Java and DevOps Bootcamp (May 2023) - Capstone Project

Decentralized Cluster-Based NoSQL DB System.

By: Ibrahem Jaradat.

Table of contents.

1.0 Introduction.

1.1 Overview.

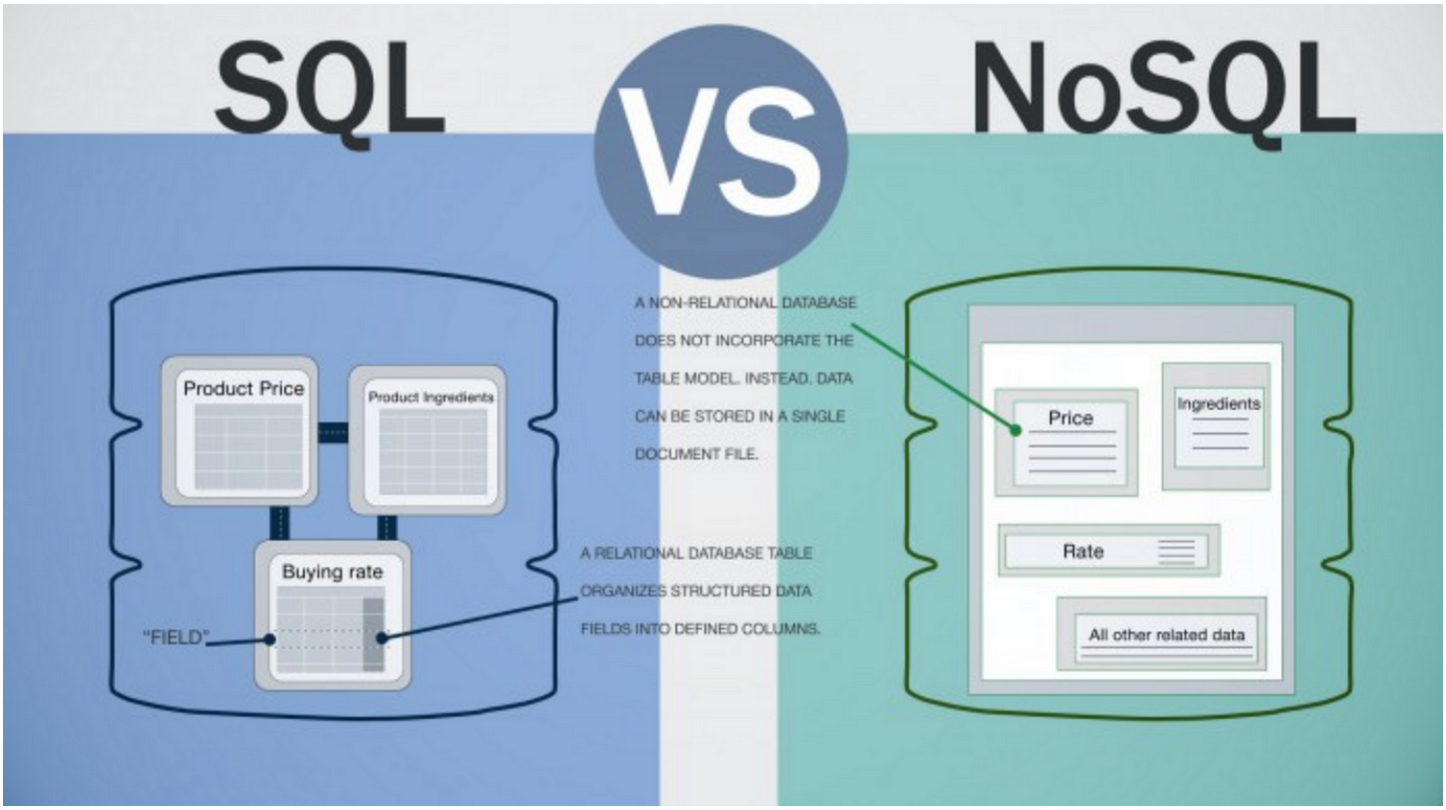
In the ever-evolving landscape of data management, the demand for efficient and scalable database solutions has driven the emergence of

NoSQL databases. These databases depart from the traditional relational model and offer flexible, schema-less data storage, catering to a wide range of modern applications. Within the realm of NoSQL databases, the concept of decentralized cluster-based systems has the ability to distribute data and workload across multiple nodes without relying on a central manager unlike the centralized systems.

This project presented a really exciting challenge to construct a Decentralized Cluster-Based NoSQL DB System using pure Java, inquiring diving down into the complexities of designing a distributed database system that offers high availability, data consistency, and load balancing while operating in a decentralized fashion.

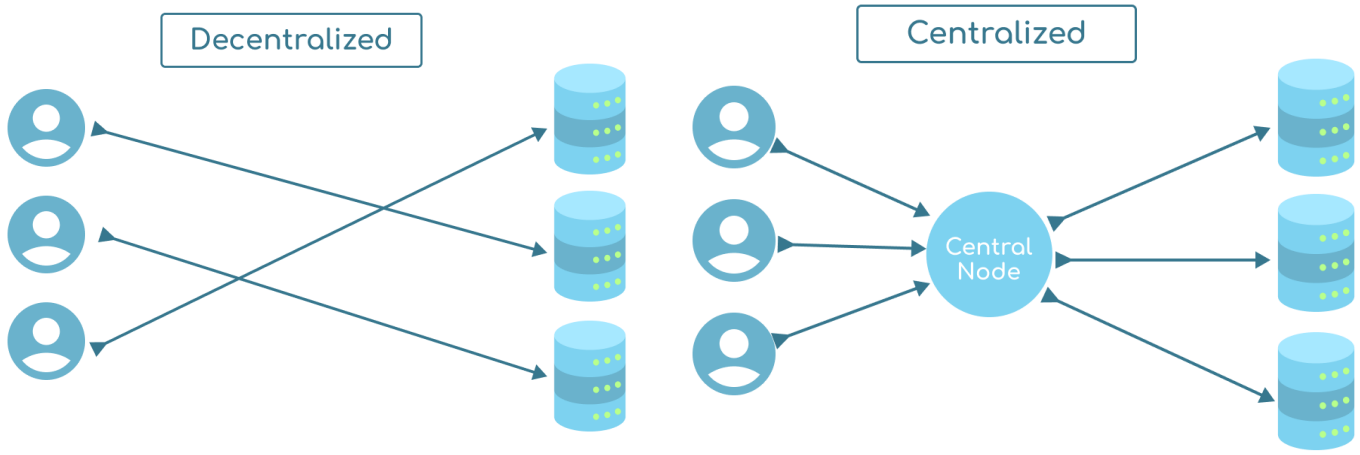
1.2 NoSQL Database in Focus.

A NoSQL database, as the name suggests, departs from traditional relational databases by storing data differently, rather than utilizing structured tables and SQL queries, NoSQL databases offer a variety of data models, including our implementation of a document-based NoSQL database, utilizing JSON objects to store and manage data.



1.3 Decentralized Cluster-Based Approach.

This project requires the implementation of a decentralized cluster-based architecture of the NoSQL database. In contrast to centralized systems, decentralized NoSQL databases distribute data and responsibilities across multiple nodes with no central manager node; instead, sophisticated schemas are employed to ensure data consistency and load balancing among the nodes.



2.0 Application Requirements.

This section outlines the fundamental requirements and specifications of the project. It serves as a foundation for understanding the objectives and functionalities that our application must cover.

2.1 Bootstrapping Step.

The project initiation involves a crucial bootstrapping step, which is responsible for setting up the decentralized cluster and initiating all nodes within it. Key aspects of this step include:

- Bootstrapping Node: A designated node that plays an important role in providing initial configuration information, this information includes user data, Nodes info, and other essentials required for the proper functioning of the cluster.

- User Mapping: The bootstrapping node is responsible for mapping new users to nodes in a load-balanced manner. Users will be

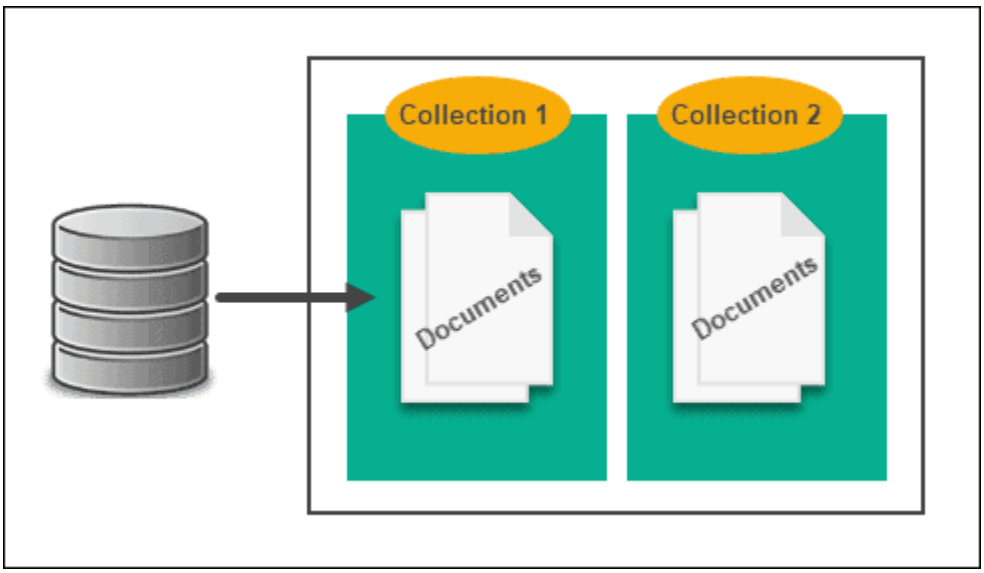
assigned to specific nodes when signing in, ensuring even distribution of workload.

2.2 System Features.

The database within the decentralized NoSQL DB system boasts several features such as:

- Document-Based Structure: The database follows a document-based model, where JSON objects are used to store and organize data.
-

- Collections: Each database has many collections, and every document within a database adheres to a JSON schema associated with that particular collection.



- Supported Queries: Database queries encompass a range of actions, including Reading, Updating Creating or Deleting databases, Collections or Documents.

- Document IDs: Each document is assigned a unique ID, which is efficiently indexed for faster searching and retrieval.

- Index Creation: The system supports the creation of indexes, thus enhancing query performance.

2.3 Data Replication and Consistency.

Data management and replication across nodes play a vital role in the functionality of the DB system:

- Replication: Data, schemas, and indexes are replicated across all nodes with each node storing this information within its local file system.

- Read Queries: Read queries can be satisfied by any node in the cluster, promoting load distribution.

- Write Queries and Node Affinity: Write queries have node affinity indicating that they should be directed to the specific node responsible for the associated data. If a node lacks affinity for a write query, it must forward the query to the node where the affinity exists.

- Data Consistency: After a write operation, the node with affinity must broadcast the data change to other nodes, during this process, if a read query occurs on any node, it may read the older version of the data. However, once a node is updated, subsequent reads are returned with the updated data copy.

2.4 Load Balancing.

Efficient load balancing is essential for the system's performance:

- Users are distributed across nodes in a load-balanced manner, also document affinities as mentioned previously.

- Document-to-Node Affinity: The assignment of documents to nodes is load-balanced, ensuring the distribution of workload.

3.0 Implementation Highlights.

This section delves into the key implementation details of the system. It highlights the strategies, technologies, and methodologies employed to meet the application requirements and address the complexities of a decentralized, Java-based NoSQL database system.

3.1 Load Balancing.

All nodes were represented via docker containers where docker was employed as a robust mechanism of creating and managing these nodes. Key aspects of this implementation include:

Docker Containers: Containers are used to encapsulate each node, ensuring independence and ease of deployment.

Docker Network: A Docker network facilitated communication between containers, emulating node interactions within the cluster efficiently.

3.2 Decentralization.

One of the fundamental principles of this project is decentralization, achieved through various means:

- Avoiding Centralization: Under no circumstance did I introduce centralization in the implementation where users are connected to

nodes directly without introducing any central node, also without missing the concept of the bootstrapping node by passing through it only when signing in, this ensures that the system operates in a truly decentralized fashion.

- Node Independence: Each node is capable of functioning independently, serving user requests without reliance on a central manager or other nodes including the bootstrapping node as mentioned.

3.3 Multithreading and Locks.

To handle multiple users concurrently and address race conditions, multithreading and locking mechanisms were integrated:

- Thread-Based User Service: Within the same node, I implemented multithreading to serve users independently, enhancing system responsiveness and avoiding race conditions.

- Optimistic Locking: To address race conditions when multiple writes target the same JSON property simultaneously, I employed the "optimistic locking" scheme where writes only proceed if the current data version matches the writer's version; otherwise, the write operation is retried after updating the data version.

3.4 JSON Property Indexing.

Efficient indexing of JSON properties is a critical component of database performance.

- Custom Indexing: I developed the indexing mechanism to meet the specific requirements of the system, indexing can be made on document level ensuring optimal query performance.

3.5 Network Communication.

Network communication is a fundamental aspect of a distributed system. Given the assumption of a small-sized DB cluster, I implemented a straightforward network broadcast mechanism to facilitate communication among nodes.

3.6 Demo Application.

A demo application was developed to showcase the capabilities of the database system:

Application Type: The demo web-based application is demonstrating an Email Application.

4.0 Node Explanation.

4.1 Controller.

Controllers are a crucial part of the project's architecture where they are responsible for handling incoming HTTP requests and defining the endpoints for my application. Controllers do routing requests to appropriate services or views , bridging the gap between requests and the underlying business logic, ensuring efficient handling and response delivery.

4.1.1 Query Controller.

This controller is designed to handle POST requests to "/query/processQuery," process the request data using queryService, and return a response entity containing the result.

```java
public class QueryController {

    2 usages
    private final ProcessQueriesService processQueriesService;

    public QueryController(ProcessQueriesService processQueriesService) {
        this.processQueriesService = processQueriesService;
    }

    @PostMapping("/processQuery")
    public ResponseEntity<QueryResponse> processQuery(@RequestBody QueryRequest queryRequest){
        log.info(queryRequest.toString());
        QueryResponse queryResponse = processQueriesService.processQuery(queryRequest);
        return ResponseEntity.status(queryResponse.getStatus()).body(queryResponse);
    }
}
```

4.1.2 Affinity Controller.

This controller is also designed to handle incoming POST requests which are affinity related, by mapping these requests to the "/affinity/processAffinity" endpoint, using AffinityService, it facilitates the execution of affinities and returning a QueryResponse object.

```java
@Slf4j
@RestController()
@RequestMapping("/affinity")
public class AffinityController {
    2 usages
    private final AffinityService affinityService;

    public AffinityController(AffinityService affinityService) { this.affinityService = affinityService; }

    @PostMapping("/processAffinity")
    public ResponseEntity<QueryResponse> processAffinity(@RequestBody QueryRequest queryRequest) {
        log.info("processAffinity " + queryRequest.toString());
        QueryResponse queryResponse = affinityService.processAffinity(queryRequest);
        return ResponseEntity.status(queryResponse.getStatus()).body(queryResponse);
    }
}
```

4.1.3 BroadCast Controller.

This Controller has various endpoints related to broadcasting and reflecting affinity information. By specifying distinct POST endpoints, such as "/broadCast/processQuery," "/broadCast/processForwardedQuery," and "/broadCast/reflectAffinity," this controller effectively handles incoming requests associated with these functionalities.

```java
@Slf4j
@RestController()
@RequestMapping("/broadCast")
public class BroadCastController {
    3 usages
    private final QueryService queryService;
    3 usages
    private final BroadCastService broadCastService;

    public BroadCastController(QueryService queryService, BroadCastService broadCastService) {...}

    @PostMapping("/processQuery")
    public ResponseEntity<QueryResponse> processQuery(@RequestBody QueryRequest queryRequest) {...}

    @PostMapping("/processForwardedQuery")
    public ResponseEntity<QueryResponse> processForwardedQuery(@RequestBody QueryRequest queryRequest) {...}

    @PostMapping("/reflectAffinity")
    public ResponseEntity<QueryResponse> reflectAffinity(@RequestBody QueryRequest queryRequest) {...}
}
```

4.2 Filter.

4.2.1 Affinity Filter.

The Affinity Filter is a component that intercepts incoming HTTP requests in the application so it examines the request content and URI, if the request operation is indeed "broadcastable" and the URI contains "query/processQuery," it forwards the request to a different URI ("affinity/processAffinity"). This filter facilitates the routing of specific requests related to affinity operations while allowing others to continue through the standard filter chain. It plays a crucial role in handling and processing requests based on a predefined criteria within the application.

```java
@Slf4j
@Component
@Order(1)
public class AffinityFilter implements Filter {
    no usages
    @Override
    public void doFilter(ServletRequest servletRequest, ServletResponse servletResponse, FilterChain filterChain) throws IOException, ServletExce
        HttpServletResponse response = (HttpServletResponse) servletResponse;
        RequestWrapper wrapper = new RequestWrapper((HttpServletRequest) servletRequest);
        byte[] body = StreamUtils.copyToByteArray(wrapper.getInputStream());
        QueryRequest queryRequest = JSONUtil.parseObject(body, QueryRequest.class);

        if (OperationType.isBroadCastable(queryRequest.getOperation()) && wrapper.getRequestURI().contains("query/processQuery")) {
            RequestDispatcher dispatcher = wrapper.getRequestDispatcher( path: "/affinity/processAffinity");
            log.info("dispatcher.forward " + queryRequest);
            dispatcher.forward(wrapper, response);
        } else {
            filterChain.doFilter(wrapper, servletResponse);
        }
    }
}
```

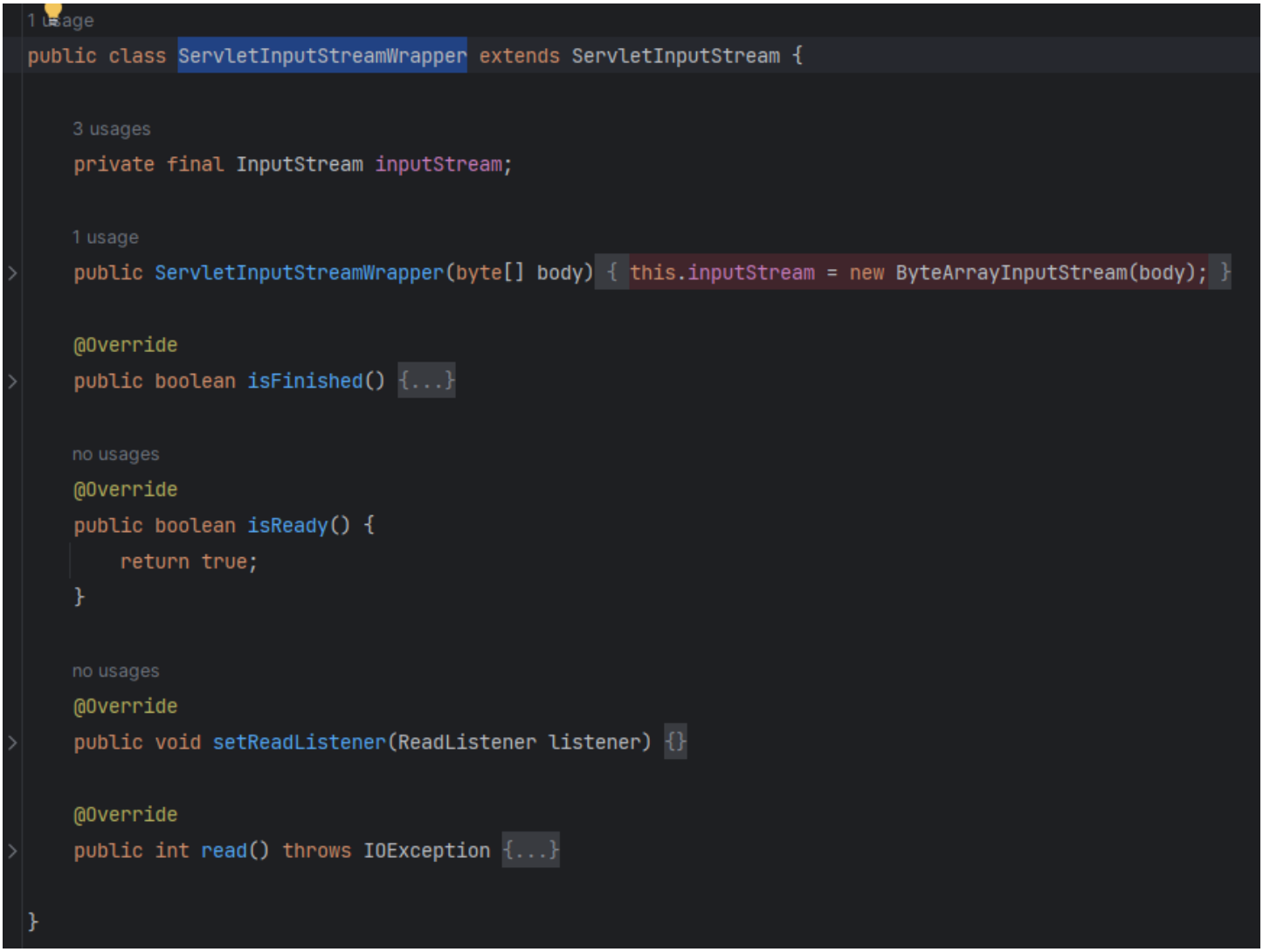
4.2.2 RequestWrapper.

This class extends HttpServletRequestWrapper so it can modify the behavior of the HttpServletRequest by providing a way to read and manipulate the request body. It does this by capturing the request's input stream and storing it in a byte array in the constructor.

```java
public class RequestWrapper extends HttpServletRequestWrapper {

    3 usages
    private final byte[] body;

    1 usage
    public RequestWrapper(HttpServletRequest request) throws IOException {
        super(request);

        this.body = StreamUtils.copyToByteArray(request.getInputStream());
    }

    @Override
    public ServletInputStream getInputStream() {
        return new ServletInputStreamWrapper(this.body);

    }

    @Override
    public BufferedReader getReader() {
        ByteArrayInputStream byteArrayInputStream = new ByteArrayInputStream(this.body);
        return new BufferedReader(new InputStreamReader(byteArrayInputStream));
    }

}
```

4.2.3 ServletInputStreamWrapper.

This class is a custom implementation of the ServletInputStream. Its primary purpose is to wrap the InputStream, specifically a ByteArrayInputStream, around the byte array containing the request body. This step is used mainly because the request body needs to be processed multiple times.



```
1 usage
public class ServletInputStreamWrapper extends ServletInputStream {

    3 usages
    private final InputStream inputStream;

    1 usage
    public ServletInputStreamWrapper(byte[] body) { this.inputStream = new ByteArrayInputStream(body); }

    @Override
    public boolean isFinished() {...}

    no usages
    @Override
    public boolean isReady() {
        return true;
    }

    no usages
    @Override
    public void setReadListener(ReadListener listener) {}

    @Override
    public int read() throws IOException {...}

}
```

4.3 Models.

4.3.1 MetaData.

4.3.1.1 Collection MetaData.

Here it holds metadata about a collection as its name, the number of documents it contains, the last document ID, and a list of indexed properties. The methods for adding and removing indexed properties

provide a way to manage the list of properties that are indexed for efficient querying.

```java
@Data
@Builder
@AllArgsConstructor
@NoArgsConstructor
public class CollectionMetaData {
    String databaseName;
    String collectionName;
    long numOfDocuments;
    long lastId;
    List<String> indexedProperties;

    2 usages
    public void addIndexedProperty(String property){
        if(indexedProperties == null)
            indexedProperties = new ArrayList<>();
        indexedProperties.add(property);
    }
    1 usage
    public void removeIndexedProperty(String property){
        if(indexedProperties == null)
            indexedProperties = new ArrayList<>();
        indexedProperties.remove(property);
    }
}
```

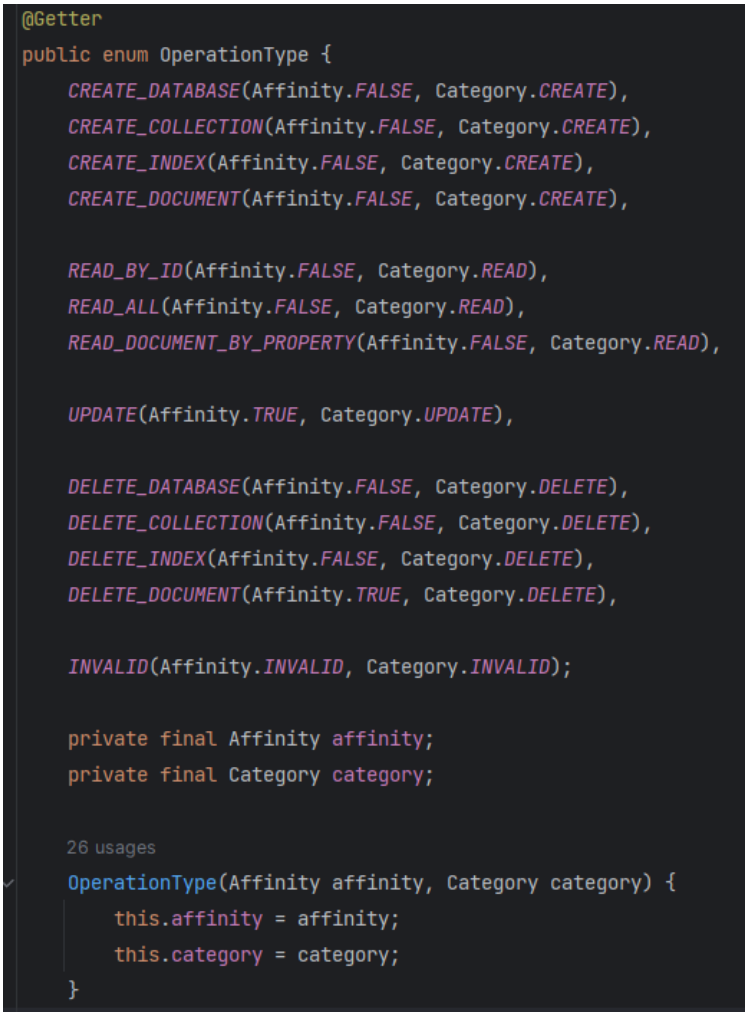
4.3.1.2 Index MetaData.

This class is designed to manage metadata related to indexing. It provides methods for adding and removing items from the list of indexed items, the constructor variations allow for initializing the list with one or more items, depending on the use case.

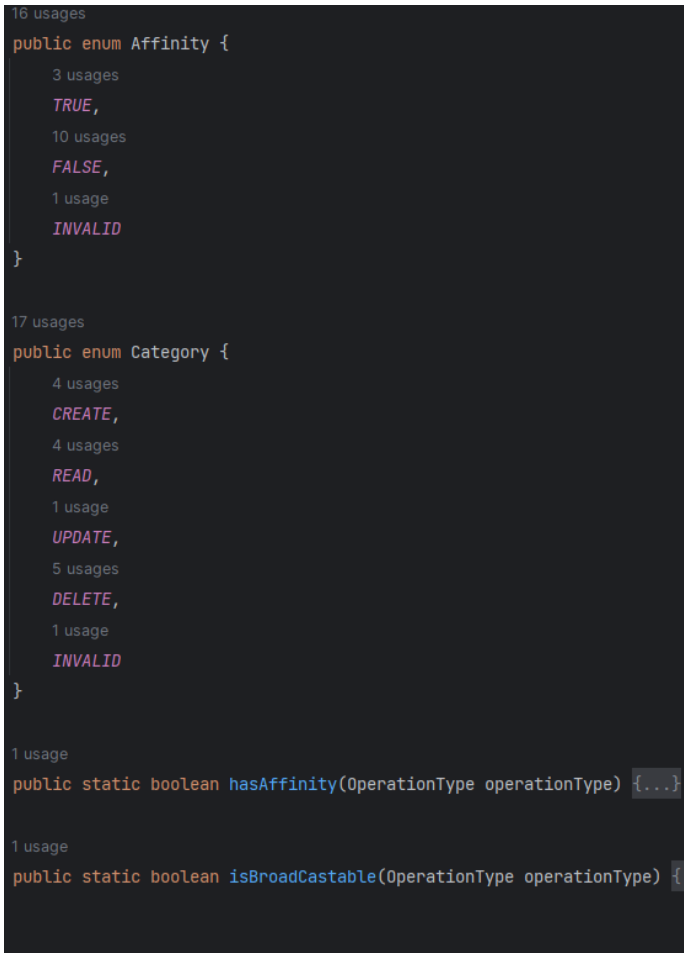
```java
@Data
@AllArgsConstructor
@ToString
public class IndexMetaData {
    List<String> items;

    no usages
    public IndexMetaData() { items = new ArrayList<>(); }

    3 usages
    public IndexMetaData(String path) {
        items = new ArrayList<>();
        items.add(path);
    }

    3 usages
    public void addItem(String item) { items.add(item); }

    2 usages
    public void removeItemById(String id) { items.removeIf(item -> item.contains(id)); }
}
```

4.3.2 Query.

This enum class is designed to provide a structured way of representing and categorizing different types of operations, along with their associated affinities. The hasAffinity and isBroadCastable static methods offer convenient ways to query and filter operations based on their attributes so they can be used later on in the system.



```java
@Getter
public enum OperationType {
    CREATE_DATABASE(Affinity.FALSE, Category.CREATE),
    CREATE_COLLECTION(Affinity.FALSE, Category.CREATE),
    CREATE_INDEX(Affinity.FALSE, Category.CREATE),
    CREATE_DOCUMENT(Affinity.FALSE, Category.CREATE),

    READ_BY_ID(Affinity.FALSE, Category.READ),
    READ_ALL(Affinity.FALSE, Category.READ),
    READ_DOCUMENT_BY_PROPERTY(Affinity.FALSE, Category.READ),

    UPDATE(Affinity.TRUE, Category.UPDATE),

    DELETE_DATABASE(Affinity.FALSE, Category.DELETE),
    DELETE_COLLECTION(Affinity.FALSE, Category.DELETE),
    DELETE_INDEX(Affinity.FALSE, Category.DELETE),
    DELETE_DOCUMENT(Affinity.TRUE, Category.DELETE),

    INVALID(Affinity.INVALID, Category.INVALID);

    private final Affinity affinity;
    private final Category category;

    26 usages
    OperationType(Affinity affinity, Category category) {
        this.affinity = affinity;
        this.category = category;
    }
```



```java
16 usages
public enum Affinity {
    3 usages
    TRUE,
    10 usages
    FALSE,
    1 usage
    INVALID
}

17 usages
public enum Category {
    4 usages
    CREATE,
    4 usages
    READ,
    1 usage
    UPDATE,
    5 usages
    DELETE,
    1 usage
    INVALID
}

1 usage
public static boolean hasAffinity(OperationType operationType) {...}

1 usage
public static boolean isBroadCastable(OperationType operationType) {
```

4.3.3 Request.

This class serves as a structured model for representing query requests within the system, It captures essential information such as the type of operation, the target database and collection, and the request's payload.

```java
@Data
@Builder
@AllArgsConstructor
@NoArgsConstructor
@ToString
public class QueryRequest {
    private OperationType operation;
    private String database;
    private String collection;
    private Map<String, Object> body;
}
```

4.3.4 Response.

Same as the Request class, here we are representing query responses, including attributes for the response data, a message, and the status code. And the jsonObject attribute can hold dynamic key-value pairs representing the response content.

```java
@Data
@Builder
@AllArgsConstructor
@NoArgsConstructor
@ToString
public class QueryResponse {
    private Map<String, Object> jsonObject;
    private String message;
    private int status;
}
```

4.3.5 System.

4.3.5.1 Affinity.

This class represents affinity and all its related information. It includes attributes for identifiers associated with the affinity, and it provides custom equality and hashing logic based on these attributes.

```java
@Data
@Builder
@AllArgsConstructor
@NoArgsConstructor
@ToString
public class Affinity {
    private String _id;
    private String _nodeId;
    private String _documentId;

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        Affinity affinity = (Affinity) o;
        return Objects.equals(_id, affinity._id) && Objects.equals(_nodeId, affinity._nodeId) && Objects.equals(_documentId, affinity._documentId
    }

    @Override
    public int hashCode() { return Objects.hash(_id, _nodeId, _documentId); }
}
```

4.3.5.2 Node.

This class represents nodes in the system and includes attributes for identifiers, names, URLs, and counts of affinities and users associated with each node. The class also provides methods for incrementing and decrementing these counts.

```java
@Data
@Builder
@AllArgsConstructor
@NoArgsConstructor
@ToString
public class Node {
    private String _id;
    private String name;
    private String url;
    private long numOfAffinities;
    private long numOfUsers;
    1 usage
    public void addAffinity() { numOfAffinities++; }

    1 usage
    public void removeAffinity() { numOfAffinities--; }

    no usages
    public void addUsers() { numOfUsers++; }

    no usages
    public void removeUsers() { numOfUsers--; }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        Node node = (Node) o;
        return Objects.equals(_id, node._id);
    }

    @Override
    public int hashCode() { return Objects.hash(_id); }
}
```

4.3.5.3 User.

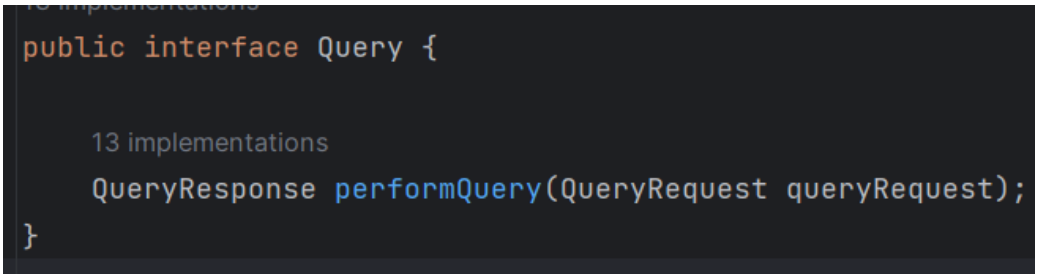
This class represents user information within the system and includes attributes for identifiers, full names, usernames, passwords, and roles

also with the custom equality and hashing logic here where they are based on the _id and username attributes.

```java
@Data
@Builder
@AllArgsConstructor
@NoArgsConstructor
@ToString
public class User {
    private String _id;
    private String full_name;
    private String username;
    private String password;
    private String role;

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        User user = (User) o;
        return Objects.equals(_id, user._id) && Objects.equals(username, user.username);
    }

    @Override
    public int hashCode() { return Objects.hash(_id, username); }
}
```

4.4 Queries.

I implemented the queries in a way that makes it easier to implement and alter any logic in any query, or even add new queries without affecting any existing code, starting with the Query interface that we can implement and write our own logic in the PerformQuery function.

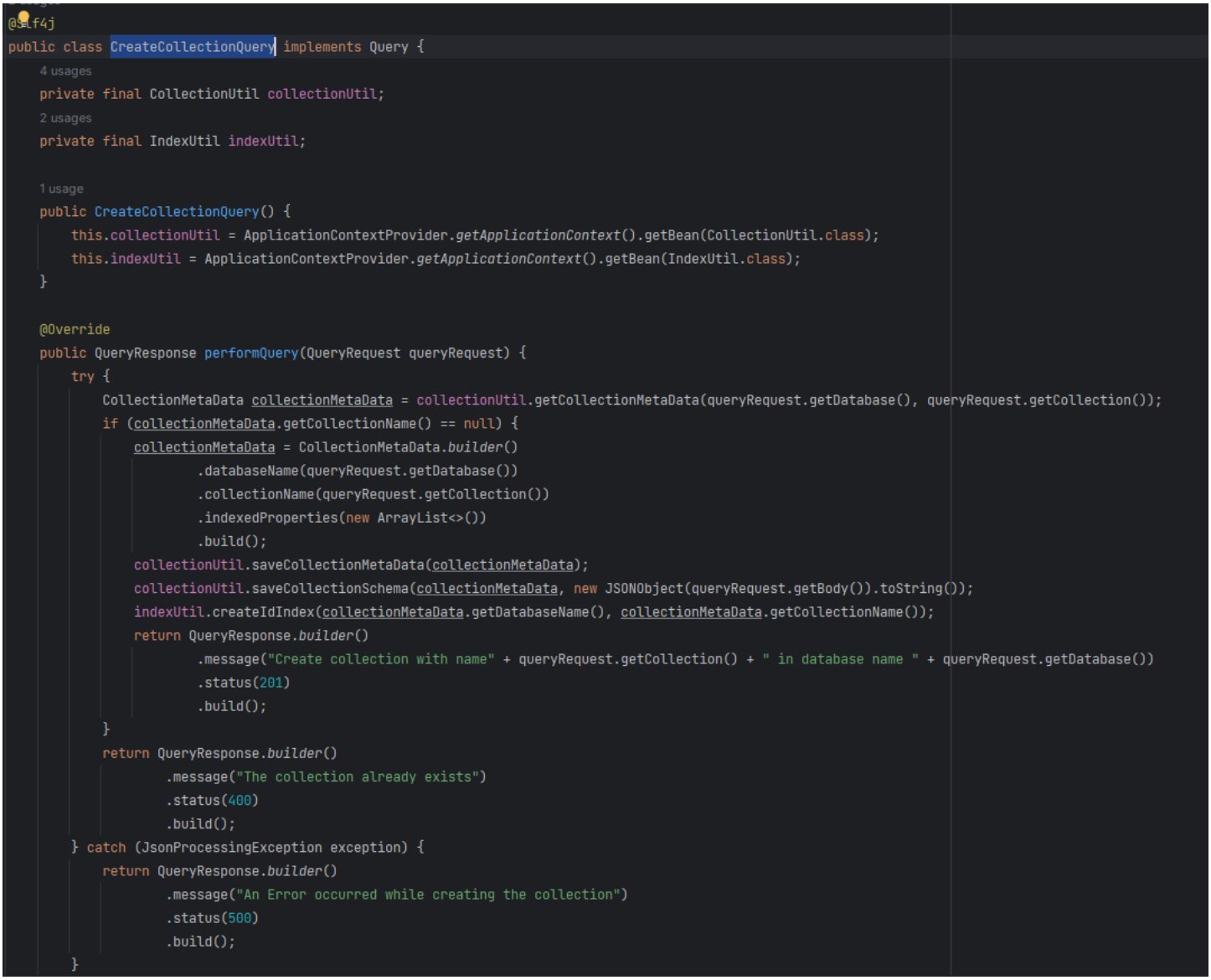


4.4.1 Create Queries.

The Create query can be used to create multiple things, such as (Database, Collection, Document and Index). Query classes also provides responses indicating the status of the operation, including success, already existing , or error conditions.

4.4.1.1 Create Collection Query

This Query class encapsulates the logic for creating a new collection within a database. It relies on utility classes for handling the creation of collection metadata and index management.



```java
@Slf4j
public class CreateCollectionQuery implements Query {
    4 usages
    private final CollectionUtil collectionUtil;
    2 usages
    private final IndexUtil indexUtil;

    1 usage
    public CreateCollectionQuery() {
        this.collectionUtil = ApplicationContextProvider.getApplicationContext().getBean(CollectionUtil.class);
        this.indexUtil = ApplicationContextProvider.getApplicationContext().getBean(IndexUtil.class);
    }

    @Override
    public QueryResponse performQuery(QueryRequest queryRequest) {
        try {
            CollectionMetaData collectionMetaData = collectionUtil.getCollectionMetaData(queryRequest.getDatabase(), queryRequest.getCollection());
            if (collectionMetaData.getCollectionName() == null) {
                collectionMetaData = CollectionMetaData.builder()
                        .databaseName(queryRequest.getDatabase())
                        .collectionName(queryRequest.getCollection())
                        .indexedProperties(new ArrayList<>())
                        .build();
                collectionUtil.saveCollectionMetaData(collectionMetaData);
                collectionUtil.saveCollectionSchema(collectionMetaData, new JSONObject(queryRequest.getBody()).toString());
                indexUtil.createIdIndex(collectionMetaData.getDatabaseName(), collectionMetaData.getCollectionName());
                return QueryResponse.builder()
                        .message("Create collection with name" + queryRequest.getCollection() + " in database name " + queryRequest.getDatabase())
                        .status(201)
                        .build();
            }
            return QueryResponse.builder()
                    .message("The collection already exists")
                    .status(400)
                    .build();
        } catch (JsonProcessingException exception) {
            return QueryResponse.builder()
                    .message("An Error occurred while creating the collection")
                    .status(500)
                    .build();
        }
```

4.4.1.2 Create Database Query.

This Query class encapsulates the logic for creating a new database. It relies on the FileStorageUtil class for handling the actual creation of the database.

```java
@Slf4j
public class CreateDatabaseQuery implements Query {

    2 usages
    private final FileStorageUtil fileStorageUtil;

    1 usage
    public CreateDatabaseQuery() {
        this.fileStorageUtil = ApplicationContextProvider.getApplicationContext().getBean(FileStorageUtil.class);

    }

    @Override
    public QueryResponse performQuery(QueryRequest queryRequest) {
        try {
            boolean created = fileStorageUtil.createDataBase(queryRequest.getDatabase());
            if (created) {
                return QueryResponse.builder()
                        .message("Create database with name " + queryRequest.getDatabase())
                        .status(201)
                        .build();
            }
            return QueryResponse.builder()
                    .message("The database already exists")
                    .status(400)
                    .build();
        } catch (RuntimeException exception) {
            return QueryResponse.builder()
                    .message("Internal Error occurred")
                    .status(500)
                    .build();
        }
    }
}
```

4.4.1.3 Create Document Query.

This Query class encapsulates the logic for creating a new document within a collection. It relies on utility classes for handling various aspects such as document validation, storage, and indexing.

```java
public CreateDocumentQuery() {
    this.collectionUtil = ApplicationContextProvider.getApplicationContext().getBean(CollectionUtil.class);
    this.documentUtil = ApplicationContextProvider.getApplicationContext().getBean(DocumentUtil.class);
    this.indexUtil = ApplicationContextProvider.getApplicationContext().getBean(IndexUtil.class);
}

@Override
public QueryResponse performQuery(QueryRequest queryRequest) {
    try {
        CollectionMetaData collectionMetaData = collectionUtil.getCollectionMetaData(queryRequest.getDatabase(), queryRequest.getCollection());

        JSONObject body = new JSONObject(queryRequest.getBody());
        long documentId = collectionMetaData.getLastId();
        body.put("_id", collectionMetaData.getCollectionName() + documentId);
        log.info(body.toString());
        collectionUtil.validateDocumentToSchema(collectionMetaData, body);
        Path documentPath = documentUtil.saveDocument(collectionMetaData, body.toString());
        collectionUtil.addCollectionMetaDataDocument(collectionMetaData);
        indexUtil.indexNewDocument(collectionMetaData.getDatabaseName(), collectionMetaData.getCollectionName(), body, documentPath);
        return QueryResponse.builder()
                .message("Create a document in collection name " + queryRequest.getCollection() + "in database name " + queryRequest.getDatabase())
                .jsonObject(body.toMap())
                .status(201)
                .build();
    } catch (JsonProcessingException exception) {
        return QueryResponse.builder()
                .message("The document cannot be parsed")
                .status(500)
                .build();
    } catch (ValidationException exception) {
        return QueryResponse.builder()
                .message(exception.getMessage())
                .status(400)
                .build();
    } catch (FileNotFoundException | NullPointerException e) {
        return QueryResponse.builder()
                .message("The collection doesn't have a schema")
                .status(404)
                .build();
    }
}
```

4.4.1.4 Create Index Query..

Here the encapsulates the logic for creating an index on a document. It relies on the IndexUtil class for handling the actual creation of the index.

```java
@Slf4j
public class CreateIndexQuery implements Query {

    2 usages
    private final IndexUtil indexUtil;

    1 usage
    public CreateIndexQuery() {
        this.indexUtil = ApplicationContextProvider.getApplicationContext().getBean(IndexUtil.class);
    }

    @Override
    public QueryResponse performQuery(QueryRequest queryRequest) {
        if (indexUtil.createPropertyIndex(queryRequest.getDatabase(), queryRequest.getCollection(), queryRequest.getBody().get("indexedBy").toString())) {
            return QueryResponse.builder()
                    .message("Create an index on " + queryRequest.getBody().get("indexedBy").toString()
                            + " collection name " + queryRequest.getCollection() + "in database name " + queryRequest.getDatabase())
                    .status(201)
                    .build();
        }
        return QueryResponse.builder()
                .message("error while indexing")
                .status(500)
                .build();
    }
}
```

4.4.2 Delete Queries.

The Delete query can be used to delete multiple things, such as (Database, Collection, Document and Index). Query classes also provide responses indicating the status of the operation, including success or error conditions.
The DeleteQuery classes encapsulates the logic for deleting. It relies on the FileStorageUtil class for handling the actual deletion of the database, It also provides responses indicating the status of the operation, including success or error conditions.

4.4.2.1 Delete Collection Query.

```java
@Slf4j
public class DeleteCollectionQuery implements Query {

    2 usages
    private final FileStorageUtil fileStorageUtil;

    1 usage
    public DeleteCollectionQuery() {
        this.fileStorageUtil = ApplicationContextProvider.getApplicationContext().getBean(FileStorageUtil.class);
    }


    @Override
    public QueryResponse performQuery(QueryRequest queryRequest) {
        try {
            fileStorageUtil.deleteCollection(queryRequest.getDatabase(), queryRequest.getCollection());
            return QueryResponse.builder()
                    .message("Collection deleted Successfully")
                    .status(200)
                    .build();
        } catch (RuntimeException exception) {
            return QueryResponse.builder()
                    .message("Internal Error occurred")
                    .status(500)
                    .build();
        }
    }
}
```

4.4.2.2 Delete Database Query.

```java
public class DeleteDatabaseQuery implements Query {
    2 usages
    private final FileStorageUtil fileStorageUtil;

    1 usage
    public DeleteDatabaseQuery() {
        this.fileStorageUtil = ApplicationContextProvider.getApplicationContext().getBean(FileStorageUtil.class);
    }


    @Override
    public QueryResponse performQuery(QueryRequest queryRequest) {
        try {
            fileStorageUtil.deleteDatabase(queryRequest.getDatabase());
            return QueryResponse.builder()
                    .message("Database deleted Successfully")
                    .status(200)
                    .build();
        } catch (RuntimeException exception) {
            return QueryResponse.builder()
                    .message("Internal Error occurred")
                    .status(500)
                    .build();
        }
    }
}
```

4.4.2.3 Delete Document Query.

```java
@Override
public QueryResponse performQuery(QueryRequest queryRequest) {
    try {
        CollectionMetaData collectionMetaData = collectionUtil.getCollectionMetaData(queryRequest.getDatabase(), queryRequest.getCollection());
        JSONObject document = documentUtil.getDocumentById(queryRequest.getDatabase(), queryRequest.getCollection(), queryRequest.getBody().get("id").toString());
        documentUtil.deleteDocumentById(collectionMetaData, queryRequest.getBody().get("id").toString());
        indexUtil.deleteIndexedDocument(collectionMetaData, document);
        collectionUtil.deleteCollectionMetaDataDocument(collectionMetaData);
        return QueryResponse.builder()
                .message("Document deleted Successfully")
                .status(200)
                .build();
    } catch (JsonProcessingException exception) {
        log.error(exception.getMessage());
        return QueryResponse.builder()
                .message("Document not deleted Successfully")
                .status(500)
                .build();
    } catch (NoSuchElementException exception) {
        return QueryResponse.builder()
                .message("Document not found")
                .status(404)
                .build();
    }
}
```

4.4.2.4 Delete Index Query.

```java
@Override
public QueryResponse performQuery(QueryRequest queryRequest) {
    try {
        indexUtil.deleteIndex(queryRequest.getDatabase(), queryRequest.getCollection(), queryRequest.getBody().get("index").toString());
        return QueryResponse.builder()
                .message("Index deleted Successfully")
                .status(200)
                .build();
    } catch (FileNotFoundException e) {
        return QueryResponse.builder()
                .message("Index not found")
                .status(404)
                .build();
    }
}
```

4.4.3 Read Queries.

Read Query classes encapsulates the logic for reading all documents within a collection. It relies on the DocumentUtil class for retrieving the document or multiple documents by Id or by a specified property and combining them into a single JSON object. It also provides responses

indicating the status of the operation, including success or document not found.

4.4.3.1 Read All Documents Query.

```
@Override
public QueryResponse performQuery(QueryRequest queryRequest) {
    try {
        List<JSONObject> documents = documentUtil.getAllDocuments(queryRequest.getDatabase(), queryRequest.getCollection());

        JSONObject combined = new JSONObject();
        for (int i = 0; i < documents.size(); i++) {
            combined.put("document" + i, documents.get(i));
        }
        return QueryResponse.builder()
                .message("Here is your document chief")
                .jsonObject(combined.toMap())
                .status(200)
                .build();
    } catch (NoSuchElementException exception) {
        return QueryResponse.builder()
                .message("You missed up chief, the document not found")
                .status(404)
                .build();
    }
}
```

4.4.3.2 Read Document By Id Query.

```
@Override
public QueryResponse performQuery(QueryRequest queryRequest) {
    try {
        JSONObject document = documentUtil.getDocumentById(queryRequest.getDatabase(), queryRequest.getCollection(), queryRequest.getBody().get("id").toStri
        return QueryResponse.builder()
                .message("Here is your document chief")
                .jsonObject(document.toMap())
                .status(200)
                .build();
    } catch (NoSuchElementException exception) {
        return QueryResponse.builder()
                .message("You missed up chief, the document not found")
                .status(404)
                .build();
    }
}
```

4.4.3.3 Read Document By Property Query.

```java
@Override
public QueryResponse performQuery(QueryRequest queryRequest) {
    try {
        CollectionMetaData collectionMetaData = collectionUtil.getCollectionMetaData(queryRequest.getDatabase(), queryRequest.getCollection());
        List<JSONObject> documents;
        if (collectionMetaData.getIndexedProperties().contains(queryRequest.getBody().get("property").toString())) {
            documents = indexUtil.getDocumentByProperty(collectionMetaData, queryRequest.getBody().get("property").toString(), queryRequest.getBody().get("value").toString());
        } else {
            documents = documentUtil.getDocumentByProperty(collectionMetaData, queryRequest.getBody().get("property").toString(), queryRequest.getBody().get("value").toString()
        }

        JSONObject combined = new JSONObject();
        for (int i = 0; i < documents.size(); i++) {
            combined.put("document" + i, documents.get(i));
        }

        return QueryResponse.builder()
                .message("Here is your document chief")
                .jsonObject(combined.toMap())
                .status(200)
                .build();
    } catch (NoSuchElementException exception) {
        return QueryResponse.builder()
                .message("You missed up chief, the document not found")
                .status(404)
                .build();
    } catch (JsonProcessingException e) {
        throw new RuntimeException(e);
    }
}
```

4.4.4 Update Document Query.

The UpdateQuery class encapsulates the logic for updating a document within a collection, while making sure to use synchronization and optimistic locking . It relies on utility classes for handling various aspects such as document retrieval, updating, and index maintenance. It provides responses indicating the status of the update operation, including success, data conflict, or error conditions.

```java
@Override
public QueryResponse performQuery(QueryRequest queryRequest) {
    try {
        CollectionMetaData collectionMetaData = collectionUtil.getCollectionMetaData(queryRequest.getDatabase(), queryRequest.getCollection());
        Path path = documentUtil.updateDocument(collectionMetaData, queryRequest.getBody());
        indexUtil.updateDocument(collectionMetaData, queryRequest.getBody(), path);
        return QueryResponse.builder()
                .message("updated a document in collection name " + queryRequest.getCollection() + "in database name " + queryRequest.getDatabase())
                .status(200)
                .jsonObject(documentUtil.getDocumentByAbsolutePath(path.toString()).toMap())
                .build();
    } catch (JsonProcessingException e) {
        return QueryResponse.builder()
                .message("Your document is not good man")
                .status(500)
                .build();
    } catch (OptimisticLockingFailureException exception) {
        return QueryResponse.builder()
                .message("Failed to Update the document due to data conflict")
                .status(400)
                .jsonObject(queryRequest.getBody())
                .build();
    }
}
```

4.4.5 Invalid Query.

The InvalidQuery class serves as a fallback or catch-all query handler for unsupported or invalid queries. It provides a response indicating that the query is not supported.

```java
@Slf4j
public class InvalidQuery implements Query {


    1 usage
    public InvalidQuery() {
    }

    @Override
    public QueryResponse performQuery(QueryRequest queryRequest) {
        Map<Integer, Object> map = IntStream.range(0, OperationType.values().length)
                .boxed()
                .collect(Collectors.toMap(i -> i, i -> OperationType.values()[i]));

        return QueryResponse.builder()
                .message("Not Supported Query")
                .status(400)
                .jsonObject(new JSONObject(map).toMap())
                .build();
    }
}
```

4.4.6 Query Factory.

This class is responsible for creating specific query objects based on the OperationType specified in a QueryRequest; it returns an instance of the corresponding query class. The supported query classes include those for creating, reading, updating, and deleting databases, collections, documents, and indexes.
If the OperationType is not recognized or not supported, it returns an instance of the InvalidQuery class,
This design pattern allows for the dynamic selection of query handlers based on the requested operation.

```java
public class QueryFactory {

    1 usage
    public Query makeQuery(QueryRequest queryRequest) {
        switch (queryRequest.getOperation()) {
            case CREATE_DATABASE:
                return new CreateDatabaseQuery();
            case CREATE_COLLECTION:
                return new CreateCollectionQuery();
            case CREATE_INDEX:
                return new CreateIndexQuery();
            case CREATE_DOCUMENT:
                return new CreateDocumentQuery();
            case READ_BY_ID:
                return new ReadByIdQuery();
            case READ_ALL:
                return new ReadAllQuery();
            case READ_DOCUMENT_BY_PROPERTY:
                return new ReadByPropertyQuery();
            case UPDATE:
                return new UpdateQuery();
            case DELETE_DATABASE:
                return new DeleteDatabaseQuery();
            case DELETE_COLLECTION:
                return new DeleteCollectionQuery();
            case DELETE_INDEX:
                return new DeleteIndexQuery();
            case DELETE_DOCUMENT:
                return new DeleteDocumentQuery();
            default:
                return new InvalidQuery();
        }
    }

}
```

4.5 Service.

The service classes serve as intermediaries between the controllers and the data access layer, their primary job of service classes is to encapsulate the business logic and application-specific functionality.

4.5.1 Query Service.

This class is a Spring service responsible for processing query requests by creating the appropriate query objects using the QueryFactory and then executing those queries.

```java
@Service
public class QueryService {

    2 usages
    private final QueryFactory queryFactory;

    public QueryService() { this.queryFactory = new QueryFactory(); }

    4 usages
    public QueryResponse processQuery(QueryRequest queryRequest) {
        Query query = queryFactory.makeQuery(queryRequest);
        return query.performQuery(queryRequest);
    }

}
```

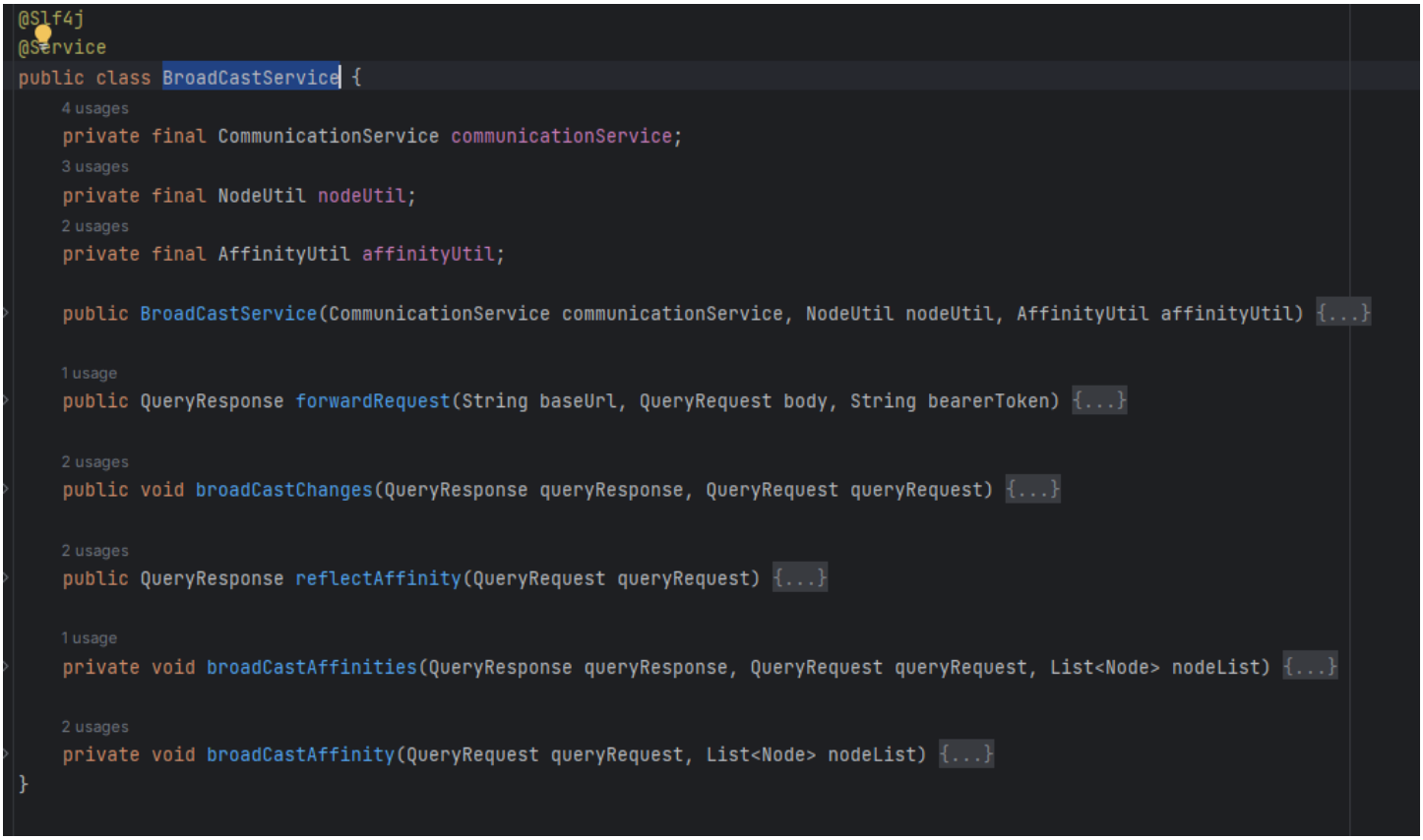
4.5.2 Affinity Service.

This class is a Spring service responsible for handling database operations that involve affinity management. It ensures that database operations are routed to the appropriate nodes with affinity, and it also handles broadcasting changes as needed.

```java
@Service
public class AffinityService {
    4 usages
    private final NodeUtil nodeUtil;
    2 usages
    private final AffinityUtil affinityUtil;
    2 usages
    private final QueryService queryService;

    3 usages
    private final BroadCastService broadCastService;

    public AffinityService(NodeUtil nodeUtil, AffinityUtil affinityUtil, QueryService queryService, BroadCastService broadCastService) {...}

    1 usage
    public QueryResponse processAffinity(QueryRequest queryRequest) {
        Node affinityNode;
        QueryResponse queryResponse;
        if (OperationType.hasAffinity(queryRequest.getOperation())) {
            log.info("has affinity ");
            String documentId = queryRequest.getBody().get("id").toString();
            String nodeId = affinityUtil.findAffinityNode(documentId);
            affinityNode = nodeUtil.getNodeByNodeId(nodeId);
        } else {
            affinityNode = nodeUtil.findBalancedNode();
        }
        log.info("affinityNode -> " + affinityNode);
        if (nodeUtil.isInLocalNode(affinityNode)) {
            log.info("is local node ");
            queryResponse = queryService.processQuery(queryRequest);
            broadCastService.broadCastChanges(queryResponse, queryRequest);
        } else {
            log.info("forwarding to node -> " + affinityNode);
            queryResponse = broadCastService.forwardRequest(affinityNode.getUrl(), queryRequest, affinityNode.get_id());
        }

        return queryResponse;
    }
}
```

4.5.3 BroadCast Service.

This class is responsible for broadcasting changes and reflecting affinity within the system. It plays a crucial role in broadcasting changes and ensuring that updates and affinities are synchronized across the distributed system. It abstracts the complexity of communication and affinity management between nodes in a distributed environment.



```java
@Slf4j
@Service
public class BroadCastService {
    4 usages
    private final CommunicationService communicationService;
    3 usages
    private final NodeUtil nodeUtil;
    2 usages
    private final AffinityUtil affinityUtil;

    public BroadCastService(CommunicationService communicationService, NodeUtil nodeUtil, AffinityUtil affinityUtil) {...}

    1 usage
    public QueryResponse forwardRequest(String baseUrl, QueryRequest body, String bearerToken) {...}

    2 usages
    public void broadCastChanges(QueryResponse queryResponse, QueryRequest queryRequest) {...}

    2 usages
    public QueryResponse reflectAffinity(QueryRequest queryRequest) {...}

    1 usage
    private void broadCastAffinities(QueryResponse queryResponse, QueryRequest queryRequest, List<Node> nodeList) {...}

    2 usages
    private void broadCastAffinity(QueryRequest queryRequest, List<Node> nodeList) {...}
}
```

4.5.4 Communication Service.

This Service class is responsible for sending POST requests to multiple URLs and handling the responses.

- sendPostRequestsToMultipleUrls Method: This method sends POST requests to multiple URLs in parallel. It takes a list of Node objects (representing remote nodes), a QueryRequest body, and a path as inputs.
- Using a Flux.fromIterable operation to iterate over the list of Node objects in parallel.
- Concurrency: Both the sendPostRequestsToMultipleUrls and sendPostRequest methods use Reactor Core's operators to perform operations concurrently. This allows for parallel execution of POST requests to multiple URLs.

Overall, the CommunicationService class abstracts the communication with nodes, making it easier to send POST requests in parallel and handle the responses asynchronously using reactive programming techniques.

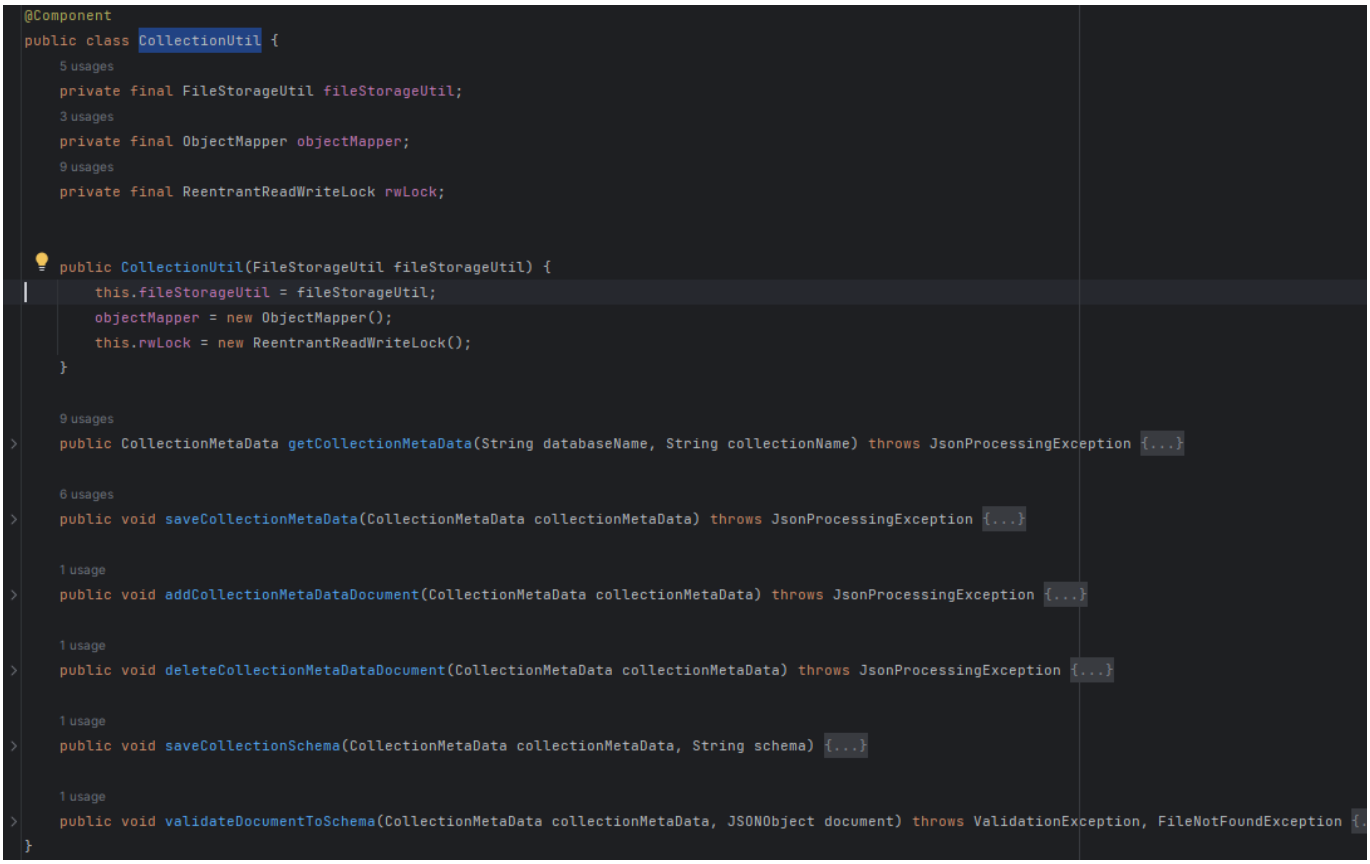
```java
@Slf4j
@Service
public class CommunicationService {
    2 usages
    private final WebClient webClient;

    public CommunicationService() { this.webClient = WebClient.builder().build(); }

    2 usages
    public Flux<QueryResponse> sendPostRequestsToMultipleUrls(List<Node> nodeList, QueryRequest body, String path) {
        return Flux.fromIterable(nodeList) Flux<Node>
                .flatMap(node -> sendPostRequest(node.getUrl(), body, node.get_id(), path)) Flux<QueryResponse>
                .subscribeOn(Schedulers.boundedElastic());
    }


    2 usages
    public Mono<QueryResponse> sendPostRequest(String baseUrl, QueryRequest body, String bearerToken, String path) {
        log.info("sendPostRequest");
        return webClient.post() RequestBodyUriSpec
                .uri(baseUrl, uriBuilder -> uriBuilder.
                        path("broadCast/")
                        .path(path)
                        .build()) RequestBodySpec
                .header( headerName: "Authorization", ...headerValues: "Bearer " + bearerToken)
                .bodyValue(body) RequestHeadersSpec<capture of ?>
                .retrieve() ResponseSpec
                .bodyToMono(QueryResponse.class);
    }
}
```

4.6 Utility.

Utility classes are used where I can Implement any operations to use in the Query classes to keep the code dry and avoid repeating myself ensuring data integrity and consistency.

4.6.1 Collection Utility.

This class provides essential methods for managing collection metadata and schema, it abstracts the interaction with the underlying file storage and schema validation, making it easier to work with collections and documents.



```java
@Component
public class CollectionUtil {
    5 usages
    private final FileStorageUtil fileStorageUtil;
    3 usages
    private final ObjectMapper objectMapper;
    9 usages
    private final ReentrantReadWriteLock rwLock;


    public CollectionUtil(FileStorageUtil fileStorageUtil) {
        this.fileStorageUtil = fileStorageUtil;
        objectMapper = new ObjectMapper();
        this.rwLock = new ReentrantReadWriteLock();
    }

    9 usages
    public CollectionMetaData getCollectionMetaData(String databaseName, String collectionName) throws JsonProcessingException {...}

    6 usages
    public void saveCollectionMetaData(CollectionMetaData collectionMetaData) throws JsonProcessingException {...}

    1 usage
    public void addCollectionMetaDataDocument(CollectionMetaData collectionMetaData) throws JsonProcessingException {...}

    1 usage
    public void deleteCollectionMetaDataDocument(CollectionMetaData collectionMetaData) throws JsonProcessingException {...}

    1 usage
    public void saveCollectionSchema(CollectionMetaData collectionMetaData, String schema) {...}

    1 usage
    public void validateDocumentToSchema(CollectionMetaData collectionMetaData, JSONObject document) throws ValidationException, FileNotFoundException {
}
```

4.6.2 Document Utility.

This Utility is responsible for managing documents in collections, including operations like saving, updating, deleting, and retrieving documents providing essential utility methods including saving, updating, deleting, and retrieving documents. It also handles concurrency control and optimistic locking.

```java
public class DocumentUtil {
    8 usages
    private final FileStorageUtil fileStorageUtil;
    17 usages
    private final ReentrantReadWriteLock rwLock;


    public DocumentUtil(FileStorageUtil fileStorageUtil) {
        this.fileStorageUtil = fileStorageUtil;
        this.rwLock = new ReentrantReadWriteLock();
    }

    1 usage
    public Path saveDocument(CollectionMetaData collectionMetaData, String document) {...}

    1 usage
    public Path updateDocument(CollectionMetaData collectionMetaData, JSONObject document) {...}

    1 usage
    public void deleteDocumentById(CollectionMetaData collectionMetaData, String id) {...}

    3 usages
    public JSONObject getDocumentById(String databaseName, String collectionName, String id) throws NoSuchElementException {...}

    3 usages
    public JSONObject getDocumentByAbsolutePath(String absolutePath) {...}

    1 usage
    public List<JSONObject> getDocumentByProperty(CollectionMetaData collectionMetaData, String property, String value) {...}

    1 usage
    public List<JSONObject> getAllDocuments(String databaseName, String collectionName) {...}

    1 usage
    public Path updateDocument(CollectionMetaData collectionMetaData, Map<String, Object> body) throws OptimisticLockingFailureException {
}
```

4.6.3 File Storage Utility.

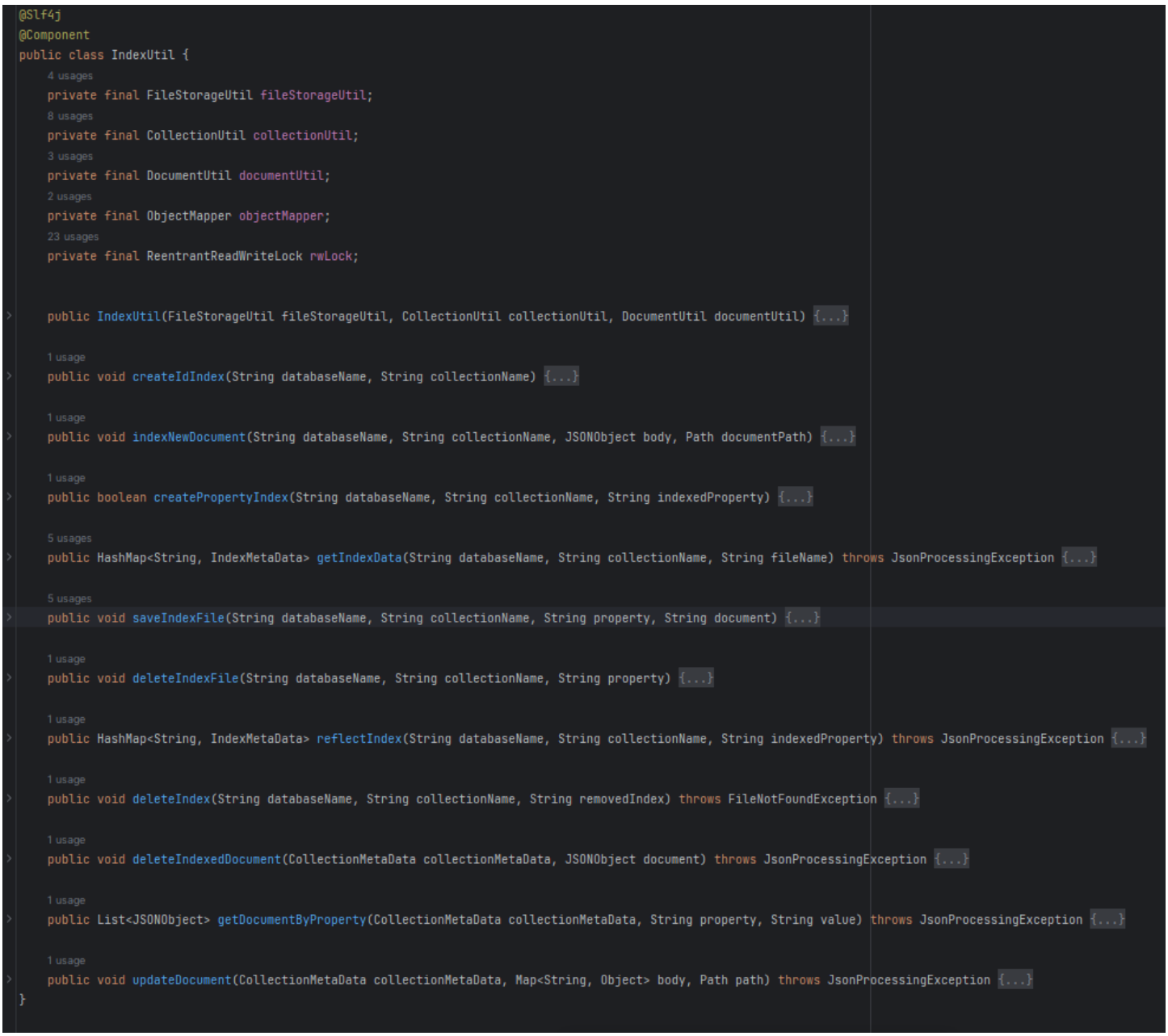
Is responsible for managing file storage and retrieval. It is used for storing JSON documents and other related data for different databases and collections within those databases. It provides essential utility methods for managing file storage for JSON documents, databases, and collections.

```java
@Component
public class FileStorageUtil {

    7 usages
    private final String ROOT_DIR = "storage";
    5 usages
    private final String FILE_EXTENSION = ".json";

    1 usage
    public boolean createDataBase(String databaseName) {...}

    1 usage
    public Path createCollection(String databaseName, String collectionName) {...}

    5 usages
    public Path storeFile(String databaseName, String collectionName, String fileName, String content) {...}

    5 usages
    public Optional<JSONObject> getFile(String databaseName, String collectionName, String fileName) {...}

    1 usage
    public Optional<JSONObject> getFileByAbsolutePath(String absoluteFilePath) {...}

    2 usages
    private Optional<JSONObject> getJsonObject(Path filePath) {...}
    1 usage
    public void deleteDatabase(String databaseName) {...}

    1 usage
    public void deleteCollection(String databaseName, String collectionName) {...}

    2 usages
    public void deleteFile(String databaseName, String collectionName, String fileName) {...}

    2 usages
    public List<JSONObject> getAllDocumentsInCollection(String databaseName, String collectionName) {...}

}
```

4.6.4 Index Utility.

Is responsible for managing indexing of documents within collections. It provides various methods for creating, updating, and deleting indexes, as well as performing operations related to indexed properties to ensure

that documents are appropriately indexed and that the indexes are updated or removed when necessary.



```java
@Slf4j
@Component
public class IndexUtil {
    4 usages
    private final FileStorageUtil fileStorageUtil;
    8 usages
    private final CollectionUtil collectionUtil;
    3 usages
    private final DocumentUtil documentUtil;
    2 usages
    private final ObjectMapper objectMapper;
    23 usages
    private final ReentrantReadWriteLock rwLock;


    public IndexUtil(FileStorageUtil fileStorageUtil, CollectionUtil collectionUtil, DocumentUtil documentUtil) {...}

    1 usage
    public void createIdIndex(String databaseName, String collectionName) {...}

    1 usage
    public void indexNewDocument(String databaseName, String collectionName, JSONObject body, Path documentPath) {...}

    1 usage
    public boolean createPropertyIndex(String databaseName, String collectionName, String indexedProperty) {...}

    5 usages
    public HashMap<String, IndexMetaData> getIndexData(String databaseName, String collectionName, String fileName) throws JsonProcessingException {...}

    5 usages
    public void saveIndexFile(String databaseName, String collectionName, String property, String document) {...}

    1 usage
    public void deleteIndexFile(String databaseName, String collectionName, String property) {...}

    1 usage
    public HashMap<String, IndexMetaData> reflectIndex(String databaseName, String collectionName, String indexedProperty) throws JsonProcessingException {...}

    1 usage
    public void deleteIndex(String databaseName, String collectionName, String removedIndex) throws FileNotFoundException {...}

    1 usage
    public void deleteIndexedDocument(CollectionMetaData collectionMetaData, JSONObject document) throws JsonProcessingException {...}

    1 usage
    public List<JSONObject> getDocumentByProperty(CollectionMetaData collectionMetaData, String property, String value) throws JsonProcessingException {...}

    1 usage
    public void updateDocument(CollectionMetaData collectionMetaData, Map<String, Object> body, Path path) throws JsonProcessingException {...}
}
```

4.6.5 Affinity Utility.

Here it provides a set of methods for managing affinity-related operations, including adding, deleting, and reflecting affinities. It

interacts with other utility classes to perform these operations and maintain data consistency within the application.

```java
@Slf4j
@Component
public class AffinityUtil {
    5 usages
    private final IndexUtil indexUtil;
    5 usages
    private final NodeUtil nodeUtil;

    public AffinityUtil(IndexUtil indexUtil, NodeUtil nodeUtil) {...}

    1 usage
    public String findAffinityNode(String documentId) {...}

    1 usage
    public QueryResponse addAffinity(QueryRequest queryRequest) {...}

    1 usage
    public QueryResponse deleteAffinityDocument(QueryRequest queryRequest) {...}

    1 usage
    public QueryResponse deleteAffinitiesByDatabase(QueryRequest queryRequest) {...}

    1 usage
    public QueryResponse deleteAffinitiesByCollection(QueryRequest queryRequest) {...}


    1 usage
    public QueryResponse reflectAffinity(QueryRequest queryRequest) {...}
}
```

4.6.6 JSON Utility.

This Utility provides various methods for working with JSON data in the node. It uses the Jackson library for JSON serialization and

deserialization, It provides flexibility for handling JSON data in various formats and converting it to Java objects.

```java
public class JSONUtil {
    7 usages
    private static final ObjectMapper mapper = new ObjectMapper();

    2 usages
    public static <T> String generateJsonSchema(Class<T> classToInspect) {...}

    8 usages
    public static <T> T parseObject(JSONObject content, Class<T> valueType) {...}

    1 usage
    public static <T> T parseObject(JSONObject content, TypeReference<T> valueType) {...}

    1 usage
    public static <T> T parseObject(byte[] body, Class<T> valueType) {...}

    no usages
    public static <T> List<T> parseJsonToList(String jsonString, Class<T> clazz) {...}

    2 usages
    public static <T> List<T> parseJsonListToList(List<JSONObject> jsonObjectList, Class<T> clazz) {...}

    1 usage
    private static <T> T mapJsonToObject(JSONObject jsonObject, Class<T> clazz) {...}
}
```

4.6.7 Node Utility.

This class plays a crucial role in managing nodes, balancing their workloads, and handling affinities within a distributed system. It relies on the documentUtil and indexUtil components to interact with the database and indexes, the class helps maintain the state of nodes and their affinities, allowing for efficient distribution across nodes.

```java
public class NodeUtil {
    3 usages
    private final IndexUtil indexUtil;
    3 usages
    private final DocumentUtil documentUtil;

    @Value("${node_url:http://localhost:${server.port}}")
    private String nodeUrl;
    6 usages
    private Node thisNode;

    public NodeUtil(IndexUtil indexUtil, DocumentUtil documentUtil) {...}


    public Node getNode() {...}

    3 usages
    public Node getNodeByNodeId(String nodeId) {...}

    1 usage
    public boolean isInLocalNode(Node node) {...}


    1 usage
    public Node findBalancedNode() {...}

    1 usage
    public List<Node> getAllOtherNodes() {...}

    2 usages
    private List<Node> getAllNodes() {...}

    1 usage
    public void addAffinity(Affinity affinity) {...}

    2 usages
    public void removeAffinity(Affinity affinity) {...}

    2 usages
    public void removeAffinities(List<Affinity> affinities) {...}

    2 usages
    private void saveNode(Node node) {...}
}
```

5.0 Connector Explanation.

5.1 Connection.

5.1.1 NoSQLCommunicationProtocol.

This class is responsible for handling communication between the node and the database using HTTP requests. supports two main operations: post and login , we send a request with a path and token, to be authenticated on a certain URL.

```java
@Slf4j
public class NoSQLCommunicationProtocol {

    1 usage
    public QueryResponse post(String url, String token, JSONObject request, String path) {
        CloseableHttpClient httpClient = HttpClients.createDefault();
        try {
            URIBuilder uriBuilder = new URIBuilder(url).setPath(path);
            HttpPost httpPost = new HttpPost(uriBuilder.build());

            String jsonBody = request.toString();
            StringEntity entity = new StringEntity(jsonBody);
            httpPost.setEntity(entity);

            httpPost.setHeader( name: "Content-Type",  value: "application/json");
            httpPost.setHeader( name: "Authorization",  value: "Bearer " + token);

            HttpResponse response = httpClient.execute(httpPost);
            int statusCode = response.getStatusLine().getStatusCode();
            System.out.println("Response Code: " + statusCode);
            return JSONUtil.parseObject(EntityUtils.toString(response.getEntity()), QueryResponse.class);
        } catch (IOException | URISyntaxException e) {
            throw new RuntimeException(e);
        } finally {
            try {
                httpClient.close();
            } catch (IOException e) {
                log.error(e.getMessage());
            }
        }
    }
    1 usage
    public LoginResponse login(String url, String token, JSONObject request, String path) {...}
}
```

5.1.2 NoSQLDatabaseConnection.

Is responsible for managing the connection to the database, it handles tasks such as initializing database configuration, performing login/authentication, and making query requests to the database, so

here the user can specify the URL of the bootstrapping node and we assign him a token and we give the user the URL of the assigned node so he can sign in to the node with a username and password.

```java
public class NoSQLDatabaseConnection {

    private final NoSQLConfig config = new NoSQLConfig();
    private final NoSQLCommunicationProtocol noSQLCommunicationProtocol = new NoSQLCommunicationProtocol();

    1 usage
    public NoSQLDatabaseConnection() {
        try (InputStream input = getClass().getClassLoader().getResourceAsStream( name: "application.properties")) {
            Properties properties = new Properties();
            if (input == null) {
                throw new IOException("Unable to find db.properties");
            }
            properties.load(input);
            this.config.setBootStrappingNodeUrl(properties.getProperty("NoSQL.Connection.url"));
            this.config.setDatabase(properties.getProperty("NoSQL.Connection.database"));
            this.config.setUser(properties.getProperty("NoSQL.Connection.username"));
            this.config.setPassword(properties.getProperty("NoSQL.Connection.password"));
            login();
        } catch (IOException ex) {
            log.error(ex.getMessage());
        }
    }

    12 usages
    public QueryResponse post(QueryRequest queryRequest) {...}

    1 usage
    private void login() {...}
}
```

5.2 Repository.

5.2.1 NoSQLRepository Interface.

This interface defines a set of methods for performing various operations on the database, it provides a contract for performing CRUD (Create, Read, Update, Delete) operations on entities within the NoSQL database

```java
1 usage  2 implementations
public interface NoSQLRepository<Entity, ID> {
    1 usage  1 implementation
    Entity createDocument(Entity entity);

    2 usages  1 implementation
    boolean createIndex(String index);

    1 usage  1 implementation
    Entity getDocumentByID(ID id);

    1 usage  1 implementation
    List<Entity> getAllDocuments();

    1 usage  1 implementation
    List<Entity> getAllDocumentsByProperty(String property, String value);

    1 usage  1 implementation
    Entity updateDocument(UpdateDocumentQuery updateDocumentQuery);

    1 usage  1 implementation
    boolean deleteIndex(String index);

    no usages  1 implementation
    boolean deleteDocumentById(ID id);
}
```

5.2.2 CRUDNoSQLRepository.

Here we have the implementation for performing CRUD operations on entities within the database through a defined connection and communication protocol.

The user can extend this class to grant access to all the available requests. When the user connects and specifies an Entity and Id, we configure the request and pull the entity type and the super class and cast it, then we initiate the connection of the specified type and create the repository that has the Database.
After, we try to create the database and the collection, it depends on whether the user is connected or not.
The collection name is made according to the entity type he specifies, and since the user provided the class, we can pull the schema from this class.
The user isn't limited since he can implement his logic and override methods.
The user can use this Repository without any complications and without bothering what's hidden from him, the user only enters the configuration info in the App property file .

```java
public abstract class CRUDNoSQLRepository<Entity, ID> implements NoSQLRepository<Entity, ID> {

    3 usages
    private final ConfigRepository configRepository;
    18 usages
    private final NoSQLDatabaseConnection connection;
    16 usages
    private final Class<Entity> entityType;
    1 usage
    public CRUDNoSQLRepository() {...}
    1 usage
    private void init() {...}

    1 usage
    @Override
    public Entity createDocument(Entity entity) {...}
    2 usages
    @Override
    public boolean createIndex(String index) {...}
    1 usage
    @Override
    public Entity getDocumentByID(ID id) {...}
    1 usage
    @Override
    public List<Entity> getAllDocuments() {...}
    1 usage
    @Override
    public List<Entity> getAllDocumentsByProperty(String property, String value) {...}
    1 usage
    @Override
    public Entity updateDocument(UpdateDocumentQuery updateDocumentQuery) {...}
    1 usage
    @Override
    public boolean deleteIndex(String index) {...}
    no usages
    @Override
    public boolean deleteDocumentById(ID id) {...}
}
```

6.0 BootStrapping Node Explanation.

The Bootstrapping node is a pivotal point in the system since it initiates the nodes and is represented as the entry point for users, so it handles the process of making the nodes and mapping authenticated users in a load balanced manner through the system when signing in.

6.1 Repositories.

These Repositories extend the CRUDNoSQLRepository to grant access to all the available database related requests and use it to manipulate the database.

```java
@Repository
public class NodeRepository extends CRUDNoSQLRepository<Node, String> {
}
```

```java
@Repository
public class UserRepository extends CRUDNoSQLRepository<User, String> {
}
```

6.2 AppStartupListener.

This class is responsible for initializing a set of Docker containers representing nodes and performing all setup tasks related to these nodes and we initiate an admin user when the Spring application starts.

```java
@Component
public class AppStartupListener implements ApplicationListener<ApplicationReadyEvent> {

    3 usages
    private final CommunicationService communicationService;
    2 usages
    private final DockerService dockerService;
    2 usages
    private final NodeService nodeService;

    @Value("${node_number:5}")
    private String numOfNodes;
    @Value("${node_image:node}")
    private String nodeImage;
    @Value("${node_network:final_testnet}")
    private String networkName;


    public AppStartupListener(CommunicationService communicationService, DockerService dockerService, NodeService nodeService) {...}

    no usages
    @Override
    public void onApplicationEvent(@NonNull ApplicationReadyEvent event) {...}

    1 usage
    private void initiateNodesMetaData(Node node, List<Node> nodeList) {...}

    1 usage
    private void initiateAdminUser(List<Node> nodeList) {...}
}
```

7.0 Demo App Explanation.

Here I created a Demo Email Application as an example to use my system. I have two repositories simply representing emails and users using my dependency.

```xml
<dependency>
    <groupId>org.example</groupId>
    <artifactId>NoSQL-Connector</artifactId>
    <version>1.0-SNAPSHOT</version>
</dependency>
```

```java
@Repository
public class EmailRepository extends CRUDNoSQLRepository<Email, String> {
}
```

```java
@Repository
public class UserRepository extends CRUDNoSQLRepository<User,String> {
```

8.0 SOLID principles.

SOLID is an acronym for five principles that guide object-oriented design. We assess our codebase's adherence to these principles:

8.1 Single Responsibility Principles (SRP).

SRP Adherence: Each class and module adheres to the SRP, ensuring that they have one reason to change.
That was demonstrated in the system as much as possible and for example you can clearly notice this in the queries classes.

8.2 Open/Closed Principle (OCP).

OCP Adherence: The codebase is designed to be open for extension but closed for modification, facilitating the addition of new features without altering existing code.

If we decide for example to add a new query in the database, we can simply extend the Query interface and implement the necessary logic, thus we guarantee no modification to any existing code.

8.3 Liskov Substitution Principle (LSP).

LSP Adherence: Subclasses are designed to be substitutable for their base classes, promoting polymorphism and ensuring that derived classes adhere to the contract of their base class.

8.4 Interface Segregation Principle (ISP).

ISP Adherence: Interfaces are segregated to include only the methods relevant to the implementing classes, preventing unnecessary dependencies.

8.5 Dependency Inversion Principle (DIP).

DIP Adherence: Dependencies are inverted to rely on abstractions rather than concrete implementations, promoting flexibility and testability.
By creating abstract classes or interfaces, we can define a common API that both the high-level and low-level modules can use to interact with each other. This promotes loose coupling between the modules and makes the system more modular and extensible.

9.0 Clean Code (Uncle Bob).

Clean code principles, by Robert C. Martin (Uncle Bob), promote code that is readable, maintainable, and elegant, so I tried my best to follow these principles as I'm coding.

9.1 Readability.

- Descriptive Naming: Variable and function names are chosen carefully to be descriptive and easy to understand, enhancing code readability.

- Consistent Formatting: Code formatting adheres to consistent style guidelines, ensuring uniformity throughout the project.

9.2 Maintainability.

- Modularization: The code is modular, with distinct components responsible for specific functionalities, making it easier to maintain and extend.

- Comments : Comments are avoided as much as possible while the code can be self explained.

- Documentation: Documentation is provided where necessary to explain complex logic and make sure that future developers can understand the code.

9.3 Simplicity.

- I tried my best to avoid any unnecessary complexity and always look for straightforward solutions.

- Single Responsibility Principle (SRP): Again, each class and function adheres to the SRP, ensuring that they have a single, well-defined purpose.

10.0 DevOps Practices .

10.1 Maven.

I have utilized Maven as a build tool, which is an incredibly powerful tool that makes it easier to add library dependencies from the Maven repository where I have incorporated these libraries into my system. Additionally, I separated my Maven project from my Spring project, and used Maven to both install and package the JAR file for use in the Spring application.

10.2 Git.

Using Git even as a solo developer provides version control, backup, branching for experimentation, and potential for future collaboration, making it a valuable tool for code management and organization.

10.4 Docker.

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

11.0 Effective Java Items (Joshua Bloch).

The Effective Java book by Joshua Bloch offers a set of guidelines and best practices for Java programming. We assess our codebase against some of these items such as :

Item 1: Consider Static Factory Methods over Constructors Factory Methods

- Where appropriate, static factory methods are used to create instances, providing more expressive and flexible object creation.

Item 3: Enforce the singleton property with a private constructor or an enum type.

- ensures a class has only one instance and provides a global point of access to that instance.

Item 9: Prefer try-with-resources to try-finally:

- advises using try-with-resources over try-finally for managing resources such as files, streams, or sockets. Try-with-resources is more concise, less error-prone, and ensures that resources are closed properly.

Item 11: Always override hashCode when you override equals

Item 13: Minimize the accessibility of classes and members:

- you should limit the accessibility of classes, interfaces, and members as much as possible, expose only the necessary public APIs. This encapsulation helps in managing the complexity of the code, reduces the risk of misuse, and allows for future changes without affecting external

Item 15: Minimize Mutability

- Immutability: Immutable objects are preferred where possible to minimize unexpected side effects and simplify concurrent programming.

Item 34: Use Enums instead of int constants:

- Using enums to represent a fixed set of related constants instead of int constants. Enums provide better type safety, eliminate magic numbers, and improve code readability. They also allow you to add behavior to individual constants, making your code more expressive and maintainable."

Item 47: Know and use the libraries:

- This item emphasizes the importance of using libraries and built-in functionality rather than reinventing the wheel in the code.

Item 50: Avoid Strings for Identifiers

- Avoiding String Identifiers: Instead of using string-based identifiers, we employ enums or type-safe alternatives to improve code reliability.