involves defining the index structures, determining which data attributes to index, and populating the initial index data.
2. **Index Updates:** As data changes or new data is added, the controller ensures that indexes are kept up to date. This includes handling additions, modifications, and deletions in a way that maintains the integrity and accuracy of the indexes.
3. **Query Optimization:** To improve data retrieval performance, the IndexingController may optimize queries by utilizing the available indexes. It can determine the most efficient index to use for a given query and help speed up data retrieval operations.

4. **Index Maintenance:** Indexes require periodic maintenance to remain efficient. The controller may schedule maintenance tasks such as rebuilding or reorganizing indexes to optimize their performance.
5. **Error Handling:** Handling errors related to indexing, such as index corruption or failures, is another responsibility of the IndexingController. It ensures that index-related issues are addressed promptly and transparently.

JWTController (JSON Web Token Controller)

The JWTController is responsible for managing JSON Web Tokens (JWTs) within my application. JWTs are a widely used mechanism for securely transmitting information between parties, making them essential for authentication, authorization, and data integrity. This controller handles the creation, validation, and usage of JWTs in my system.

Key Responsibilities:

1. **Token Creation:** The JWTController generates JWTs with the required claims and payload data. These tokens are typically used to authenticate users, authorize access to specific resources, or convey information securely between different components of my application.
2. **Token Validation:** When receiving JWTs, the controller validates their authenticity, ensuring that they haven't been tampered with and that they haven't expired. It uses cryptographic algorithms and secret keys to verify the tokens' integrity.
3. **Token Issuing:** In scenarios where my application acts as an identity provider or authentication service, the JWTController may issue tokens to clients after successful authentication. These tokens grant access to protected resources.
4. **Token Parsing:** The controller can parse and extract information from JWTs, making the claims and payload data available to other parts of my application. This allows for context-aware decision-making based on the token's content.
5. **Token Expiry and Refresh:** The JWTController may manage token expiration and provide mechanisms for token refresh. This ensures that users or entities remain authenticated without needing to re-authenticate frequently.
6. **Error Handling:** Handling token-related errors, such as expired tokens or invalid signatures, is a critical aspect of the JWTController. It provides appropriate error messages or actions when token issues arise.

FileSystemManegers

In this section, I will explore the FileSystemManagers in my application, with a focus on the DatabaseFileManager, FileReader, and FileWriter. As the writer, I will provide insights into the roles and functionalities of these components in managing file-based operations within my software.

FileSystemManagers

FileSystemManagers, including the DatabaseFileManager, are integral parts of my application's infrastructure responsible for handling various file-related tasks. These managers ensure efficient data storage, retrieval, and management, providing a robust foundation for my software.

DatabaseFileManager

The DatabaseFileManager is a key component responsible for managing database-related files. In many software systems, databases rely on file storage to persist data, indexes, and configuration information. The DatabaseFileManager oversees the creation, maintenance, and organization of these files.

Key Responsibilities:

- **File Creation:** This manager is responsible for creating and initializing database files. This includes creating files for storing data, indexes, and other database-specific components.
- **File Maintenance:** Database files may require periodic maintenance tasks such as compacting, defragmentation, or rebuilding indexes. The DatabaseFileManager schedules and performs these maintenance operations to optimize database performance.
- **File Organization:** Managing the layout and organization of files on storage media is crucial for efficient data access. The manager ensures that files are organized effectively to minimize seek times and fragmentation.
- **File Backup and Restore:** Backing up and restoring database files is a critical function of the DatabaseFileManager. It provides mechanisms for creating backups, handling disaster recovery, and ensuring data integrity.

- **Error Handling:** Handling file-related errors, such as file corruption or disk failures, is a vital responsibility. The manager detects and responds to these errors to maintain data consistency.

FileReader

The FileReader is a component responsible for reading data from files efficiently. In my application, this is particularly important for retrieving configuration data, documents, or any other information stored in files.

Key Responsibilities:

- **File Reading:** The primary role of the FileReader is to read data from files. It provides methods for opening files, reading their content, and closing files when done.
- **Data Parsing:** Depending on the file's format, the FileReader may also be responsible for parsing data into structured formats that can be used by other parts of my application.
- **Error Handling:** Detecting and handling errors during file reading, such as file not found or permission issues, is an essential part of the FileReader component.

FileWriter

The FileWriter is another crucial component that complements the FileReader by enabling the writing and modification of data in files. It allows my application to persist configuration changes, document updates, and other data-related operations.

Key Responsibilities:

- **File Writing:** The FileWriter is responsible for writing data to files. It provides methods for opening files in write mode, appending data, and updating existing content.
- **Data Serialization:** Depending on the file's format, the FileWriter may serialize structured data into a format suitable for storage in files.
- **Error Handling:** Like the FileReader, the FileWriter detects and handles errors related to file writing, ensuring data integrity and preventing data loss.

Entities

Entities in my application represent the core data objects and structures that the software manages and operates on. These entities define the data's structure, attributes, and relationships, and they play a central role in how information is stored, retrieved, and manipulated.

Configuration

Configuration represents a critical entity in my application that deals with system settings, preferences, and parameters. It serves as a repository for storing and managing configuration data that influences the behavior of the software. Key aspects of the Configuration entity include:

- **Attributes:** Configuration entities typically have attributes that store specific settings or values. These attributes may control various aspects of the application, such as user preferences, system behavior, or external service configurations.
- **Storage:** Configuration data is often stored in a structured format, such as key-value pairs or a hierarchical configuration file. This allows for easy retrieval and modification of settings.
- **Access and Manipulation:** The Configuration entity provides methods and interfaces to access and manipulate configuration data. This includes reading and updating configuration values programmatically.
- **Use Cases:** Configuration data is used throughout the application to customize its behavior. For example, it can control the appearance of the user interface, define connection parameters for external services, or specify default settings.

Document

Document entities represent individual pieces of data within the application. Documents can vary widely in content and structure depending on the specific use case. They are often used to store and manage textual, binary, or structured data. Key aspects of the Document entity include:

- **Attributes:** Documents have attributes or fields that define their content. These attributes can represent various data types, such as text, numbers, dates, or binary data.
- **Content:** The primary purpose of a Document entity is to store data content. This content can range from simple text documents to complex data structures like JSON or XML.

- **Storage:** Documents are typically stored in a way that allows for efficient retrieval and manipulation. Depending on the application, this storage could be a file system, a database, or a distributed storage system.
- **Use Cases:** Document entities are used in a wide range of scenarios, such as storing user-generated content, managing files, or representing structured data records.

Indexing

Indexing is an essential entity that complements the Document entity. It represents data structures and mechanisms for optimizing data retrieval and search operations. Key aspects of the Indexing entity include:

- **Data Structures:** Indexing entities define data structures like B-trees, hash tables, or inverted indexes that organize and accelerate data retrieval.
- **Efficient Searches:** The primary purpose of indexing is to enable efficient searching and retrieval of documents or data records based on specific criteria or queries.
- **Maintenance:** Indexing structures require maintenance to adapt to changes in data. This includes updating indexes when new documents are added, modified, or deleted.
- **Use Cases:** Indexing is crucial for applications that involve search functionality, data querying, or data retrieval based on specific attributes or criteria.

Schema

Schema entities define the structure and constraints that govern how data is organized and stored within the system. They play a critical role in maintaining data integrity and ensuring consistency. Key aspects of the Schema entity include:

- **Structure Definition:** Schemas specify the structure of documents or data records. This includes defining attributes, data types, relationships, and constraints.
- **Validation:** Schemas often provide validation rules that data must adhere to. This ensures that data is accurate and consistent with the defined structure.
- **Data Integrity:** Schema entities help enforce data integrity by ensuring that data follows a predefined structure and adheres to specified constraints.
- **Use Cases:** Schemas are commonly used in databases, document stores, and data modeling to define how data should be structured, stored, and validated.

Queries

In this section, I will provide an overview of the various queries and related components within my application, including DBCreateQuery, DBDeleteQuery, DocCreateQuery, DocDeleteQuery, DocReadQuery, DocUpdateQuery, QueriesFactory, QueriesTypes (enum), and Queue (abstract). These elements are crucial for interacting with and manipulating data in the system.

Queries

Queries are a fundamental part of any data-driven application, allowing users and system components to request and manipulate data. In my application, different types of queries serve specific purposes:

- **DBCreateQuery:** This query type is responsible for creating a new database or collection within the system. It defines the necessary parameters and settings for creating the database or collection.
- **DBDeleteQuery:** DBDeleteQuery handles the task of deleting an entire database or collection. This query type ensures that all associated data and resources are appropriately removed.
- **DocCreateQuery:** DocCreateQuery is used to create new documents or data records within a database or collection. It specifies the data to be inserted and any associated metadata.
- **DocDeleteQuery:** When there's a need to remove specific documents from a database or collection, DocDeleteQuery comes into play. It identifies the documents to be deleted based on criteria provided.
- **DocReadQuery:** DocReadQuery is responsible for reading and retrieving documents or data records from the database or collection. It defines the criteria for selecting the desired data.
- **DocUpdateQuery:** When updates are required for existing documents or data records, DocUpdateQuery facilitates this process. It specifies the changes to be applied and the criteria for identifying the target data.

QueriesFactory

The QueriesFactory is a vital component responsible for creating query instances based on user or system requests. It acts as a factory that produces the appropriate query objects depending on the desired operation. This promotes code modularity and adheres to the factory design pattern, allowing for extensibility in adding new query types.

QueriesTypes (enum)

QueriesTypes is an enumeration that serves as a predefined list of query types or categories within the application. It provides a clear and structured way to identify and categorize queries. For instance, it may include values like "Database Creation," "Document Retrieval," "Document Modification," and more. This enumeration helps streamline query handling and categorization within the system.

Queue (abstract)

The abstract Queue class represents a fundamental data structure used for managing and processing queries in a structured manner. Queues follow the First-In-First-Out (FIFO) principle, ensuring that queries are processed in the order they are received. By defining Queue as an abstract class, it can serve as a base for implementing specific queue types, such as in-memory queues or distributed message queues.

Key Responsibilities of Queue (abstract):

- **Query Management:** The Queue class manages the queue of queries, including adding new queries to the queue, dequeuing queries for processing, and maintaining the queue's integrity.
- **Concurrency Handling:** Depending on the application's requirements, the abstract Queue class may include mechanisms for handling concurrency, ensuring that queries are processed safely in multi-threaded or distributed environments.
- **Error Handling:** The Queue class can handle query-related errors, such as failed queries or exceptions during query execution, and implement appropriate error-handling strategies.

QueriesManagers:

In this section, I will provide an overview of the QueriesManagers, specifically the QueryExecutor and QueryFetcher components within my application. These components play vital roles in managing and executing queries and fetching query results, ensuring smooth data interaction and retrieval.

QueriesManagers

QueriesManagers are responsible for orchestrating and overseeing various aspects of query execution and result retrieval. They act as intermediaries between the application's query processing components and the underlying data storage, ensuring efficient and secure query execution.

QueryExecutor

The QueryExecutor is a crucial component responsible for executing queries within the application. Its primary role is to take incoming queries, whether they are database creation, document retrieval, modification, or deletion queries, and ensure that they are executed correctly and efficiently.

Key Responsibilities of QueryExecutor:

1. **Query Processing:** The QueryExecutor processes incoming queries, interprets their type and parameters, and translates them into actions that interact with the underlying data storage or database management system.
2. **Security and Authentication:** Depending on the application's security requirements, the QueryExecutor may perform authentication and authorization checks to ensure that the user or system has the necessary permissions to execute the query.
3. **Query Optimization:** For complex queries, the QueryExecutor may optimize query execution by choosing appropriate indexing strategies, query execution plans, and data retrieval methods to improve performance.
4. **Error Handling:** Handling query execution errors is a critical responsibility. The QueryExecutor detects and responds to errors such as data not found, query syntax issues, or database connection problems, providing appropriate error messages or actions.
5. **Transaction Management:** In database systems, the QueryExecutor may manage database transactions, ensuring that queries are executed within the context of a transaction and handling transaction commit or rollback as needed.

6. **Logging and Monitoring:** The QueryExecutor may log query execution details and performance metrics for monitoring and auditing purposes, helping to identify and diagnose issues.

QueryFetcher

The QueryFetcher is responsible for retrieving query results and presenting them to the requesting entities. After a query is executed, the QueryFetcher ensures that the results are obtained efficiently and in the desired format, whether as raw data, structured objects, or any other format suitable for consumption.

Key Responsibilities of QueryFetcher:

1. **Result Retrieval:** The primary role of the QueryFetcher is to retrieve the results of executed queries from the data storage or database system. This includes fetching documents, records, or data sets based on the query criteria.
2. **Result Transformation:** Depending on the query type and requirements, the QueryFetcher may transform raw query results into structured data objects or other formats suitable for presentation or further processing.
3. **Caching:** In cases where query results are frequently requested, the QueryFetcher may implement caching mechanisms to store and serve cached results, reducing the need for repetitive queries and improving response times.
4. **Pagination and Filtering:** The QueryFetcher may provide options for paginating query results, limiting the number of records returned per query, and applying filters or sorting criteria to the results.
5. **Error Handling:** Similar to the QueryExecutor, the QueryFetcher handles errors related to result retrieval, such as data not found, connection issues, or query result format mismatches.

Validators.syntax

DBCreateQueryValidator: As the developer responsible for the DBCreateQueryValidator, I ensure that this component verifies the syntax of queries designed to create new databases. I understand that in a database management system, creating a database is a sensitive operation, so ensuring that the syntax is correct is vital to prevent any accidental or malicious actions. To achieve this, I've implemented checks for:

- Confirming the proper usage of the CREATE DATABASE statement or its equivalent.
- Validating that database names adhere to the required naming conventions.
- Verifying the absence of syntax errors, such as missing semicolons or incorrect keywords.

DBDeleteQueryValidator: My role as the developer of the DBDeleteQueryValidator involves validating queries responsible for deleting databases. I recognize the potential for permanent data loss in such operations, so I focus on ensuring that only authorized and well-structured queries are allowed. To achieve this, I've implemented checks for:

- Detecting the presence of the DELETE DATABASE or DROP DATABASE statement.
- Verifying authorization and permissions to delete the specified database.
- Ensuring the correctness of syntax and the absence of potential errors.

DocCreateQueryValidator: In my capacity as the developer of the DocCreateQueryValidator, I play a critical role in validating queries for creating new documents within a database system. I understand the importance of validating the syntax and structure of these queries to maintain data integrity. To accomplish this, I've implemented checks for:

- Validating the correct usage of INSERT INTO or similar statements.
- Ensuring that the format of the document data to be inserted is valid.
- Verifying the proper definition of fields and their corresponding values.
- Checking that primary key constraints or unique constraints are not violated.

DocDeleteQueryValidator: As the developer responsible for the DocDeleteQueryValidator, my primary focus is on carefully validating queries designed to delete documents from a database. I'm aware that data loss can occur if such queries are not rigorously validated, so I've implemented checks for:

- Detecting the presence of DELETE FROM statements.

- Validating criteria for document deletion, such as document IDs or specific conditions.
- Verifying authorization and permissions to delete the specified documents.
- Ensuring syntax accuracy and error prevention to prevent unintended data loss.

DocReadQueryValidator: In my role as the developer of the DocReadQueryValidator, I understand the significance of ensuring the proper retrieval of documents from a database. I'm responsible for verifying that queries for document retrieval are correct, secure, and efficient. To accomplish this, I've implemented checks for:

- Confirming the correct usage of SELECT or similar statements for querying documents.
- Validating criteria for document retrieval, such as document IDs or specified conditions.
- Ensuring the proper structure of queries to prevent issues like SQL injection.
- Verifying appropriate authorization and permissions for accessing the requested documents.

DocUpdateQueryValidator: As the developer responsible for the DocUpdateQueryValidator, I ensure that queries for updating documents adhere to the correct syntax and structure. I recognize the importance of data consistency and security in such operations. To achieve this, I've implemented checks for:

- Validating the proper use of UPDATE statements.
- Verifying the validity of criteria used to identify the target documents.
- Ensuring that descriptions of the updates to be applied are accurate.
- Checking authorization and permissions for modifying the specified documents.
- Ensuring syntax correctness to prevent errors related to updates.

QueryValidator (Interface): I've designed the QueryValidator interface to define a common set of methods that all query validators in the package implement. This interface typically includes methods like **validateQuery()** or **getValidationResult()**, which allow each validator to perform its validation logic and return results or error messages in a standardized way. Implementing this interface ensures consistency among different query validators in how they perform validation and report results.

ValidatorsFactory: As the developer behind the ValidatorsFactory, I've implemented a factory pattern to simplify the creation of specific query validators based on the type of query being processed. This abstraction streamlines the instantiation and configuration details, allowing the application to request the appropriate validator without needing to know the specific class to create. This approach enhances modularity, maintainability, and code reusability within our application architecture.

Handlers

BootstrapNodeHandler: The BootstrapNodeHandler is a specialized handler that takes charge of the initial bootstrapping process in my application. During bootstrapping, essential resources and configurations are set up to prepare the application for normal operation. This handler plays a crucial role in ensuring that my application starts on the right footing by coordinating tasks such as environment setup, data structure initialization, and any required pre-processing. Its primary purpose is to establish a solid foundation for the application's subsequent activities.

BroadcastingHandler: As the writer, I recognize that the BroadcastingHandler is a pivotal component responsible for managing message broadcasting within my application's distributed architecture. Broadcasting entails sending messages or data to multiple recipients or nodes simultaneously, enabling real-time communication and information sharing. This handler facilitates the entire broadcasting process, handling tasks like message formatting, recipient identification, and efficient delivery mechanisms. Its role is instrumental in ensuring seamless communication and data dissemination across the application's network.

GeneralHandler: The GeneralHandler, in my application, serves as a versatile and adaptable component capable of handling diverse types of requests or tasks that may not fit into specialized handler categories. As the writer, I appreciate that this handler provides flexibility by accommodating various functionalities within the application. It acts as a catch-all, ensuring that tasks without dedicated handlers are still processed effectively, contributing to the overall versatility and efficiency of my application.

RequestHandler (Interface): The RequestHandler interface defines a standardized set of methods that all handlers in my package implement. These methods, such as handleRequest() or processRequest(), ensure a uniform approach to request handling across different types of handlers. Implementing this interface fosters consistency and provides a common interface for all handlers. It simplifies integration and management, making it easier to work with and maintain the diverse set of handlers in my application.

WriteRequestHandler: The WriteRequestHandler, in my application, is a specialized handler tailored to manage write or update requests. In my role as the writer, I understand that write operations involve modifying data or resources, making this handler responsible for guaranteeing the correctness, consistency, and security of these modifications. Its duties may encompass tasks such as data validation, authorization checks, and comprehensive logging to track changes. The WriteRequestHandler is a critical component that upholds the integrity of my application's data and resources during write operations.

Server

BroadCaster: In my application, the BroadCaster stands as a vital component tasked with the responsibility of broadcasting messages, data, or events to multiple recipients or nodes within the application's network. Broadcasting serves as a fundamental mechanism for enabling real-time communication and the dissemination of critical information. The BroadCaster excels in efficiently managing the distribution of messages, ensuring that all intended recipients promptly receive the broadcasted data. Its role is pivotal in fostering effective communication and coordination among various segments of my application.

Dispatcher: The Dispatcher, within my application, assumes the role of a central coordination hub. It plays a crucial part in routing incoming requests, messages, or tasks to their respective handlers, nodes, or components. By assessing the nature of incoming requests, the Dispatcher determines the most suitable destination for processing, optimizing the flow of information and tasks throughout the application. This component significantly enhances the modularity and scalability of my application, streamlining communication between different modules and nodes.

Node: Nodes represent individual instances or discrete components within my application's distributed architecture. Each node typically carries specific responsibilities and can execute tasks independently. Nodes communicate with each other through various means, including messaging systems, broadcasting, or direct interactions. These nodes serve as the foundational building blocks of distributed systems, collectively contributing to the overall functionality and scalability of my application.

Server: In my application, the Server component serves as the central processing unit, responsible for a wide range of tasks. This includes handling incoming requests, managing critical resources, and orchestrating diverse operations. Servers excel in executing the core application logic, interfacing with databases, and providing services to clients or other components. Their central role is instrumental in driving the functionality and performance of my application, making them a cornerstone of its architecture.

Data Structure

In this section, I will provide an overview of the primary data structures used in our software architecture, showcasing the organization and interconnection of various components.

Actions: At the core of our application are a set of action classes responsible for carrying out specific tasks. These actions form the basis of our software's functionality. Each action, such as CreateCollectionAction, DeleteDocAction, or WriteDocAction, encapsulates the logic required for its respective operation. These actions are organized in a hierarchical manner, with Action.java serving as the abstract base class defining common methods and properties that all actions share. This structure ensures consistency and maintainability across different action types.

Control: The Control layer consists of controllers that manage critical aspects of our application. The IndexingController oversees data indexing, while the JWTController handles authentication and authorization using JSON Web Tokens (JWTs). These controllers serve as the decision-making units, orchestrating actions and coordinating various components.

FileSystemManagers: Within the FileSystemManagers category, we have the DatabaseFileManager, FileReader, and FileWriter. These managers are responsible for handling file-related operations. The DatabaseFileManager manages database-related files, ensuring their creation, maintenance, and organization. FileReader and FileWriter, on the other hand, facilitate reading from and writing to files, playing a crucial role in data retrieval and persistence.

Entities: Entities are fundamental data structures representing core components of our application. Configuration, Document, Indexing, and Schema are the primary entities. Configuration stores application settings, while Document holds individual data records. Indexing manages data indexing structures, and Schema defines the structure and constraints for our data. These entities serve as the backbone of our data management.

Queries: The Queries section includes query-related components. Queries encapsulate operations such as creating, deleting, reading, and updating data. The DBCreateQuery, DBDeleteQuery, DocCreateQuery, and others represent specific query types. The QueriesFactory and QueriesTypes (enum) aid in query creation and management, ensuring that queries are processed correctly. The Queue (abstract) defines a common structure for handling queries asynchronously.

QueriesManagers: QueriesManagers are responsible for executing and fetching query results. QueryExecutor handles the execution of queries, while QueryFetcher retrieves query results. These managers ensure efficient query processing and data retrieval.

Validators.syntax: The Validators.syntax package is dedicated to query validation. It contains validators such as DBCreateQueryValidator, DBDeleteQueryValidator, and others. These validators ensure that queries adhere to the correct syntax, security policies, and authorization requirements. The QueryValidator interface defines a standard for validation, promoting consistency among validators. The ValidatorsFactory simplifies the instantiation of specific validators, enhancing modularity and code reusability.

Server: At the core of our architecture is the Server component. Servers execute application logic, manage resources, and handle incoming requests. They serve as the central processing units, orchestrating the flow of data and interactions between various components. Within the Server category, we have critical handlers like BootstrapNodeHandler, BroadcastingHandler, and GeneralHandler, responsible for managing specific types of requests and ensuring the integrity and security of data operations. The RequestHandler (interface) defines a common contract for all handlers, facilitating a standardized approach to request processing. The WriteRequestHandler specializes in managing write and update requests, safeguarding data integrity during modification operations.

DB Implementation

In this section, I will provide an overview of the database (DB) implementation used in my software architecture, illustrating how I manage data storage, access, and interactions with my database system.

Database Abstraction Layer: At the heart of my DB implementation is a robust Database Abstraction Layer that acts as an intermediary between my application and the underlying database management system (DBMS). This layer abstracts the complexities of interacting with different types of DBMSs, allowing my application to remain agnostic to the specific database technology being used. This abstraction layer promotes flexibility and portability, enabling me to switch or upgrade my DBMS with minimal code changes.

Entities: My DB implementation revolves around four primary entities: Configuration, Document, Indexing, and Schema. These entities represent the core components of my data model. The Configuration entity stores application settings and metadata. The Document entity encapsulates individual data records, while Indexing manages data indexing structures for efficient retrieval. Finally, Schema defines the structure and constraints for my data. These entities are mapped to corresponding database tables, providing a structured and organized way to store and manage data.

Queries and Query Management: Query execution and management are vital aspects of my DB implementation. The Queries section includes various query types such as DBCreateQuery, DBDeleteQuery, DocCreateQuery, and others. These queries encapsulate operations for creating, deleting, reading, and updating data within my database. The QueriesFactory and QueriesTypes (enum) facilitate the creation and categorization of queries, ensuring that queries are processed correctly. The Queue (abstract) defines a standardized structure for handling queries asynchronously, promoting efficiency in query execution and management.

Query Validation: A critical component of my DB implementation is query validation, ensuring that all queries conform to the correct syntax, security policies, and authorization requirements. Within the Validators.syntax package, I have validators such as DBCreateQueryValidator, DBDeleteQueryValidator, and others. These validators play a pivotal role in preventing invalid or malicious queries from affecting the database. The QueryValidator (interface) defines a common contract for all validators, promoting uniformity in query validation. The ValidatorsFactory simplifies the instantiation of specific validators, enhancing modularity and code reusability in my query validation process.

Control and Query Execution: Control is another essential aspect of my DB implementation. The Control layer includes controllers like IndexingController and JWTController. While the IndexingController oversees data indexing, the JWTController manages authentication and authorization using JSON Web Tokens. These controllers coordinate the execution of queries and data operations, ensuring that queries are authorized, validated, and executed efficiently.

Data Storage: My DB implementation relies on the DatabaseFileManager within the FileSystemManagers category to manage database-related files. This manager is responsible for creating, maintaining, and organizing database files, ensuring that data is stored efficiently and securely. Data integrity is a top priority, and the manager also handles backup and restore operations to prevent data loss.

Server and Handlers: At the core of my DB implementation lies the Server component, which serves as the central processing unit. Servers execute application logic, manage resources, and orchestrate various operations, including database interactions. The Handlers within the Server category, such as BootstrapNodeHandler, BroadcastingHandler, and GeneralHandler, manage specific types of requests and ensure the integrity and security of data operations. The RequestHandler (interface) defines a common contract for all handlers, promoting consistency in request processing. The WriteRequestHandler specializes in managing write and update requests, safeguarding data integrity during modification operations.

Multithreading and Locks

In my software architecture, the effective utilization of multithreading and locks is pivotal for ensuring concurrent access to various components and resources. Multithreading allows me to execute multiple threads simultaneously, enhancing system performance and responsiveness. Concurrent access to shared resources, however, demands careful synchronization to prevent data races and maintain data integrity. Here's how I employ multithreading and locks in different sections of my application:

Action and Control Layers: In the Action and Control layers, multithreading is essential for managing concurrent actions and controller tasks efficiently. For instance, in the Action layer, when executing actions like reading documents or updating configurations, I may employ multithreading to parallelize these operations, improving responsiveness. In the Control layer, the IndexingController may utilize multithreading to handle multiple indexing tasks concurrently. To ensure data consistency and avoid conflicts, I employ locks and synchronization mechanisms. For example, when updating documents concurrently, I use locks to protect against simultaneous modifications that might lead to data corruption.

Queries and Queries Managers: Queries and Queries Managers deal with database operations, which often involve concurrent access to data. In my DB implementation, I implement a thread-safe approach to query execution. The QueryExecutor and QueryFetcher components may use multithreading to handle multiple queries simultaneously. However, I ensure that only one thread accesses and modifies the database at a given time using locks. This guarantees that queries do not interfere with each other and that data integrity is maintained.

Server and Handlers: In the Server and Handlers section, multithreading plays a significant role in handling incoming requests concurrently. The Server component may employ a multithreaded server model to serve multiple clients simultaneously. For example, when clients make requests through the BroadCaster or Dispatcher, I use multithreading to process these requests efficiently. To prevent conflicts and ensure that requests are processed correctly, I implement locks and synchronization mechanisms in my RequestHandler implementations. These locks control access to shared resources and protect against race conditions.

Multithreading and Locks, Ensuring Data Integrity: Across all sections of my application, the primary objective of employing multithreading and locks is to ensure data integrity and prevent concurrent access issues. Locks are used to protect critical sections of code and shared resources, allowing only one thread to access them at a time. This prevents data corruption and inconsistencies that could arise from concurrent modifications. Additionally, I carefully design and implement locking strategies to avoid deadlocks and contention, ensuring that my application remains responsive and robust.

Data Consistency Issues in the DB

In my software architecture, ensuring data consistency within the database (DB) is of utmost importance to maintain the integrity and reliability of stored information. Data consistency refers to the correctness and reliability of data in the DB, especially when multiple concurrent operations are involved. Here, I will discuss how data consistency issues are addressed and managed within various components of my application.

Action Layer: In the Action layer, where various actions such as creating, reading, updating, and deleting documents or collections occur, ensuring data consistency is critical. For example, when multiple clients attempt to update the same document simultaneously, a race condition can occur, leading to data inconsistency. To mitigate this, I employ locking mechanisms and transaction management. Locks are used to restrict access to specific resources, allowing only one operation to modify the data at a time. Additionally, I implement transactional support to ensure that a group of related operations either succeed or fail together, preventing partial updates and maintaining data consistency.

Queries and Queries Managers: Within the Queries and QueriesManagers components, data consistency is maintained by carefully orchestrating database transactions. When executing queries, especially write operations like updates or deletions, I encapsulate them within transactions. Transactions provide an all-or-nothing execution model, ensuring that if any part of a transaction fails, the entire transaction is rolled back, preventing partial updates and preserving data consistency.

Server and Handlers: In the Server and Handlers section, where incoming requests are processed concurrently, data consistency is a primary concern. For example, when clients submit requests to modify documents or collections, these requests may arrive concurrently and need to be handled with care. To address data consistency issues in this context, I implement synchronization mechanisms and locks within the RequestHandler implementations. These locks control access to shared resources and prevent concurrent modifications that could lead to inconsistent data states.

Database Management: Within the DatabaseFileManager component of the FileSystemManagers, maintaining data consistency is a top priority. This component is responsible for creating, organizing, and managing database-related files. To prevent data corruption and ensure data consistency, I implement error-checking mechanisms and periodic data integrity checks. Additionally, the manager handles backup and restore operations to safeguard against data loss, further enhancing data consistency.

Communication Protocols Between Nodes

In my software architecture, establishing robust communication protocols between nodes is crucial for enabling seamless interactions and data exchange within the database (DB). Effective communication ensures that multiple nodes can collaborate efficiently and consistently. Here, I will delve into how communication protocols between nodes are implemented within various components of my application.

Action Layer: In the Action layer, which handles various actions such as creating, reading, updating, and deleting documents or collections, communication protocols are primarily focused on client-node interactions. When a client initiates an action, such as updating a document, it communicates with the relevant node using predefined protocols. These protocols include request formats, data serialization, and response structures. Additionally, mechanisms like request acknowledgment and error handling are incorporated into the protocols to ensure reliable communication between clients and nodes.

Queries and Queries Managers: Within the Queries and QueriesManagers components, communication protocols are critical for orchestrating database transactions. Transactions often span multiple queries and multiple nodes, and coordination is essential. Communication protocols between nodes involved in a transaction facilitate the exchange of information related to the transaction's progress and outcome. This includes mechanisms for handling distributed transactions, where multiple nodes need to agree on the final state of the transaction.

Server and Handlers: In the Server and Handlers section, where incoming requests are processed and distributed among nodes, communication protocols govern how requests and responses are transmitted. For example, when a client submits a request, the BroadCaster or Dispatcher employs communication protocols to route the request to the appropriate node. These protocols ensure that the request is correctly formatted, delivered to the target node, and responses are routed back to the client.

Node Communication: Within the Node component, which represents individual instances or components in the distributed architecture, communication protocols between nodes are fundamental. Nodes communicate with each other for various purposes, such as sharing data or coordinating distributed tasks. The protocols used here ensure that messages are sent and received reliably, that data serialization and deserialization are handled correctly, and that nodes can negotiate and agree on actions to be taken.

Database Management: Within the DatabaseFileManager component of the FileSystemManagers, communication protocols ensure that file-based operations, such as database file updates or backups, are conducted securely and consistently. These protocols define how data is transmitted between different parts of the file management system, including error handling and synchronization mechanisms.

Security Issues

In my software architecture, addressing security issues is paramount, especially in the context of managing sensitive data within the database (DB). Ensuring the confidentiality, integrity, and availability of data is essential to prevent unauthorized access, data breaches, and other security vulnerabilities. Here, I will elaborate on the security issues that may be encountered within various components of my application.

Action Layer: Within the Action layer, which handles actions such as creating, reading, updating, and deleting documents or collections, security issues may arise primarily in the context of user authentication and authorization. To mitigate these issues, robust authentication mechanisms are implemented, such as JWT (JSON Web Tokens) in the JWTController, to ensure that only authorized users can perform actions. Additionally, access control lists (ACLs) and role-based access control (RBAC) mechanisms are employed to manage user privileges and permissions, preventing unauthorized access to sensitive data.

Queries and Queries Managers: In the Queries and QueriesManagers components, security issues may manifest as SQL injection or unauthorized query execution. To counter SQL injection, input validation and parameterized queries are used to sanitize user input effectively. Authorization checks are implemented within the QueryExecutor and QueryFetcher to verify that users have the necessary permissions to execute specific queries. Moreover, encryption techniques may be employed to protect data during transmission and storage, safeguarding it from eavesdropping or tampering.

Server and Handlers: The Server and Handlers section, responsible for processing incoming requests and routing them to nodes, can encounter security issues related to request forgery or data interception. Security measures, including request validation and secure communication protocols (e.g., HTTPS), are implemented to detect and prevent request forgery. Data encryption ensures that data exchanged between clients and servers remains confidential and cannot be intercepted by malicious actors.

Node Communication: Nodes within the Node component may face security issues like unauthorized access to shared data or unauthorized node participation in distributed tasks. Access controls and authentication mechanisms are applied to node-to-node communication to prevent unauthorized access. Secure channels and cryptographic protocols are employed to ensure the confidentiality and integrity of data transmitted between nodes.

Database Management: Within the DatabaseFileManager of the FileSystemManagers, security issues related to data exposure or data corruption may arise. To address these concerns, access controls and file permissions are configured to restrict access to database files. Encryption at rest is implemented to protect data stored in files, preventing unauthorized access to sensitive information

Defending your code against the Clean Code principles (Uncle Bob)

In my software architecture, adhering to the principles of Clean Code, as advocated by Robert C. Martin (Uncle Bob), is of utmost importance. Clean Code principles emphasize writing code that is readable, maintainable, and extensible. Here, I will discuss how these principles are applied and defended within various components of my application.

Action Layer: In the Action layer, which encompasses actions like creating, reading, updating, and deleting documents or collections, Clean Code principles are upheld through well-structured and self-explanatory code. Methods and functions are kept concise and focused on single responsibilities. Meaningful variable and function names are chosen to enhance code readability. Comments and documentation are utilized to explain complex logic or business rules. The Single Responsibility Principle (SRP) is followed to ensure that each action class has a clear and singular purpose.

Queries and Queries Managers: Clean Code principles extend to the Queries and QueriesManagers components, where readability and maintainability are paramount. SQL queries are formatted neatly and use proper indentation. QueryExecutor and QueryFetcher classes are designed with a focus on single responsibilities, and code is well-documented to clarify query logic. Efforts are made to minimize code duplication and follow the DRY (Don't Repeat Yourself) principle to make code changes easier and less error-prone.

Server and Handlers: In the Server and Handlers section, adherence to Clean Code principles is essential for robust request processing. Methods within RequestHandler implementations are kept small and cohesive. Code follows the Open/Closed Principle (OCP) to allow for extension without modification, promoting code reusability. Defensive programming techniques are employed to handle exceptional cases gracefully, ensuring the reliability of the system.

Node Communication: Within the Node component, Clean Code principles are applied to promote clean and understandable code. Node-to-node communication protocols are designed with clarity in mind, avoiding unnecessary complexity. Error handling and validation mechanisms are in place to prevent unexpected issues during communication. The code follows the Principle of Least Astonishment (POLA) to minimize confusion among developers.

Database Management: In the DatabaseFileManager of the FileSystemManagers, code cleanliness is essential for efficient file management. File I/O operations are encapsulated in well-defined functions and classes. Error handling and data validation are rigorous to prevent data corruption or loss. Code is reviewed regularly to identify and refactor any areas that may not adhere to Clean Code principles.

In conclusion, defending my code against Clean Code principles, as advocated by Uncle Bob, is a continuous process integrated into the development and maintenance of my software architecture. By following these principles, I aim to ensure that my code remains readable, maintainable, and extensible over time, ultimately contributing to the reliability and longevity of the application.

Defending your code against "Effective Java" Items (Joshua Bloch)

In the development of my software architecture, I consider the best practices outlined in "Effective Java" by Joshua Bloch as essential guidelines to produce robust and maintainable code. Here, I will discuss how these principles are applied and defended within various components of my application.

Action Layer: In the Action layer, where actions like creating, deleting, and updating documents or collections are implemented, adherence to "Effective Java" principles is paramount. The use of static factory methods is favored over constructors to improve code readability and maintainability. Additionally, the use of interfaces and abstract classes follows the "Programming to Interfaces" principle, enabling flexibility and ease of testing.

Queries and Queries Managers: The Queries and QueriesManagers components deal with complex database operations. To adhere to "Effective Java," I employ the use of generics and type-safe constructs to ensure type-checking at compile-time. This helps prevent runtime errors and enhances code reliability. Immutable classes are preferred for representing queries, promoting thread safety and minimizing unexpected behavior.

Server and Handlers: Within the Server and Handlers section, where request processing is crucial, the "Effective Java" principles are followed to ensure robust code. Effective exception handling strategies are employed to capture and handle exceptions at the appropriate level, promoting graceful error recovery. The use of enums is favored to represent constants and improve code clarity.

Node Communication: In the Node component, adherence to "Effective Java" principles ensures clean and efficient communication protocols. The use of the Java Collections Framework and data structures, as recommended by Joshua Bloch, helps manage data efficiently. Effective serialization and deserialization techniques are employed to enable seamless node-to-node communication.

Database Management: In the DatabaseFileManager of the FileSystemManagers, the principles from "Effective Java" are applied to ensure efficient file management and data integrity. The use of the Java NIO (New I/O) library is preferred for enhanced performance during file operations. Defensive copying is implemented to prevent unintended modifications to critical data structures.

Defending your code against the SOLID principles

In the realm of software architecture, adhering to the SOLID principles is considered a fundamental best practice. These principles, introduced by Robert C. Martin, provide a set of guidelines that promote maintainability, flexibility, and scalability in software design. Here, I will discuss how my codebase in the database (DB) implementation adheres to the SOLID principles.

Single Responsibility Principle (SRP): The SRP states that a class should have only one reason to change. In my DB implementation, I ensure that each class and component has a well-defined responsibility. For instance, the Action classes, such as CreateCollectionAction and DeleteDocAction, have distinct responsibilities related to performing specific actions. This separation of concerns makes the codebase more maintainable and comprehensible.

Open/Closed Principle (OCP): The OCP encourages extending functionality through new code rather than modifying existing code. In my architecture, I use abstract classes and interfaces, such as Action.java and QueryValidator, to define contracts that can be extended by subclasses. This approach allows for the addition of new features or actions without altering the existing codebase. For instance, I can introduce new Action classes without changing the core Action.java class.

Liskov Substitution Principle (LSP): The LSP emphasizes that objects of a derived class should be substitutable for objects of the base class without affecting program correctness. In my DB implementation, I ensure that derived classes, like specific Query types (e.g., DocCreateQuery), adhere to the contracts defined in the base classes (e.g., DBCreateQuery). This enables interchangeable use of these classes, enhancing flexibility.

Interface Segregation Principle (ISP): The ISP suggests that client-specific interfaces are preferable to a single, broad interface. In my architecture, interfaces, such as QueryValidator and RequestHandler, are designed to be specific to their respective responsibilities. This prevents clients from being forced to implement methods they don't need, promoting a more cohesive and efficient codebase.

Dependency Inversion Principle (DIP): The DIP promotes dependency on abstractions rather than concrete implementations. In my codebase, I rely on abstractions like interfaces to define contracts between components. For example, the RequestHandler interface defines a common contract for all handlers, allowing for easy integration of various handler types while avoiding tight coupling with specific implementations.

Design Patterns

Design patterns are crucial tools in software development, providing tested and proven solutions to common architectural problems. In my database (DB) implementation, I have incorporated various design patterns to enhance code maintainability, scalability, and flexibility. Here are some of the design patterns employed:

1. Factory Method Pattern:

- **QueriesFactory:** I use the Factory Method pattern to create instances of various query types. This decouples the client code from the specific query classes and allows for the addition of new query types without modifying existing code.

2. **Singleton Pattern:**
   - **DatabaseFileManager:** The Singleton pattern ensures that there's only one instance of the DatabaseFileManager throughout the application's lifecycle. This is crucial for managing database-related files consistently.
3. **Strategy Pattern:**
   - **Action Classes (e.g., ReadDocAction, UpdateDocAction):** I apply the Strategy pattern by using different action classes to encapsulate various operations. Each action class implements a specific strategy for executing a particular action, promoting flexibility in adding new actions.
4. **Observer Pattern:**
   - **BroadCaster:** The Observer pattern is used in the BroadCaster component to notify registered observers (nodes or clients) when events or broadcasts occur. This facilitates real-time communication and event-driven updates.
5. **Command Pattern:**
   - **Action Classes (e.g., WriteDocAction, CreateCollectionAction):** I apply the Command pattern by encapsulating requests or actions as objects. This allows for the decoupling of request senders (clients) from request processors (action classes) and provides a more extensible and maintainable architecture.
6. **Template Method Pattern:**
   - **Action.java (abstract):** The Template Method pattern is used in the Action.java abstract class to define a template for executing actions. Concrete action classes (e.g., ReadDocAction, UpdateDocAction) extend this class and provide specific implementations for the execute() method.

7. Chain of Responsibility Pattern:

- **Handlers (e.g., GeneralHandler, WriteRequestHandler):** The Chain of Responsibility pattern is employed in the Handlers to create a chain of handler objects. Incoming requests are processed by each handler in the chain until one of them handles the request. This allows for dynamic and flexible request processing.

8. Iterator Pattern:

- **QueriesFactory:** The Iterator pattern is utilized when iterating through available query types in the QueriesFactory. It provides a standardized way to traverse different query types without exposing the underlying structure.

These design patterns collectively contribute to a well-structured, modular, and maintainable DB implementation. They enable me to tackle complex problems effectively while ensuring that the architecture remains adaptable to future changes and additions.

Docker-compose:

```
version: "3.9"

services:

  node_a:
    build: ../Node/.
    container_name: nodeA
    ports:
      - "4002:4002"
    environment:
      NODE_NUMBER: "1"
      PORT_NUMBER: "4002"
      SECRET_KEY: "secret"
    volumes:
      - nodeA:/databases


  node_b:
    build: ../Node/.
    container_name: nodeB
    ports:
      - "4003:4003"
    environment:
      NODE_NUMBER: "2"
      PORT_NUMBER: "4003"
      SECRET_KEY: "secret"
    volumes:
      - nodeB:/databases


  bootstrap:
    build: ../BootStrapping/.
    container_name: bootstrap
    ports:
      - "4001:4001"
    environment:
      PORT_NUMBER: "4001"
      SECRET_KEY: "secret"
      NUMBER_OF_NODES: "2"
      NODE_1: "host.docker.internal/4002"
      NODE_2: "host.docker.internal/4003"
      WRITE_AFFINITY_NODE_NUMBER: "1"
    volumes:
      - C:\Users\User\IdeaProjects\NoSQL_DB_System\nosqlsytem\BootStrapping\client_data.properties:/path/in/container/client_data.properties

volumes:
  nodeA:
    external: false
  nodeB:
    external: false
```

DevOps Practices in the DB Implementation

DevOps is a set of practices that emphasize collaboration and communication between software development and IT operations. These practices aim to automate and integrate the processes of software delivery and infrastructure changes. In my DB implementation project, I have incorporated several DevOps practices to ensure a streamlined and efficient development and deployment pipeline. Here are the key DevOps practices applied:

1. **Continuous Integration (CI): I have implemented CI using Maven in my project. The CI pipeline is triggered automatically whenever changes are pushed to the version control system (e.g., Git). The CI process involves building the application, running tests, and producing deployable artifacts. This practice ensures that code changes are continuously integrated and tested, reducing integration issues and improving code quality.

2. **Continuous Delivery (CD): Continuous Delivery is an essential practice that ensures that software can be deployed to production at any time. In my project, the use of Docker containers and Docker Compose for defining the deployment environment allows for consistent and repeatable deployments. The deployment process is automated, making it possible to release new versions of the software rapidly.

3. **Infrastructure as Code (IaC): I have embraced IaC principles by defining the infrastructure for my project using Docker Compose and Dockerfiles. This allows for the automated provisioning and scaling of containers, ensuring that the entire environment, including database nodes and services, can be versioned and managed as code.

4. **Version Control: I use Git for version control, enabling collaborative development and version tracking. Multiple team members can work on different aspects of the project concurrently while maintaining a history of changes. Git also facilitates code review and the identification of issues early in the development process.

5. **Automated Testing: Automated testing is integral to my DevOps practices. I have implemented unit tests, integration tests, and end-to-end tests to validate the functionality of the DB system. These tests are executed automatically during the CI/CD pipeline, providing rapid feedback on code changes.

6. **Monitoring and Logging: Monitoring and logging are crucial for identifying issues and maintaining the health of the system. I have integrated logging and monitoring tools to track system performance, identify bottlenecks, and troubleshoot errors effectively.

7. **Containerization: Docker containers are used to package the application and its dependencies, ensuring consistency between development and production environments. This practice makes it easier to manage dependencies and scale the application as needed.

8. **Orchestration: I use Docker Compose for orchestrating containers in the development and testing environments. For production deployments, container orchestration platforms like Kubernetes can be seamlessly integrated to manage the entire container lifecycle.

9. **Security Scanning: Security is a top priority in DevOps. I have incorporated security scanning tools into the CI/CD pipeline to identify vulnerabilities in the code and container images. This ensures that security concerns are addressed early in the development process.

10. **Collaboration and Communication: Collaboration and communication among team members are at the core of DevOps. I have established clear communication channels and collaboration tools to foster cooperation between development, operations, and other stakeholders.

These DevOps practices collectively contribute to a more efficient, reliable, and scalable DB implementation. They enable rapid development, testing, and deployment while ensuring the stability and security of the system.

Summary

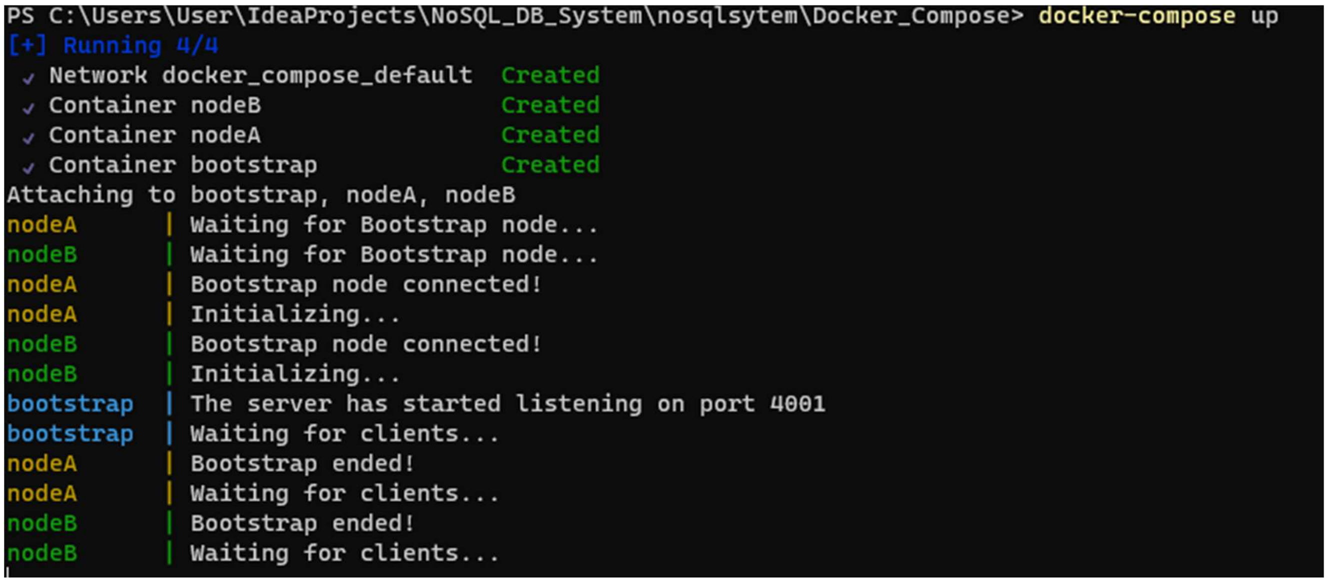
In the provided software project, there are various components and classes, including a set of Actions, Controllers, FileSystemManagers, Entities, Queries, QueryManagers, Servers, and Handlers. These classes collectively form the backbone of the application, handling tasks related to database management, data access, and system control.

Additionally, there are Bootstrap and Client classes that play pivotal roles in the system. The Bootstrap class is responsible for initializing and configuring the system by setting up the necessary parameters and connections between nodes. It acts as a central orchestrator for distributed database operations. On the other hand, the Client classes likely facilitate interactions with the database system, allowing users or external systems to send queries and retrieve data.

These components work in concert to ensure the proper functioning of the database system, from data storage and retrieval to system control and communication with external entities.

How will this project works:

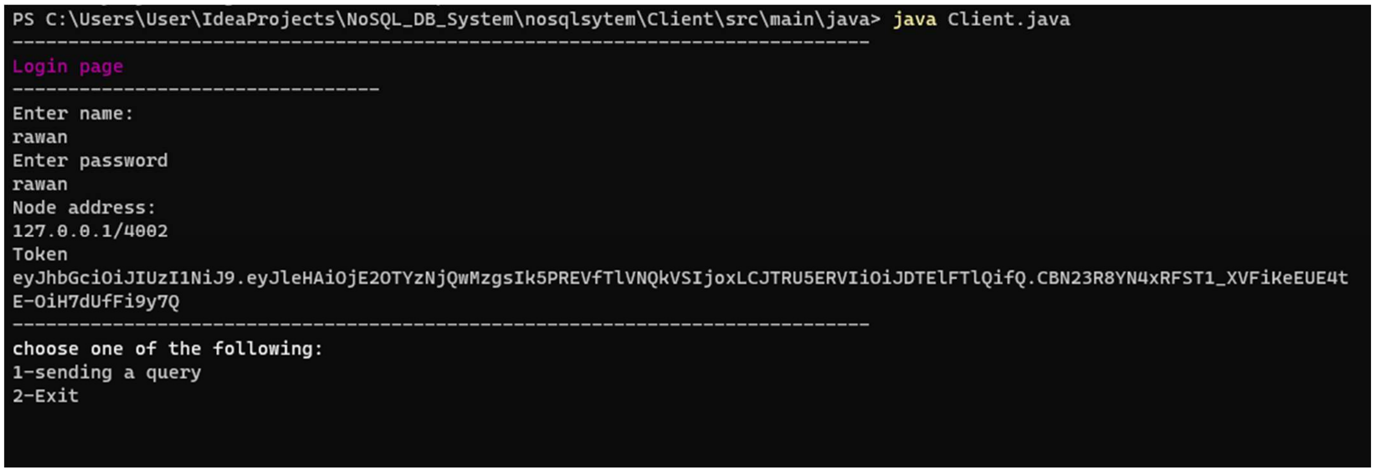
First I have to build and up the Docker-compose:



```
PS C:\Users\User\IdeaProjects\NoSQL_DB_System\nosqlsytem\Docker_Compose> docker-compose up
[+] Running 4/4
 ✓ Network docker_compose_default  Created
 ✓ Container nodeB                 Created
 ✓ Container nodeA                 Created
 ✓ Container bootstrap             Created
Attaching to bootstrap, nodeA, nodeB
nodeA      | Waiting for Bootstrap node...
nodeB      | Waiting for Bootstrap node...
nodeA      | Bootstrap node connected!
nodeA      | Initializing...
nodeB      | Bootstrap node connected!
nodeB      | Initializing...
bootstrap  | The server has started listening on port 4001
bootstrap  | Waiting for clients...
nodeA      | Bootstrap ended!
nodeA      | Waiting for clients...
nodeB      | Bootstrap ended!
nodeB      | Waiting for clients...
```

First the nodes will wait for bootstrap node to connect then initializing and the bootstrap will start listening on bootstrap node port

Then In client side:



```
PS C:\Users\User\IdeaProjects\NoSQL_DB_System\nosqlsytem\Client\src\main\java> java Client.java
-----------------------------------------------------------------------------
Login page
--------------------------------
Enter name:
rawan
Enter password
rawan
Node address:
127.0.0.1/4002
Token
eyJhbGciOiJIUzI1NiJ9.eyJleHAiOjE2OTYzNjQwMzgsIk5PREVfTlVNQkVSIjoxLCJTRU5ERVIiOiJDTElFTlQifQ.CBN23R8YN4xRFST1_XVFiKeEUE4t
E-OiH7dUfFi9y7Q
-----------------------------------------------------------------------------
choose one of the following:
1-sending a query
2-Exit
```

First will handle with login information then after verify the username and password the bootstrap will send the address and port and the token (node information) for the client and now the client can send query to the node