Capstone Project

ATYPON

Decentralized Cluster-based No-SQL Database System

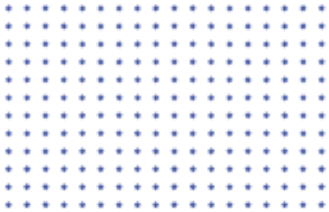
Done By: Hassan Al-Safadi

September- 2023

Contents

Figures

SUMMARY



This Report illustrates my implementation for Atypon Capstone Project: Decentralized Cluster-based No-SQL Database System, it provides many components that real world databases include.

WHAT IS JSON?

JSON is a text-based format for representing structured data. It's used to serialize (convert data into a format that can be easily transmitted or stored) and deserialize (convert the serialized data back into its original structure) data. It's a versatile format because it can represent various data types such as strings, numbers, objects, arrays, Booleans, and null values.

JSON is designed to be human-readable and easy for both humans and machines to understand. It uses a syntax consisting of key-value pairs, where keys are strings enclosed in double quotes, followed by a colon, and then a value. Key-value pairs are separated by commas, and objects are enclosed in curly braces {} while arrays are enclosed in square brackets [].

JSON supports two primary data structures:

- Objects: Objects are collections of key-value pairs. Each key is a unique string within the object, and values can be of any data type (string, number, object, array, boolean, or null). Objects are enclosed in curly braces {}, example:

```
{"name":"John", "age":30, "car":null}
```

- Arrays: Arrays are ordered collections of values. Values in an array can be of any data type, including other arrays or objects. Arrays are enclosed in square brackets [], example:

```
["Ford", "BMW", "Fiat"]
```

JSON finds widespread application in several domains, serving as a versatile tool for a variety of purposes. It is extensively employed in web applications for data exchange between clients and servers, frequently utilized within APIs to facilitate the transmission and reception of structured data. JSON's human-readable and editable nature also makes it a preferred choice for configuration files. Moreover, some NoSQL databases adopt a JSON-like format for data storage, simplifying the management of semi-structured or schema-less data, thereby illustrating JSON's adaptability and ubiquity in modern data-driven ecosystems.

NO-SQL DATABASE

SQL databases are common way to store structured data, its known in their ways to store data in tables, see figure

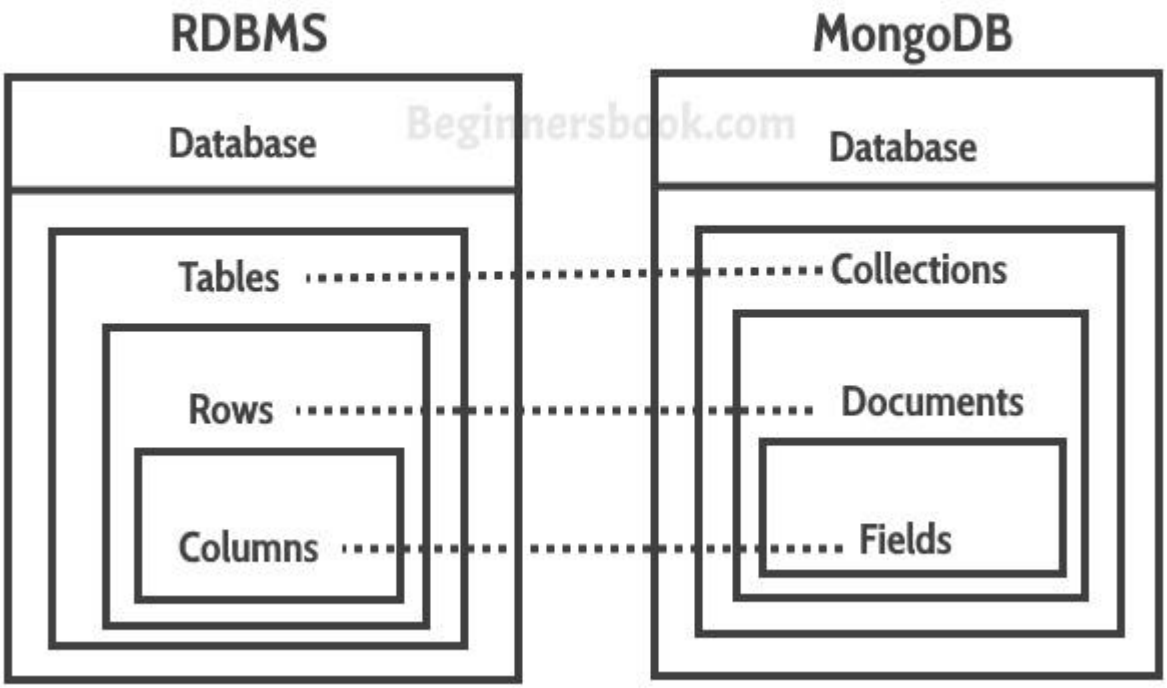


Figure 1: SQL vs NOSQL.

As you can see, in the relational databases we have 3 concurrent hierarchies, Database → Table→ Row→ Column, in mongo DB which is non-relational database, we store the data in this form Database→Collection→ Document→ Fields.

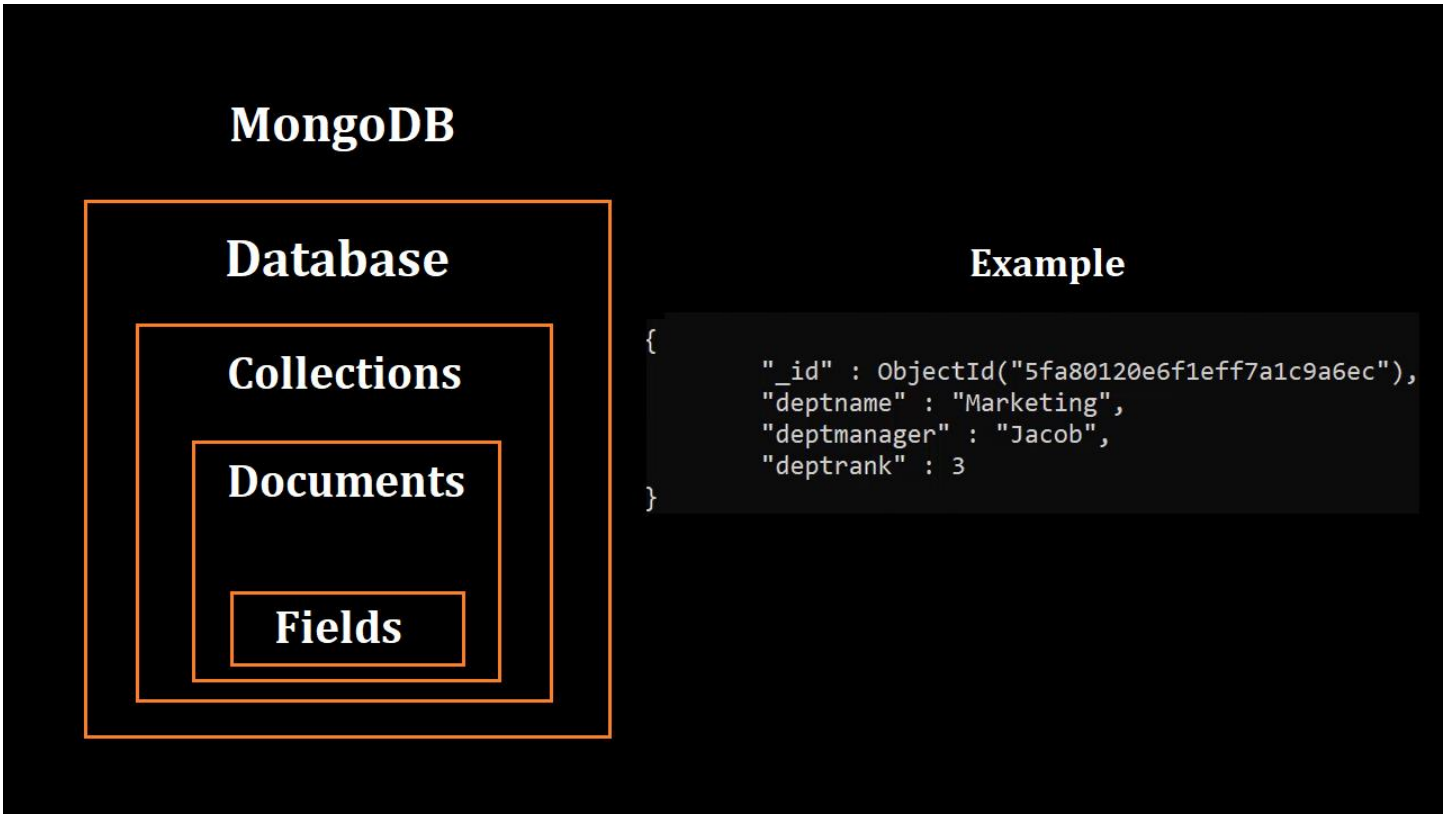


Figure 2: NOSQL Storing Architecture.

Why we need NO-SQL

NoSQL databases are used primarily to address the limitations of traditional relational databases in handling massive and unstructured data sets. They are designed to provide more flexibility and scalability, making them ideal for applications and use cases where data is constantly changing, or where high volumes of data need to be ingested and processed in real-time. NoSQL databases can handle various data types, including semi-structured and unstructured data, so for example suppose you have some form and you want people to fill it, but unfortunately, some answers are optional, and you used an SQL database, you will end up needing to fill null where some field is empty, also, that would waste space because you don't need these null which typically needs a space too.

How to get a document

In SQL, we are familiar with the process of retrieving specific rows using the SELECT statement, whether it involves the primary key or any other indexed field. However, it's important to note that fetching a row by its primary key is notably faster. An example from a YouTube video illustrates this discrepancy: when a SELECT statement is executed using the primary key, it takes a mere 0.3 seconds, whereas querying based on another field takes significantly longer, approximately 5 seconds – a tenfold difference in speed. But what accounts for this substantial gap in performance? To grasp the underlying reasons, let's delve into some foundational concepts.

JACKSON LIBRARY

There are two ways to handle the JSON documents,

1- GSON: this library is provided by Google.

2- JACKSON: this library is an open-source project, and it is used by the Spring framework to handle and deal with the JSON Objects.

Both libraries are valid, but because the spring framework can automatically do marshalling and unmarshalling for objects using the Jackson library, I will be using the Jackson library.

FILE ORGONIZATION AND INDEXING

Because this is a database, we need a stable way to store our documents, of course it will be stored in files, but the way we store and retrieve depends on usage mainly, also data structure varies with the use, but let’s start with the File Organization first.

Some Valid Approaches

1- Hash Index: A simple way for storing documents in our database, is key-value data storing, we have a hash map that we use to store keys names, and for values, we store the value offset byte, for more illustration, see figure:

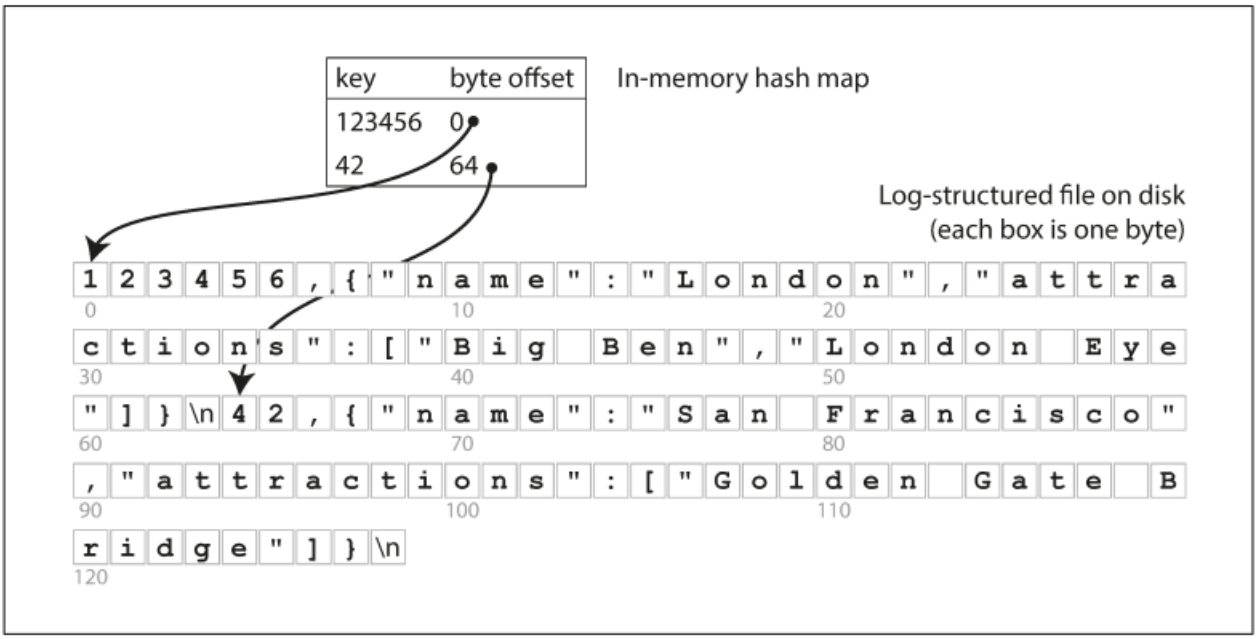


Figure 3: Hash Index Mechanism.

This might seem simple, but it could work efficiently, because as you can see, the Hash Map is used here which will remain in memory, and retrieving and writing are very fast, it could be used in situations where updating to the value is frequent, like a YouTube video that need to be updated with the views, in fact, this approach is used in Bit Cask, default engine in Riak NoSQL database, but it has some down sides:

- This might not suit situations where you want to store lots of keys, only if you want to update the value of some key frequently, because all keys are stored in the memory, this could cause problems with the stack.
- Writing in the file system is appending, but never really removing from the textual file, this means that the file will be essentially like a log for all values, this means with many updates, the

file will get larger and larger, which might fill the memory fast, this could be solved with compaction, see figure:

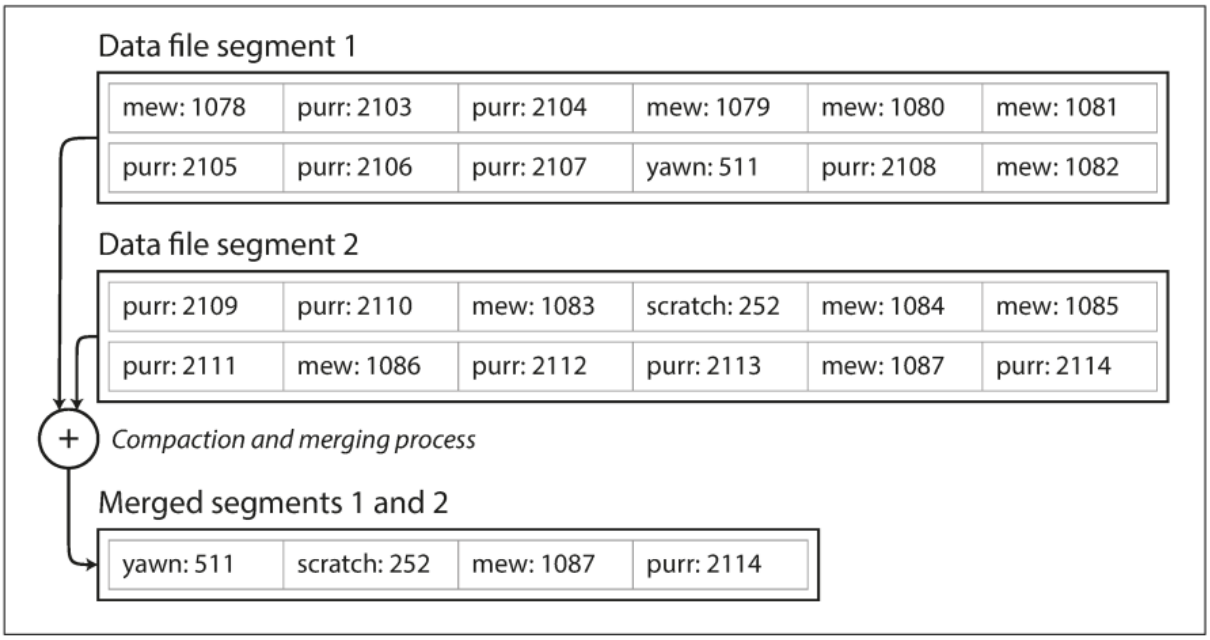


Figure 4: Compaction on segments.

As you can see, the new way is that we write to segments, then after some determined size for the segment, some background thread performs the compaction and simply delete the old segments.

Static and Variable Record Storage

This approach is used more in the SQL databases, because in SQL each type has some limited size of bytes that it can't exceed, see figure:

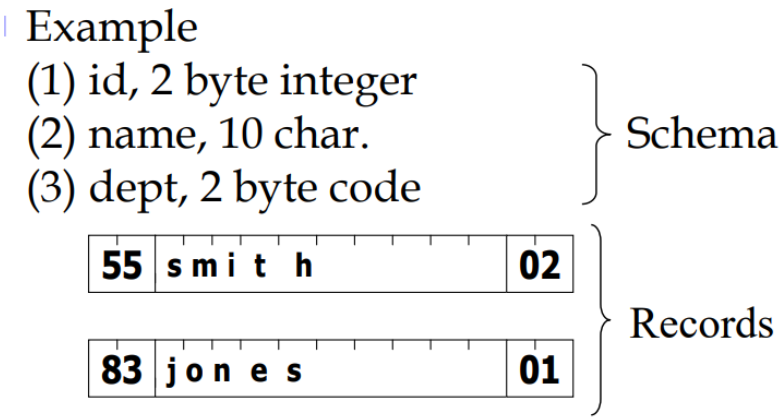


Figure 5: Static Record Storage.

As you can see, each value will have max number of bytes that it can't exceed, for example in sql, we can see that int is 4 bytes and so on, and this won't be a valid for our solution because at most cases, we don't have specific size for the values.

about variable Record Storage, I have implemented it in this way:

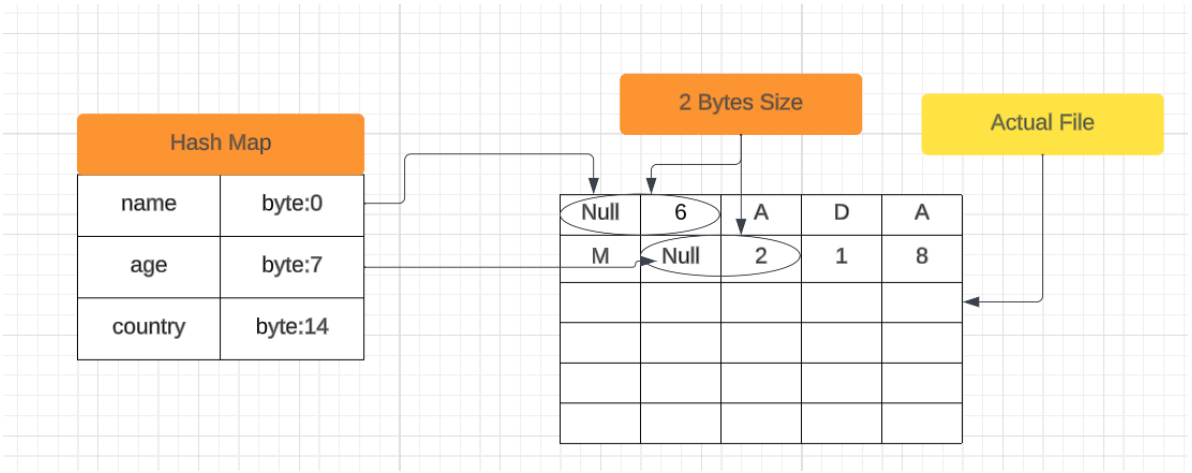


Figure 6: Variable Record Storage.

As you can see, for each document we will have a hash map, that will be mapping each field with its value specific byte offset, and before starting the actual read, we will read two bytes indicating the real size of the value, this makes the values variable, but as you might have seen, this is valid if you want to read, but if you want to write or update, you have to update the new size, with shifting the afterwards values, this actually creates a delay while doing this process, because you will need to update the rest of values byte offsets.

Now let’s see my solution, but before that, I want to introduce some data structures:

B+ Tree

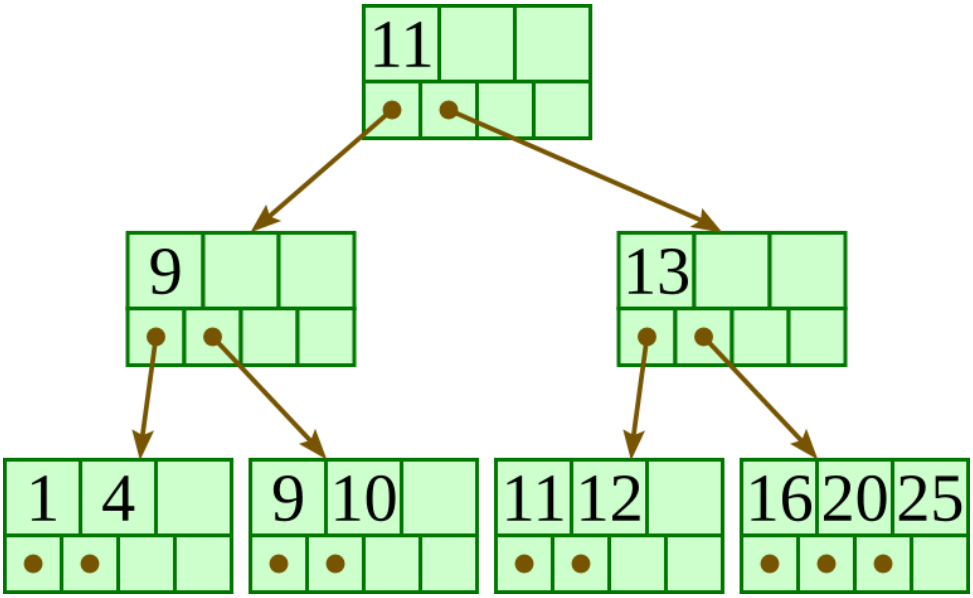


Figure 7: B+ Tree.

Features of B+ Trees

- **Balanced:** B+ Trees are self-balancing, which means that as data is added or removed from the tree, it automatically adjusts itself to maintain a balanced structure. This ensures that the search time remains relatively constant, regardless of the size of the tree.
- **Multi-level:** B+ Trees are multi-level data structures, with a root node at the top and one or more levels of internal nodes below it. The leaf nodes at the bottom level contain the actual data.
- **Ordered:** B+ Trees maintain the order of the keys in the tree, which makes it easy to perform range queries and other operations that require sorted data.
- **Fan-out:** B+ Trees have a high fan-out, which means that each node can have many child nodes. This reduces the height of the tree and increases the efficiency of searching and indexing operations.
- **Cache-friendly:** B+ Trees are designed to be cache-friendly, which means that they can take advantage of the caching mechanisms in modern computer architectures to improve performance.

- **Disk-oriented:** B+ Trees are often used for disk-based storage systems because they are efficient at storing and retrieving data from disk.

Why Use B+ Tree?

1. B+ Trees are the best choice for storage systems with sluggish data access because they minimize I/O operations while facilitating efficient disc access.
2. B+ Trees are a good choice for database systems and applications needing quick data retrieval because of their balanced structure, which guarantees predictable performance for a variety of activities and facilitates effective range-based queries.

Time complexity of the B+Tree

- Worst case search time is log(n)
- Average case search time log(n)
- Best case search time log(n)
- Worst case insertion time log(n)
- Worst case deletion time log(n)

Because the implementation of the B+ tree is not easy, I have found an already implemented B+ tree in github and modified it.

Hash Table

Hash tables are very common in java, to store some key and retrieve the value using the key, hash tables use the idea of chaining, see figure:

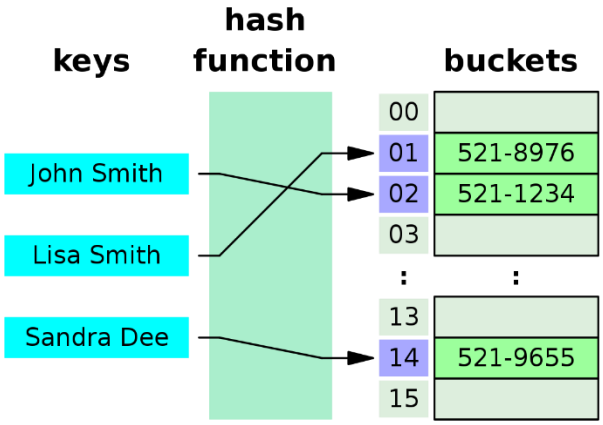


Figure 8: Hash Table.

As you can see the keys enter some hash function that takes into consideration the number of buckets, then find the suitable place for the key, a bucket is just a linked list that includes many key value pairs, this idea is called *chaining*.

Hash Multimap

Multimaps are great way to store key with many values, see figure:

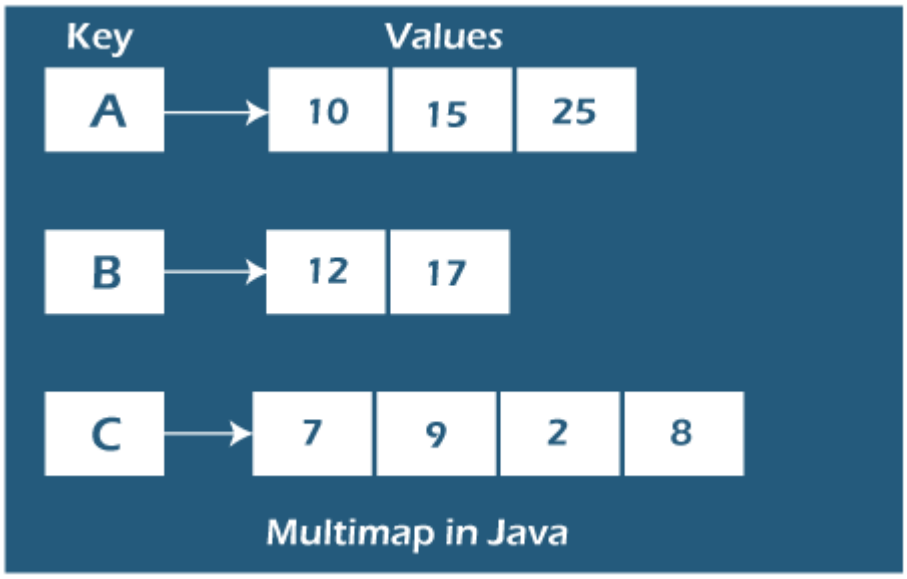


Figure 9: Hash Multimap.

As you can see, this is a good way to store a key with many values, further explanation on why I needed this in next sections.

My File System

As we have seen, we have seen many ways to store documents, but because we don't want to just index the documents, we also need to index the properties, indexing the properties is important because it lets the user to easily retrieve documents that shares some value for some property, this is actually what real databases do, lets delve into my design.

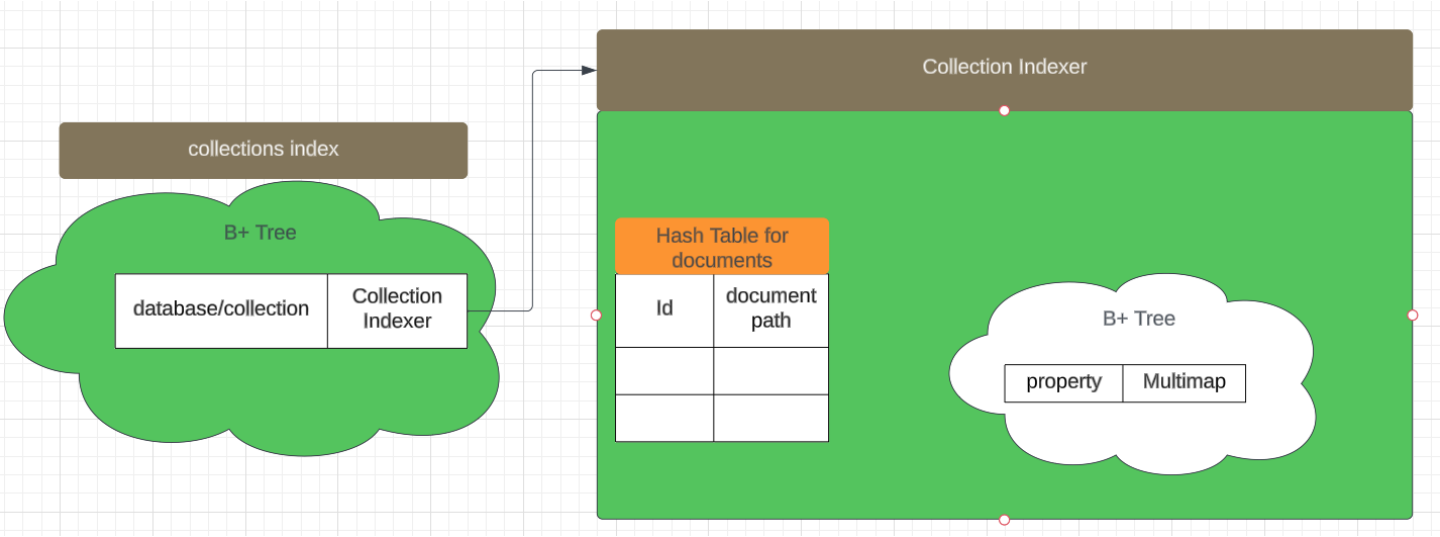


Figure 10: Base explanation for design.

As you can see, we have a main index, called **collections** index, this index will be storing all collections, and the database/collection will be the key, the value will be an object called Collection Indexer, this object will have two indexing data structures, a hash table for *indexing the documents*, and another B+ tree to *index the properties*, this way, we can retrieve any document fast by the id, some notes about my design:

- The B+ Tree can store string keys because the strings in java are comparable, it can compare between two strings lexicographically, we don't really care about how it is comparing strings, we care that it will store it and retrieve it fast.
- When the user requests a document to read or write to, this will happen fast because, first we will use the collections index to find the collection indexer for that collection, then we will look for that specific

id using the hash table in the collection indexer, and read the document, this will happen fast because all these data-structures are stored in-memory, and it is doing one disk access.

- Attempting to find a collection or a document that does not exist will be throwing an exception and that exception is handled and warning the user that the requested document or collection does not exist.

But after all that, how does the property indexer really work.

Property Indexer

Indexing the properties is important because it exists in relational and non-relational databases, and there are two ways to do that:

1- Make the user decides what properties to index, and for that, users can write queries to index properties, this will take some time when creating the index, because you will need to see all documents that have that property and find their values.

2- The system automatically indexes all properties, and that how I implemented it, see figure:

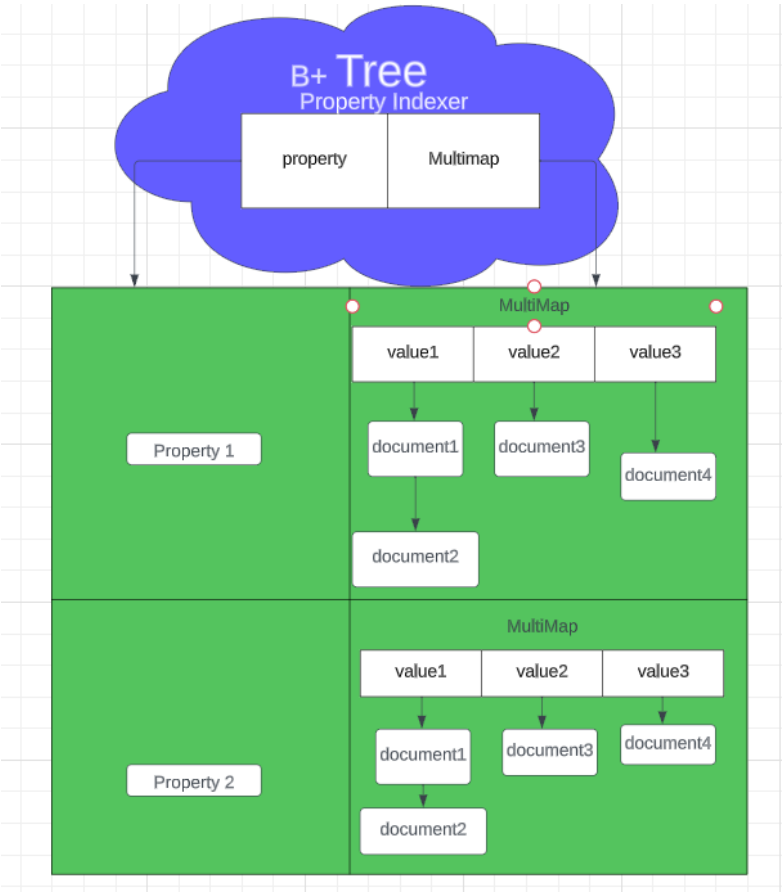


Figure 11: Property Indexer.

As you can see, each collection will have a property indexer, this property indexer will map each property with a multimap, this multimap will be storing values, and same value might be shared for multiple documents, so this is how we can retrieve documents with some property value.

SPRING

Spring is an excellent framework in java that I prefer to call as umbrella, this umbrella includes many frameworks that are made by intelligent software engineers to satisfy the developer's needs, but before diving into the topic, lets introduce some concepts:

1- Inversion Of Control (IOC): Inversion of Control is a principle in software engineering which transfers the control of objects or portions of a program to a container or framework. The advantages of this architecture are:
   - decoupling the execution of a task from its implementation.
   - making it easier to switch between different implementations.
   - greater modularity of a program.
   - greater ease in testing a program by isolating a component or mocking its dependencies and allowing components to communicate through contracts.

2- Dependency Injection (DI): Dependency injection is a pattern we can use to implement IoC, where the control being inverted is setting an object's dependencies. Connecting objects with other objects, or "injecting" objects into other objects, is done by an assembler rather than by the objects themselves.

As you can see, we transfer the control to the framework, so it controls all the objects in the application, injecting is done by the app, objects in spring are called Beans, each Bean is stored in the spring container so it can be easily retrieved from anywhere in the app, let us see some concepts:

1- Bean: is an object that is being by the spring container, so for example if you annotate some function with @Bean, it will be initiated by the spring app, and you can use this bean from anywhere in the app.

2- Component: a component class is just another form of bean, this will allow the spring app to initiate some object from that class, using the constructor of that class, it may need other beans to be initiated, for example the constructor may take several arguments, these are beans and they are injected to that constructor at runtime.

3- Configuration and Service: also, other forms of beans, Configuration class tells the spring app that this class will be a source of beans definitions, see this example:

```
@Configuration
public class AppConfig {
   @Bean
   public MyBean myBean() {
      return new MyBean();
   }
}
```

As you can see, this class will be a resource for beans.

SYSTEM DESIGN

Design plays a crucial role in this system. Before explaining the components, I will show the exposed APIs, so it paves the way to understanding the design.

APIs

1- Database Controller: this API will have many endpoints:

- GET /database/create/{db name} this will create a database in the system.
- GET /database/delete/{db name} this will delete the database in the system.

- POST /database/create/{db name}/{collection name} this will create a collection on the specified database, this is a post because it can take a schema for that collection, and validate documents using that schema.
- GET /database/delete/{db name}/{collection name} this will delete some collection in the specified database.
- GET /database/findDatabases this will return all databases names in the system.
- GET /database/findCollections/{db name} will return all the collections names in the specified database.

2- Collection Controller: this API will have many endpoints:

- POST /collection/create/{db name}/{collection name} this will store the specified document included in the request body in the specified collection.
- GET /collection/delete/{db name}/{collection name}/{id} this will delete the document with the specified id from the specified collection.
- GET /collection/find/{db name}/{collection name}/{id} returns the document with the specified id.
- GET /collection/find/{db name}/{collection name} returns all the documents in the specified collection.
- POST /collection/findByFilter/{db name}/{collection name} returns all the documents with specified property name: property value in the request body.

3- Documents endpoint: this API have many endpoints:

- POST /document/addField/{db name}/{collection name}/{id} this will add some field specified in the request body to the specified document.
- GET /document/deleteField/{db name}/{collection name}/{id}/{field name} this will delete some field specified by the request header.

- POST /document/updateField/{dbname}/{collection name}/{id} this will update some field specified in the request body in the specified document.

Now after we have seen the APIs, we can at least deconstruct the behaviors into three, **database**, **collection**, and **document**, let's see main components:

Query

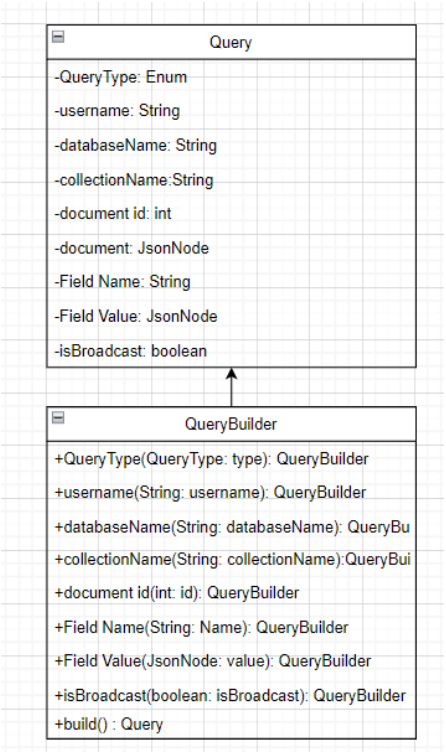
There are many ways to design the query, but we can notice there are common information between endpoints we have seen, for example database name, collection name and so on, so I have decided to use the builder pattern, a query will have:

1- Query Type: this tells the system the type of the query, this have specific types so it is Enum type, types are:
   - CreateDatabase
   - DeleteDatabase
   - GetDatabases
   - CreateCollection
   - DeleteCollection
   - GetCollections
   - AddDocument
   - DeleteDocument
   - FindDocument
   - FindByFilter
   - FindDocuments
2- Database information: this includes database name, collection name.
3- Collection information: this includes id of the document, document, and schema.

4- Document information: this includes field name, field value.

5- Communication data: this includes the username of the querier, broadcast Boolean.

And that's all for Query, to make things easier, we used the builder to set the necessary attributes in the query, see figure.



Lombok library provides an annotation called @Builder that lets you make a query builder just by annotating the Query class with it.

Collection Indexer

As we have seen before, each collection will have a collection indexer, it is capable of many things, including.

1- Write: this will write a specific document to the collection, see code

```
public boolean write(JsonNode jsonDocument) throws IOException,
KeyAlreadyExistsException, SchemaViolationException{
    int id = 0;
    if(jsonDocument.has("_id")) {
        if (jsonDocument.get("_id").isInt()) {
            id = jsonDocument.get("_id").asInt();
        } else if (jsonDocument.get("_id").isTextual()) {
```

```
            id = Math.abs(jsonDocument.get("_id").asText().hashCode());
        }
    }
    else {
        id = Math.abs(jsonDocument.hashCode());
        ObjectNode node = objectMapper.createObjectNode();
        node.put("_id",id);
        node.setAll((ObjectNode) jsonDocument);
        jsonDocument = node;

    }
    if(index.containsKey(id)||!jsonDocument.isObject()){
        throw new KeyAlreadyExistsException("Key already exists");
    }
    int finalId = id;
    try {
        if (schemaValidator.getSchema(database,collectionName) != null) {
            if (!schemaValidator.isValid(database,collectionName,jsonDocument)) {
                throw new SchemaViolationException("Schema Violation");
            }
        }
    }
    catch (NullPointerException | ProcessingException ignored){
       ignored.printStackTrace();
    }
    jsonDocument.fields().forEachRemaining(entry -> {
        if(propertyIndexer.search(entry.getKey()) == null)
            propertyIndexer.insert(entry.getKey(), HashMultimap.create());
        try {

propertyIndexer.search(entry.getKey()).put(objectMapper.writeValueAsString(entry.getValue
()), finalId);
        } catch (JsonProcessingException e) {
                e.printStackTrace();
        }
        });
    if(new File("./data/Database/"+database+"/" + collectionName + "/" + id +
".dat").createNewFile()){
        objectMapper.writeValue(new File("./data/Database/"+database+"/" + collectionName
+ "/" + id + ".dat"), jsonDocument);
        index.put(finalId, "./data/Database/"+database+"/" + collectionName + "/" + id +
".dat");
        return true;
    }
    else {
        return false;
    }
}
```

this will check if the document has and _id field, if it has, it will check if it is stored, if so it will throw a KeyAlreadyExists exception, if not it will store it, if there is no field called _id, it will generate one, after it is ready to be stored, it will store all the fields of the document in the property indexer so it easily retrieved whenever needed, and after that it will be stored.

2- Read: this will read a document just by its id, see code:

```
public JsonNode read(int id) throws IOException{
    try {
        if(!index.containsKey(id)){
            throw new IOException("Document not found");
        }
        JsonNode jsonNode = getJson.getJsonObjectFromFile(index.get(id));
        return jsonNode;
    }
     catch (ExecutionException e) {
        throw new RuntimeException(e);
    }
}
```

if the document does not exist in the indexer, it will throw an exception indicating that it can't read because there is no such file with that id.

3- deleteById: this will delete a document by the id of that document.

```
public boolean deleteById(int id) throws KeyException{
        if (!index.containsKey(id)) {
            throw new KeyException("Key not found");
        }
        try {
            JsonNode jsonNode = objectMapper.readTree(new File(index.get(id)));
            jsonNode.fields().forEachRemaining(entry -> {
                propertyIndexer.search(entry.getKey()).remove(entry.getValue().asText(),
id);
            });
            getJson.cacheDelete(index.get(id));
            index.remove(id);
            new File("./data/Database/" + database + "/" + collectionName + "/" + id +
".dat").delete();
            return true;
        }
     catch (Exception e) {
        return false;
    }
}
```

using the object mapper, we will read the document, then we will delete the document properties from the property indexer.

4- readAll():

```
public List<JsonNode> readAll() {
    List<JsonNode> result = new ArrayList<>();
    int i = 0;
    for (int id : index.keySet()) {
        try{
            result.add(read(id));
        }
        catch (Exception e){
            e.printStackTrace();
        }
    }
    return result;

}
```

5- deleteAll():

```
public boolean deleteAll(){
    try {
        for (int id : index.keySet()) {
            deleteById(id);
        }
        return true;
    }
    catch (Exception e){
        e.printStackTrace();
        return false;
    }
}
```

6- findBy():

```
public List<JsonNode> findBy(String propertyName, JsonNode propertyValue) throws
JsonProcessingException {
    List<Integer> ids =
propertyIndexer.search(propertyName).get(objectMapper.writeValueAsString(propertyValue)).
stream().toList();
    List<JsonNode> jsonNodes = new ArrayList<>();
    for (int id : ids) {
        try {
            jsonNodes.add(read(id));
        } catch (IOException e) {
            e.printStackTrace();
        }

    }
    return jsonNodes;
}
```

7- deleteField:

```
public boolean deleteField (int id, String fieldName) throws IOException,
ExecutionException {
    if (!index.containsKey(id)) {
        throw new IOException("Document not found");
    }
    try {
        if
(!propertyIndexer.search(fieldName).containsEntry(objectMapper.writeValueAsString(read(id
).get(fieldName)), id)) {
            throw new IOException("Field not found");
        }
    }
    catch (Exception e){
        throw new IOException("Field not found");
    }
    JsonNode jsonNode = objectMapper.readTree(new File(index.get(id)));
    String string = jsonNode.get(fieldName).asText();
    ((ObjectNode)jsonNode).remove(fieldName);
    propertyIndexer.search(fieldName).remove(string, id);
    objectMapper.writeValue(new File("./data/Database/"+database+"/" + collectionName +
"/" + id + ".dat"), jsonNode);
    getJson.cacheRefresh(index.get(id), jsonNode);
    return true;
}
```

8- updateField():

```
public boolean updateField (int id, String fieldName, JsonNode fieldValue) throws
IOException, ExecutionException {
    String s = objectMapper.writeValueAsString(objectMapper.readTree(new
File(index.get(id))).findValue(fieldName));
    if (!index.containsKey(id)) {
        throw new IOException("Document not found");
    }
    if(!propertyIndexer.search(fieldName).containsEntry(s, id)){
        throw new IOException("Field not found");
    }

    JsonNode jsonNode = objectMapper.readTree(new File(index.get(id)));
    propertyIndexer.search(fieldName).remove(s, id);
    ((ObjectNode)jsonNode).put(fieldName, fieldValue);
    propertyIndexer.search(fieldName).put(objectMapper.writeValueAsString(fieldValue),
id);
    objectMapper.writeValue(new File("./data/Database/"+database+"/" + collectionName +
"/" + id + ".dat"), jsonNode);
    getJson.cacheRefresh(index.get(id), jsonNode);
    return true;
}
```

9- addField():

```
public boolean addField (int id, String fieldName, JsonNode fieldValue) throws
IOException, ExecutionException {
    if (!index.containsKey(id)) {
        throw new IOException("Document not found");
    }
    if(propertyIndexer.search(fieldName) == null){
        propertyIndexer.insert(fieldName, HashMultimap.create());
    }
    else if
(propertyIndexer.search(fieldName).containsEntry(objectMapper.writeValueAsString(read(id)
.get(fieldName)), id)){
            throw new IOException("Field already exists");
    }
    System.out.println(index.get(id)+"--->index.get(id)");
    JsonNode jsonNode = objectMapper.readTree(new File(index.get(id)));
    ((ObjectNode)jsonNode).put(fieldName, fieldValue);
    propertyIndexer.search(fieldName).put(objectMapper.writeValueAsString(fieldValue),
id);
    objectMapper.writeValue(new File("./data/Database/"+database+"/" + collectionName +
"/" + id + ".dat"), jsonNode);
    getJson.cacheRefresh(index.get(id), jsonNode);
    return true;
}
```

Layer1 – Database Services

To handle the query, we first need to make a layer that handles the query, which we can use to handle the query, see figure:

As you can see, layer 1 is the direct contact to the database, which will create, delete or update databases, collections and documents, and then, the layer 2 will use these services to do the specific task, as we have seen earlier, we will have a collections indexer, this indexer will include a reference to all collections in all databases, this reference is the Collection Indexer that will be doing everything inside the collection, including but not limited to indexing, which needs to be there so if the user requests some document, property or something else, will be directly accessing the memory, instead of storing some logs and do many not necessary on-disk accesses.

Layer 1 here is something like the DAO layer in systems architecture that access database, it introduces its APIs that you can use to store your documents and databases, but I preferred not to call it DAO because it uses different approach.

We have three services:

1- Database Service: This will do many tasks:

- Create Database.

```
public Status createDatabase(String databaseName) {
    return new File("./data/Database/" + databaseName).mkdir() ?
            new Status(Status.StatusType.Success, "Database created successfully") :
            new Status(Status.StatusType.Failure, "Database already exists");
}
```

- Delete Database.

```
public Status dropDatabase(String databaseName) {
    try {
        for(String collectionName : new File("./data/Database/" + databaseName).list()) {
            FileUtils.deleteDirectory(new File("./data/Database/" + databaseName + "/" +
collectionName));
            collections.delete(databaseName + collectionName.replace(".dat", ""));
        }
        boolean isDeleted = new File("./data/Database/" + databaseName).delete();
        return isDeleted ?
                new Status(Status.StatusType.Success, "Database deleted successfully") :
                new Status(Status.StatusType.Failure, "Database not found");
    }
    catch (Exception e) {
        return new Status(Status.StatusType.Failure, "Database not found");
    }
}
```

- Create Collection.

```
public Status createCollection(String databaseName, String collectionName, JsonNode
schema) {
    try {
        CollectionIndexer collectionIndexer = new CollectionIndexer(databaseName,
collectionName);
        if(schema != null && !schema.isNull()){
            System.out.println("Schema is not null");
            collectionIndexer.setSchema(schema);
        }
        collections.insert(databaseName + collectionName, collectionIndexer);
        return new Status(Status.StatusType.Success, "Collection created successfully");
    }
    catch (Exception e) {
        e.printStackTrace();
        return new Status(Status.StatusType.Failure, e.getMessage());
```

```
    }
}
```

- Delete Collection.

```
public Status dropCollection(String databaseName, String collectionName) {
    try {
        if(new File("./data/Database/" + databaseName + "/" + collectionName).exists()){
            FileUtils.deleteDirectory(new File("./data/Database/" + databaseName + "/" +
collectionName));
            collections.search(databaseName + collectionName).deleteAll();
            collections.delete(databaseName + collectionName);
            return new Status(Status.StatusType.Success, "Collection deleted
successfully");
        }
        else{
            return new Status(Status.StatusType.Failure, "Collection not found");
        }
    } catch (Exception e) {
        return new Status(Status.StatusType.Failure, "Collection not found");
    }
}
```

- Get All Databases.

```
public Status getDatabases() {
    return new Status(Status.StatusType.Success, Arrays.stream(new
File("./data/Database").list()).collect(StringBuilder::new, StringBuilder::append,
StringBuilder::append).toString());
}
```

- Get All Collections inside a database.

```
public Status getCollections(String databaseName) {
    return new Status(Status.StatusType.Success, Arrays.stream(new
File("./data/Database/" + databaseName).list()).collect(StringBuilder::new,
StringBuilder::append, StringBuilder::append).toString());
}
```

2- Collection Service: including tasks:

- Create a document.

```
public Status addDocument(String databaseName, String collectionName, JsonNode
jsonDocument) {
    try {
        boolean isExecuted = collections.search(databaseName +
collectionName).write(jsonDocument);
        if (isExecuted) {
            return new Status(Status.StatusType.Success, "Document added successfully");
        } else {
            return new Status(Status.StatusType.Failure, "Document could not be added");
```

```
        }
    } catch (KeyAlreadyExistsException | IOException |SchemaViolationException exception)
{
        return new Status(Status.StatusType.Failure, exception.getMessage());
    }
}
```

- Delete a document.

```
public Status removeDocument(String databaseName, String collectionName, int id) {
    try {
        if (collections.search(databaseName + collectionName).deleteById(id)) {
            return new Status(Status.StatusType.Success, "Document deleted
successfully");
        } else {
            return new Status(Status.StatusType.Failure, "Document not found");
        }
    } catch (KeyException e) {
        return new Status(Status.StatusType.Failure, e.getMessage());
    }
}
```

- Get a document by id.

```
public JsonNode findDocument(String databaseName, String collectionName, int id) {
    try {
        JsonNode jsonNode = collections.search(databaseName + collectionName).read(id);
        if (jsonNode != null) {
            return jsonNode;
        } else {
            return objectMapper.nullNode();
        }
    } catch (IOException e) {
        return objectMapper.nullNode();
    }
}
```

- Get a document by filter.

```
public List<JsonNode> getDocumentsByField(String databaseName, String collectionName,
String key, JsonNode value) {
    try {
        List<JsonNode> jsonNodes = collections.search(databaseName +
collectionName).findBy(key, value);
        return jsonNodes;
    } catch (Exception e) {
        return new ArrayList<>();
    }
}
```

- Get all documents in a specific collection.

```
public List<JsonNode> findDocuments(String databaseName, String collectionName) {
    try {
        List<JsonNode> jsonNodes = collections.search(databaseName +
collectionName).readAll();
        return jsonNodes;
    } catch (Exception e) {
        return new ArrayList<>();
    }
}
```

3- Document Service: including tasks:

- Add a field.

```
public Status addField(String databaseName, String collectionName, Integer index, String
fieldName, JsonNode fieldValue){
    try {
        boolean isExecuted =
collections.search(databaseName+collectionName).addField(index,fieldName,fieldValue);
        if(isExecuted){
            return new Status(Status.StatusType.Success, "Field added successfully");
        }
        else{
            return new Status(Status.StatusType.Failure, "Field could not be added");
        }
    }catch (Exception e){
        e.printStackTrace();
        return new Status(Status.StatusType.Failure, e.getMessage());
    }
}
```

- Delete a field.

```
public Status removeField(String databaseName, String collectionName,Integer index,
String fieldName){
    try {
        boolean isExecuted =
collections.search(databaseName+collectionName).deleteField(index,fieldName);
        if(isExecuted){
            return new Status(Status.StatusType.Success, "Field deleted successfully");
        }
        else{
            return new Status(Status.StatusType.Failure, "Field could not be deleted");
        }
    }catch (Exception e){
        e.printStackTrace();
        return new Status(Status.StatusType.Failure, e.getMessage());
    }
}
```

- Update a field.

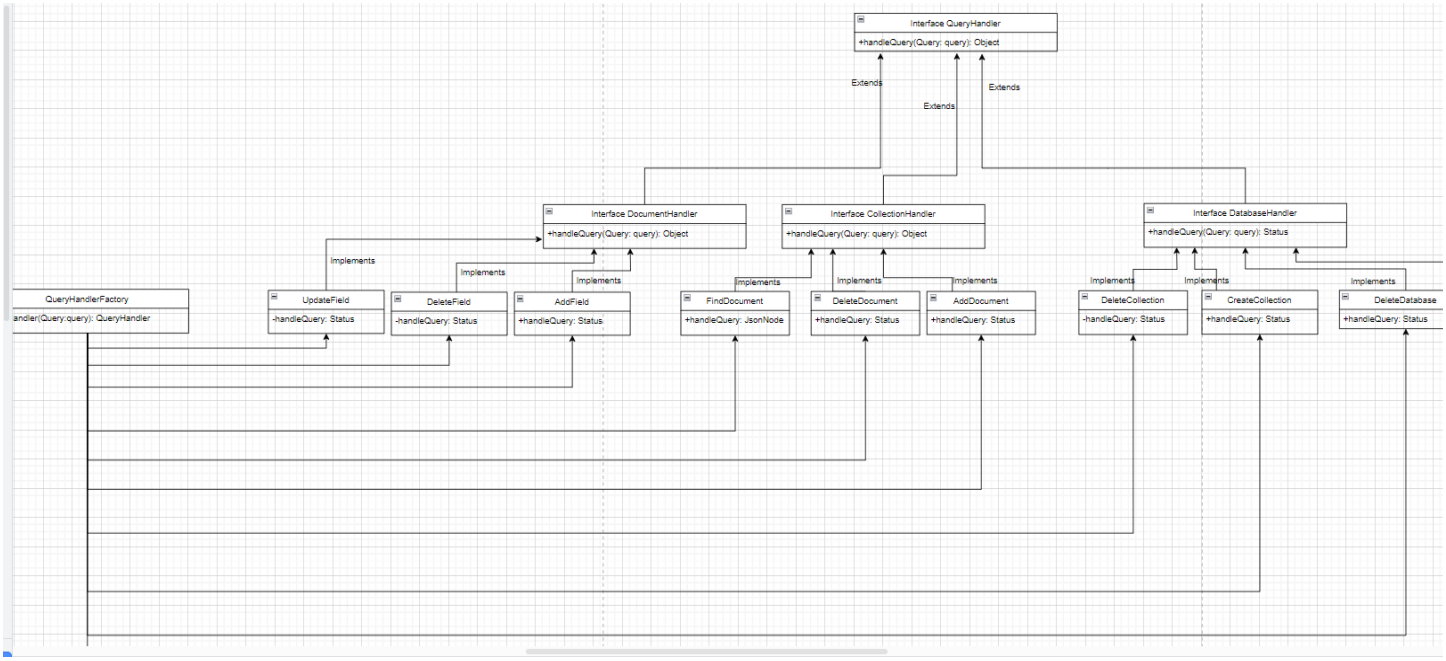
```
public Status updateField(String databaseName, String collectionName,Integer index,
String fieldName, JsonNode fieldValue){
    try {
        boolean isExecuted =
collections.search(databaseName+collectionName).updateField(index,fieldName,fieldValue);
        if(isExecuted){
            return new Status(Status.StatusType.Success, "Field updated successfully");
        }
        else{
            return new Status(Status.StatusType.Failure, "Field could not be updated");
        }
    }catch (Exception e){
        return new Status(Status.StatusType.Failure, e.getMessage());
    }
}
```

These APIs can handle the request based on the query only, so the query will include all the information required for some API to work.

Layer 2 – Query Handler

This is the second layer in the design, this layer will be responsible to handle the request and use the APIs we saw in layer 1, but query won't just affect the current node, it will also affect other nodes if it is a write query, also there will be a logging to the files which will talk about later.

So this is basically the need for the layer 2, is to do the actual query in the node, broadcast the query and do the logging, this is how I separated it so it can be extended in the future, see the figure:



As you can see, here we used the Factory method pattern, to create a Query Handler, the Factory class includes a HashMap that is used to return a query handler based in the query type, so there is always a one instance of the Query Handlers, these handlers Beans will be stored in the factory and returned so anyone can use it.

This will simplify the process of adding more types of queries in the future, all you need to do is to extend the interface and implement the handle Query method.

This is some of the code

```
public interface QueryHandler {
    Object handleQuery(Query query);
}
```

```
@Component
public class CreateCollectionHandler implements QueryHandler {
    @Autowired
    private DatabaseService databaseService;
    @Autowired
    private BroadcastSender broadcastSender;
    public Object handleQuery(Query query) {
        Status status = databaseService.createCollection(query.getDatabaseName(),
query.getCollectionName(), query.getSchema());
        log.info("User: "+query.getUsername()+" tried to create a collection
named:"+query.getDatabaseName()+"/"+query.getCollectionName()+", status:
"+status.getStatusType());
        if(!query.getBroadcastMessage() &&
status.getStatusType().equals(Status.StatusType.Success)) {
            query.setBroadcastMessage(true);
```

```
            broadcastSender.sendBroadcast(query);
        }
        return status;
    }
}
```

```
@Slf4j
@Component
public class AddDocument implements QueryHandler {
    @Autowired
    private CollectionService collectionService;
    @Autowired
    private AffinityCalculator affinityCalculator;
    @Value("${number-of-nodes}")
    private int numberOfNodes;
    @Value("${node-number}")
    private int nodeNumber;
    @Autowired
    private BroadcastSender broadcastSender;
    @Autowired
    private RequestWrite requestWrite;
    public Object handleQuery(Query query) {
        int affinityNode = affinityCalculator.calculateAffinity(query.getDocument(),
numberOfNodes);
        log.info("User: "+query.getUsername()+" tried to add a document in collection:
"+query.getDatabaseName()+"/"+query.getCollectionName());
        if (affinityNode == nodeNumber) {
            query.setBroadcastMessage(true);
            broadcastSender.sendBroadcast(query);
            return collectionService.addDocument(query.getDatabaseName(),
query.getCollectionName(), query.getDocument());
        } else if (affinityNode != nodeNumber && !query.getBroadcastMessage()) {
            return requestWrite.requestWrite(query, affinityNode);
        } else if (query.getBroadcastMessage()) {
            return collectionService.addDocument(query.getDatabaseName(),
query.getCollectionName(), query.getDocument());
        }
        return new Status(Status.StatusType.Success, "Document inserted successfully");
    }
}
```

```
@Slf4j
@Component
public class FindDocument implements QueryHandler {
    @Autowired
    private CollectionService collectionService;
    public Object handleQuery(Query query) {
        return collectionService.findDocument(query.getDatabaseName(),
query.getCollectionName(), query.getIndex());
    }
}
```

Broadcast

The broadcast is important because it will make all nodes consistent with the data entered by the clients, broadcasting messages should be concise, because instability in one node could lead to further inconsistencies in the system about the data, there are many ways to broadcast:

1- Publish-Subscribe: this way, we can publish messages to centralized place, like a server, and other nodes can grab these messages, one famous message broker is called Apache- Kafka, where there is a producer and a consumer, and there is a topic, the producer will publish to the topic, and the consumer will consume the messages from the topic, this is a good approach, but it delivers a centralization which means a single point of failure to the broadcasting system, this is not a valid option.

2- IP Multicast: this method is already implemented is protocols like UDP and TCP, but because I don't want to use sockets, I won't be able to use this method.

3- Normal Broadcasting: while this method is simple, it ensures that the messages arrive.

In the node implementation, I have two important classes to broadcast:

1- Broadcast Sender: this class is responsible for sending to all nodes a message, though it's simple, but it works.

2- Broadcast Receiver: this class will be listening to all messages from anywhere, and execute the messages, messages will be queries from other nodes, and it will get executed in the node.

About the network congestion, this should be handled, because as we have talked before, congestion would lead inconsistencies in the system data, but this requires more complex methods that we won't implement in this project.

Request Write and Handle Request Write

If some document wants to write some document, or wants to update it including (adding a field, remove a field, update a field), it will calculate the affinity node, send a request to this node, and this node will handle this request, and then if it is correct query, it will execute it, then broadcast it to other nodes, lets see these two classes:

Request Write:

```
public class RequestWrite {
    @Autowired
    private WebClient.Builder webClientBuilder;
    public Status requestWrite(Query query, int nodeNumber) {
        Status status;
        int i = 0;
        if (query.getQueryType().equals("UpdateField")) {
            do {
                status = webClientBuilder.build().post()
                        .uri("http://node-" + nodeNumber + ":8080/write/Query")
                        .bodyValue(query)
                        .exchange()
                        .flatMap(response ->
                                response.bodyToMono(Status.class)
                        ).block();
                i++;
            } while (status.getStatusType().equals(Status.StatusType.Failure) && i < 5);
        }
        else
            status = webClientBuilder.build().post()
                    .uri("http://node-" + nodeNumber + ":8080/write/Query")
                    .bodyValue(query)
                    .exchange()
                    .flatMap(response ->
                            response.bodyToMono(Status.class)
                    ).block();
        if(status.equals(Status.StatusType.Failure)){
            log.warn("Failed to write to node: "+nodeNumber);
        }
        else
            log.info("Successfully wrote to node: "+nodeNumber);
        return status;
    }
}
```

Handle Request Write:

```
@RestController
@RequestMapping("/write")
public class HandleWriteRequest {
    @Autowired
     private QueryHandlerFactory queryHandlerFactory;
    @Autowired
    OptimisticLock optimisticLock;
     @PostMapping("/Query")
     public Status receiveWriteRequest(@RequestBody Query message) {
         log.info("Received write request");
         if(message.getQueryType().equals(QueryType.UpdateField)) {
             return optimisticLock.executeWithLock(message);
         }
         else {
             return (Status)
queryHandlerFactory.getQueryHandler(message).handleQuery(message);
         }
     }
}
```

As you can see, the request write is different for the update field query, because it is handled in a different manner, it will be explained in the Race Condition section.

MULTITHREADING

Spring web apps are multithreaded by nature, but this does not mean that we didn't need to implement some multithreading.

B+ Tree

one specific need is the B+ tree, because it might be accessed by multiple threads, I have introduced to it the ReadWriteLock, and this lock is interesting because:

1- It will allow multiple threads to read simultaneously, so no blocking if there are multiple threads want to read.

2- It will block acquiring the write lock while there are threads are reading.

3- It will block reading if there is a thread still writing.

This way, we can remove or insert new data to the B tree without worrying of corrupting the data or race conditions happening, this code could illustrate how this lock works.

```
private final ReadWriteLock lock = new ReentrantReadWriteLock();
```

```
public V search(K key) {
    lock.readLock().lock();
    try {
       return root.getValue(key);
    } finally {
       lock.readLock().unlock();
    }
}
```

```
public void insert(K key, V value) {
    lock.writeLock().lock();
    try {
       root.insertValue(key, value);
    } finally {
       lock.writeLock().unlock();
    }
}
```

Multimap thread safe

What happens if two threads try to modify same property value for two different documents? See figure.

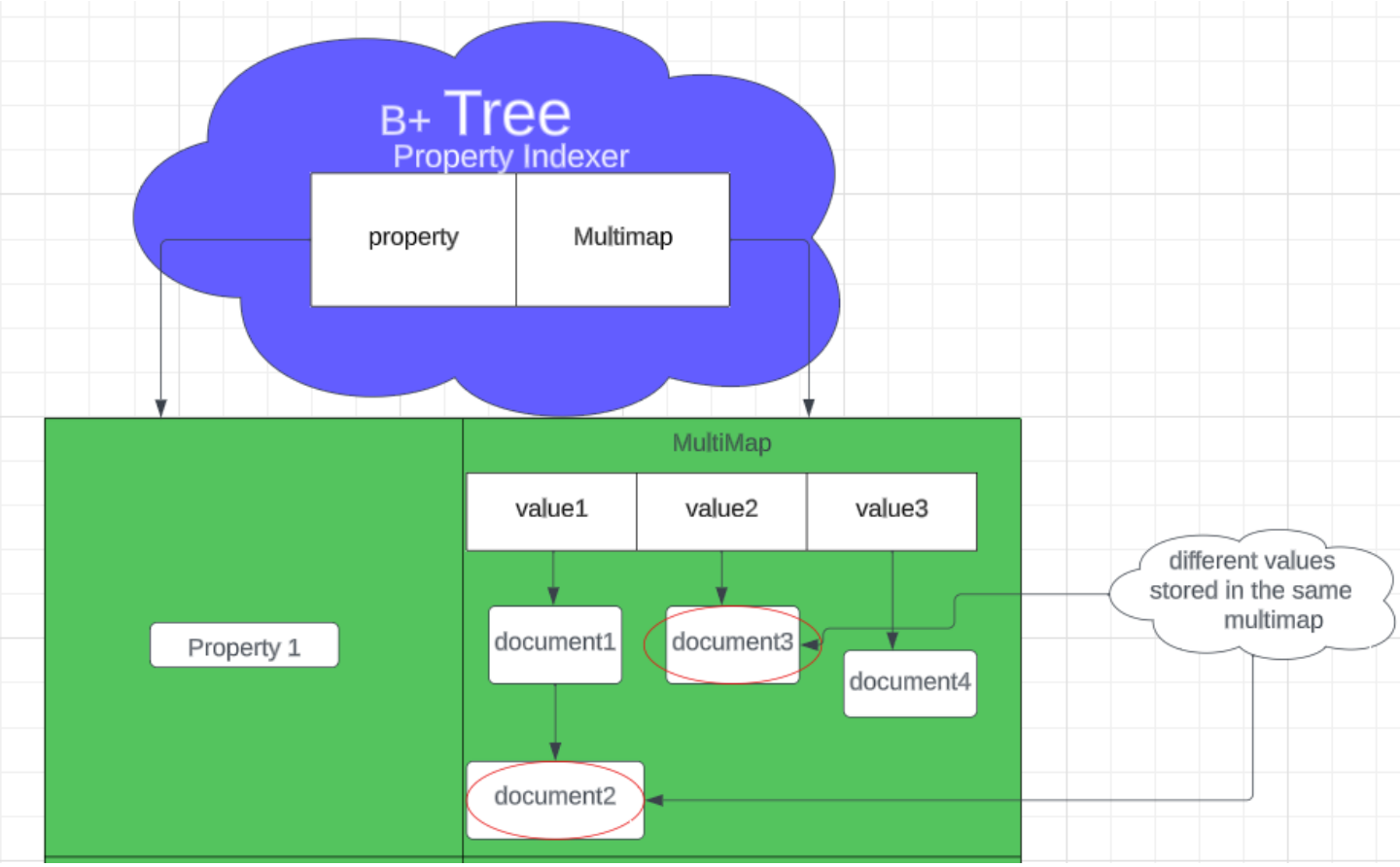


Figure 12: Multimap data Corruption.

As you can here, suppose these two values wants to change at the same time, it could result in a data corruption, so because of this, I have used the

```
Multimaps.synchronizedSetMultimap(HashMultimap.create())
```

Instead of the normal way to create a hash Multi Map, this ensures that accessing threads is safe, this provided by the Guava library.

RACE CONDITIONS

There are two ways for a race condition to happen:

1- First one is that multiple threads, ie threads want to modify same property at the same node, meaning that they are directly communicating with the affinity node, and for this I have introduced a Node Lock, see figure and code:

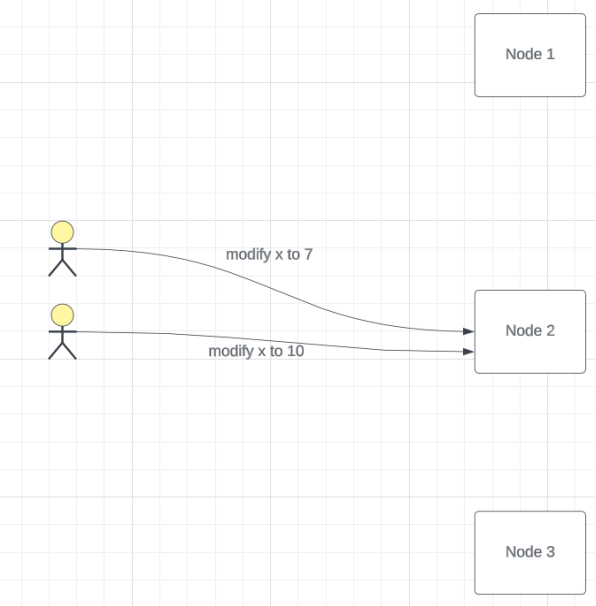


Figure 13:first Case.

```
@Component
public class NodeLock{

    @Autowired
    private BroadcastSender broadcastSender;
    @Autowired
    private DocumentService documentService;
    private final Map<String, Lock> lockMap = new HashMap<>();
    public Status executeWithLock(Query query) {
        Lock lock = getLockForString(query.getFieldName());
        lock.lock();
        try {
            query.setBroadcastMessage(true);
            broadcastSender.sendBroadcast(query);
            return documentService.updateField(query.getDatabaseName(),
query.getCollectionName(), query.getIndex(), query.getFieldName(),
query.getFieldValue());
        } catch (Exception e) {
            return new Status(Status.StatusType.Failure, "Document not updated");
        } finally {
            lock.unlock();
        }
    }
    private Lock getLockForString(String stringValue) {
        synchronized (lockMap) {
            return lockMap.computeIfAbsent(stringValue, k -> new ReentrantLock());
        }
    }

}
```

As you can see, we have a hash map that we use to store the locks, these locks will be specifically assigned to nodes, if there are not already made lock, it will create one, store it, and return it, so when other users try to access this property and update it, it will be locked until the first user finishes. This way, we can ensure that the updating for the property on the same node is thread safe.

To test this case, I have created made a delay when updating the value, so when the other request is received it will keep waiting until the last user finished updating, and it works, because the other request keeps waiting until the last update messege is done.

2- The other case is that you try modifying the property from multiple other nodes, see figure:

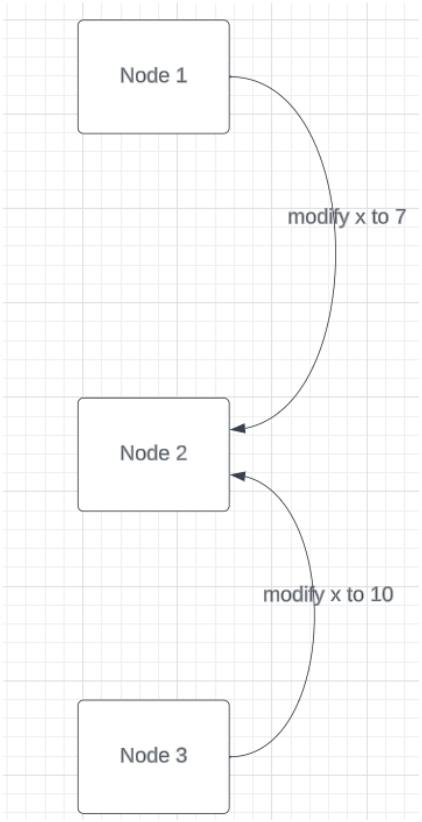


Figure 14: second case.

This can be solved by optimistic locking, Optimistic locking is about keeping versions, so for example, all properties starts with version 0, after one modification for example it will become version 1 and so on, but because it is difficult to keep versions with the Json document, I used a different approach, why not just use the value of the property itself to lock and not allow modifications for those who does not have the same value stored in that node? See the code

```
@Component
public class OptimisticLock {
    @Autowired
    private CollectionService collectionService;
    @Autowired
    private DocumentService documentService;
    @Autowired
    private BroadcastSender broadcastSender;
    public Status executeWithLock(Query query)  {
        if(query.getQueryType().equals(QueryType.UpdateField)) {
            JsonNode json = collectionService.findDocumentAsJson(query.getDatabaseName(),
query.getCollectionName(), query.getIndex());
            try {
                if
(json.get(query.getFieldName()).toString().equals(query.getOldValue().toString())) {
                    Status status =documentService.updateField(query.getDatabaseName(),
query.getCollectionName(), query.getIndex(), query.getFieldName(),
query.getFieldValue());
                    if (status.getStatusType().equals(Status.StatusType.Success) &&
!query.getBroadcastMessage()) {
                        query.setBroadcastMessage(true);
                        broadcastSender.sendBroadcast(query);
                    }
                    return status;
                } else {
                    return new Status(Status.StatusType.Failure, "Document not updated");
                }
            } catch (Exception e) {
                return new Status(Status.StatusType.Failure, "Document not updated");
            }
        }
        else{
            Status status =
collectionService.addDocument(query.getDatabaseName(),query.getCollectionName(),query.get
Document());
            if (status.getStatusType().equals(Status.StatusType.Success) &&
!query.getBroadcastMessage()) {
                query.setBroadcastMessage(true);
                broadcastSender.sendBroadcast(query);
            }
            return status;
        }
    }
}
```

As you can see, we check the value stored in the node, and compare it with the old value stored in the query, if it is the same, we allow the updating, otherwise, it will refuse it, and the sender will send 5 times to make sure its not a one time problem, then it will show the user the status whether it succeeded or not.

To test this case, I have made a different approach, I have delayed the broadcast, so the when the old value is not updated yet, the node who is requesting the update will fail because it will try 5 times and it will keep failing so it tells the user that the update is failed.

SCHEMA

So far we have seen that what makes the document-like database is special because it allow you to enter any kind of documents, this is what make it different, but what if you want to specify your schema for some collection, see figure

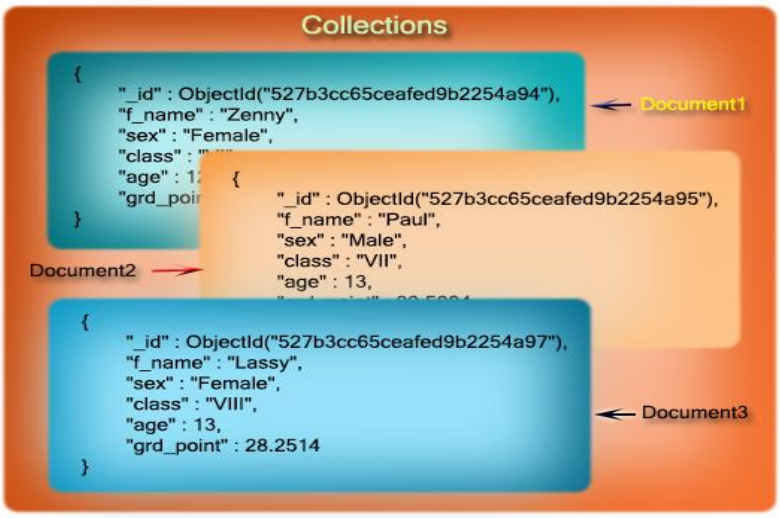


Figure 15: Collection on Schema.

As you can see, this collection has some schema, the schema mainly specifies the required properties, not only that, but it can also restrict some property value type, Object, string, integer and more, and for that, there is a library called GitHub FGE, that has a JSON Schema that you can use, see figure:

```
{
  "type": "object",
  "properties": {
    "name": {
      "type": "string"
    },
    "age": {
      "type": "integer"
    }
  },
  "required": ["name"]
}
```

As you can see, you can specify the type of document, the properties, and their types. This way, the collection will have a schema that it can use to check whether the document inserted is acceptable for this schema, but how I did that?

I have a Schema validator class that holds all the schemas for all collections, and it will be retrieving these schemas to use it so it can check and so on, see code:

```
public class SchemaValidator {
    private final Map<String,JsonNode> schemaMap;
    private static SchemaValidator schemaValidator;
    private SchemaValidator(){
        schemaMap = new HashMap<>();
    }
    public static SchemaValidator getInstance(){
        if (schemaValidator == null)
            schemaValidator = new SchemaValidator();
        return new SchemaValidator();
    }
    public void createSchema(String databaseName, String collectionName,JsonNode schema)
throws ProcessingException {
        schemaMap.put(databaseName+"/"+collectionName, schema);
    }
    public boolean isValid(String databaseName, String collectionName,JsonNode
jsonDocument) throws ProcessingException {
            if (schemaMap.containsKey(databaseName+"/"+collectionName)) {
                JsonSchemaFactory schemaFactory = JsonSchemaFactory.byDefault();
                JsonSchema jsonSchema =
schemaFactory.getJsonSchema(schemaMap.get(databaseName+"/"+collectionName));
                return jsonSchema.validInstance(jsonDocument);
            }
            else {
                return true;
            }
    }
    public JsonNode getSchema(String databaseName, String collectionName) {
        return schemaMap.get(databaseName+"/"+collectionName);
    }
}
```

as you can see, the isValid method is used to check if some document is valid for some collection.

CACHING

Using the library Google Guava, we can use some called Cache, which will be using the cache to retrieve the documents, see code:

```
@Service
@Data
public class GetJson {
    @Autowired
    private ObjectMapper objectMapper;
    private static final int MAX_CACHE_SIZE = 100; // Maximum number of JSON objects to
cache
    private Cache<String, JsonNode> cache = CacheBuilder.newBuilder()
            .maximumSize(MAX_CACHE_SIZE) // Maximum number of entries in the cache
            .build();
    public GetJson() {
        objectMapper = new ObjectMapper();
    }
    public JsonNode getJsonObjectFromFile(String filePath) throws ExecutionException {
        // Check if the JSON object is already in the cache
        return cache.get(filePath, () -> readJsonFromFile(filePath));
    }
    private JsonNode readJsonFromFile(String filePath) throws IOException {
        return objectMapper.readTree(new File(filePath));
    }
    public void cacheRefresh(String filePath,JsonNode jsonNode) throws ExecutionException
{
        cache.put(filePath, jsonNode);
    }
    public void cacheDelete(String filePath) throws ExecutionException {
        cache.invalidate(filePath);
    }
}
```

As you can see, a cache have a method called getJsonObjectFromFile which will be retrieving data from the file if it does not access in the cache, it also have a cache refresh method which will be used whenever a document is updated, this make sure that the data between files and cache is consistent, also we have a cache delete method that will be used whenever we delete a document from the files.

LOGGING

Logging is an important step to be implemented, what also makes it more important is that multiple users are actually using it, so sometimes is it important to know who did some transaction to the database.

SLF4J

SLF4J is a simple façade API interface, that does not really implement the actual logging, instead, it will be doing the binding to the behind scene library, these libraries may be Logback Log4j or something else.

It has four different log levels:

1- DEBUG

2- INFO

3- WARN

4- ERROR

These messages logging levels depends in the case, so for example if some user tried to login and failed, you can display a warn into your logs file or the server screen.

These logs will be stored in a file, and only the admin can access them.

BOOTSTRAP AND SECURITY

There are many ways for the method of communication between Bootstrap node and the cluster,

1- Issuing a token, see figure.

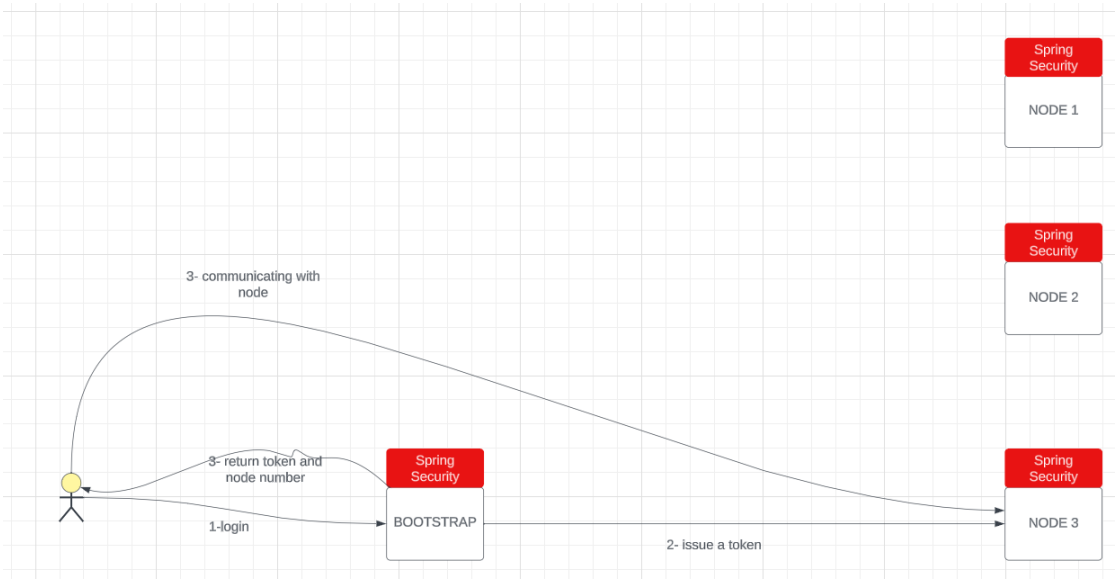


Figure 16: First method.

This method is really good and most modern websites use it, as long as you are in the website, the token will be with your requests, until the token is expired, the token can expire in mins, hours or even weeks, this way, we can make sure our cluster is secured.

In spring, you can use this method by using the JWT, and I implemented it, but some problem was noticed in the system, and I couldn't solve it because of lake of time.

2- Using Basic Auth for both Bootstrap and Nodes, this way, you can secure your endpoints in both.

We have two scenarios for a client to deal with the server:

1- Login: the user might want to login, and this what will happen, see figure

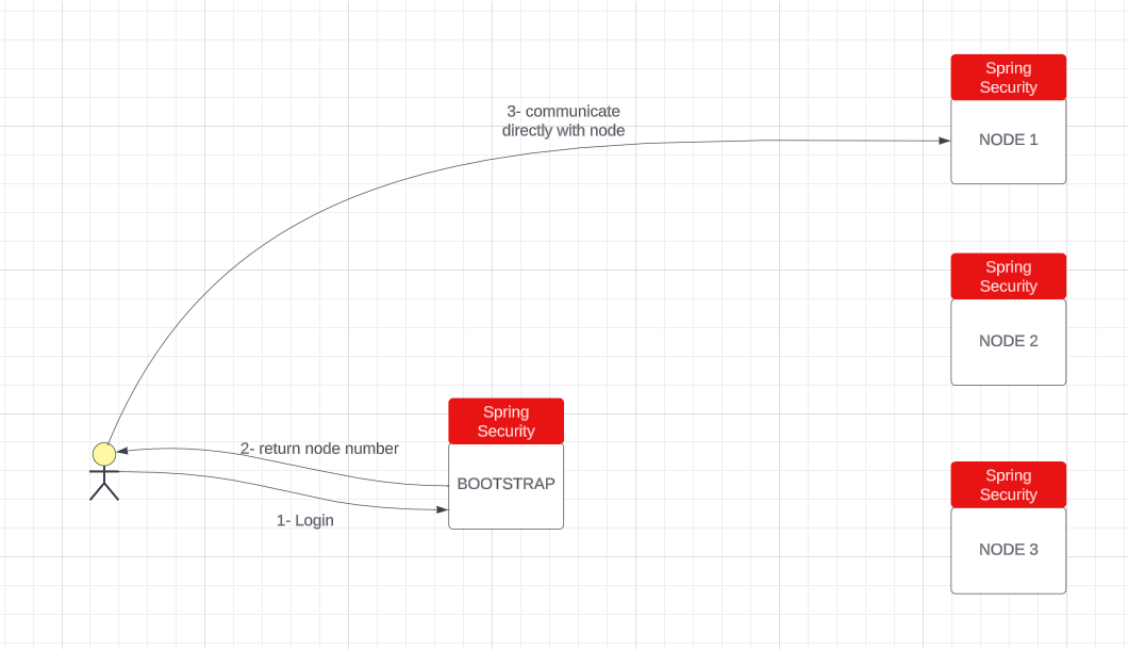


Figure 17:second method to secure the app.

The client will first communicate with the bootstrap to login, if he's authenticated, he will be given a node address to communicate to, then he will be directly communicating with that node, and if the user tried to communicate to the node directly without providing credentials, he will not be able to get into the system because the endpoints are secured.

2- Signup: the user might want to sign up to the system, see figure:

Figure 18: sign up.

As you can see, the first step that the client will first request to sign up, then the bootstrap will check if he's providing a valid credentials, if so the bootstrap will be communicating with the node to register the user, note that the process of registering these users are load-balanced, in round robin fashion, then the bootstrap will return to the user the node address that he is registered to, and he can now communicate directly with the node.

Spring Security

Spring security plays a significant role in securing our system, I will be showing the security filter chain, which will be receiving the requests before the controllers and check whether the user is authenticated to access the endpoint.

1- Bootstrap:

```
@Bean
public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
    http
            .cors().disable()
            .csrf().disable()
            .authorizeHttpRequests( auth -> auth
                    .requestMatchers(HttpMethod.POST,"/signup").permitAll()
                    .requestMatchers("/returnNodeAddress").hasAnyRole( ...roles: "ADMIN", "USER")
                    .anyRequest().denyAll()
            )
            .httpBasic(Customizer.withDefaults())
    ;
    return http.build();
}
```

As you can see, we have two endpoints, first one is sign up, which anyone can access, and the other one is return Node address which can be accessed by registered users only.

2- Worker (Node):

```
@Configuration
@EnableWebSecurity
public class SecurityConfig {

    @Bean
    public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
        http
                .cors().disable()
                .csrf().disable()
                .authorizeHttpRequests( auth -> auth
                        .requestMatchers("/RequestLogging").hasRole("ADMIN")
                        .requestMatchers("/register/user").hasRole("BOOTSTRAP")
                        .requestMatchers("/database/getdatabases").hasAnyRole( ...roles: "ADMIN", "USER")
                        .requestMatchers("/database/getcollections/**").hasAnyRole( ...roles: "ADMIN", "USER")
                        .requestMatchers("/database/**").hasAnyRole( ...roles: "ADMIN")
                        .requestMatchers("/collection/**").hasAnyRole( ...roles: "ADMIN", "USER")
                        .requestMatchers("/document/***").hasAnyRole( ...roles: "ADMIN", "USER")
                        .requestMatchers("/api/auth/**").hasAnyRole( ...roles: "ADMIN", "USER")
                        .requestMatchers("/write/Query").permitAll()
                        .requestMatchers("/broadcast/Query").permitAll()
                        .anyRequest().permitAll()
                        //hasAuthority("SCOPE_READ")
                )
                .httpBasic(Customizer.withDefaults())
        ;
        return http.build();
    }
```

Notices are:

- hasRole() indicates that the user who requestsed this endpoint should have authority of some type.
- hasAnyRole indicates that user should have any role of authority.
- Notice that the register/user endpoint is only accessible to BOOTSTRAP.

BCRYPT

1. **Hashing Algorithm**: Bcrypt is a one-way cryptographic hashing algorithm. It takes an input (typically a password) and converts it into a fixed-length string of characters, known as a hash. The key feature of bcrypt is that it is intentionally slow, making it resistant to brute force and dictionary attacks.
2. **Salting**: Bcrypt automatically handles the generation of a random salt (a random value) for each password hash. Salting is crucial for preventing attackers from using precomputed tables (rainbow tables) to crack passwords. The salt is typically included as part of the final hash.
3. **Work Factor**: Bcrypt allows for a work factor (sometimes called the cost factor) to be specified when hashing a password. This work factor determines the number of iterations the algorithm should perform, making it computationally expensive and slowing down potential attacks. A higher work factor results in a more secure hash but also requires more processing power to compute.
4. **Resistance to Speed Attacks**: Bcrypt is designed to be slow intentionally, which helps protect against various speed-based attacks, including brute force and dictionary attacks. As hardware and computing power advance, the work factor can be increased to maintain a consistent level of security.
5. **Security**: Bcrypt is considered a secure choice for password hashing, and it has withstood extensive security analysis over the years. It is widely recommended for securely storing passwords in databases and is used in various programming languages and frameworks.

I used the Bcrypt for both in the worker nodes and the Bootstrap node, this way, we can make sure our databases details are secured.

ROLES

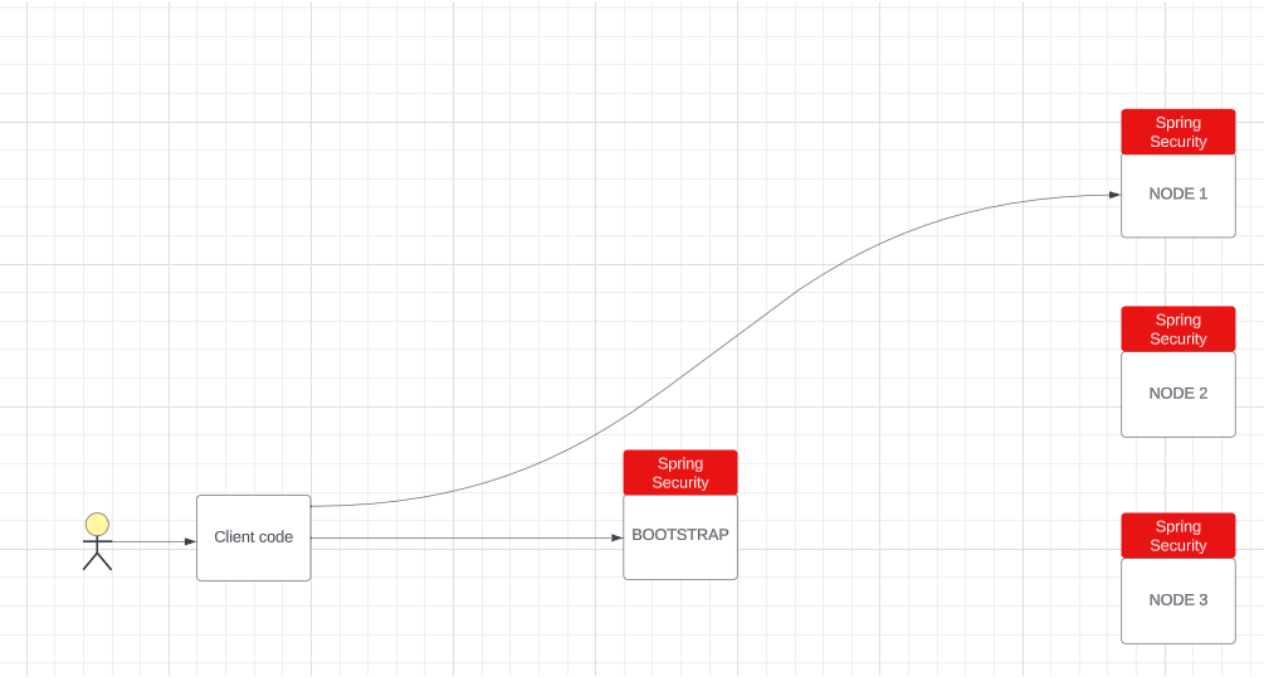
There are two roles in the system:

1- Admin: admin can add databases and delete them, and he can also request the logs, all other he can do, including collection commands, and documents commands.

2- User: user can do the collection commands and the document commands, but he can't create databases or delete them, he will get UNAUTHRIZED if he tries to do so.

SHELL

Because it's difficult to memorize all the endpoints of the system, it is good practice to use a shell that we can use to send requests to the system, see figure:



As you can see, the client code could help and make the system more friendly, but how we can make that client code?

Spring Shell

Spring shell is another excellent framework that we can use as a command line app, it allow users to enter normal commands and transform these commands to http requests, receive responses and show them to the user, see an example:

```
@ShellComponent("Login and Register Commands")
public class LoginAndRegisterCommands extends AbstractShellComponent {
    @Autowired
    private User user;
    @Autowired
    private WebClient.Builder authBuilder;


    @ShellMethod("Contact Bootstrap to login")
    public ObjectNode login() {
        System.out.println("Enter username: ");
        Scanner sc = new Scanner(System.in);
        String username = sc.nextLine();
        System.out.println("Enter password: ");
        String password = sc.nextLine();
        WebClient webClient = authBuilder.baseUrl("http://localhost:8000")
                .defaultHeader(HttpHeaders.AUTHORIZATION, ...values: "Basic yourBase64EncodedCredentials")
                .filter(ExchangeFilterFunctions.basicAuthentication(username, password))
                .build();

        ObjectNode responseMono = webClient.get() RequestHeadersUriSpec<capture of ?>
                .uri( uri: "/returnNodeNumber") capture of ?
                .retrieve() ResponseSpec
                .bodyToMono(ObjectNode.class).block();
        ObjectNode objectNode = new ObjectMapper().createObjectNode();
        if(responseMono.has( fieldName: "nodeNumber")){
            this.user.setUsername( username);
            this.user.setPassword(password);
            this.user.setNode_number(responseMono.get("nodeNumber").asInt());
            return objectNode.put( fieldName: "statusType", v: "Success");
        }
        else
            return objectNode.put( fieldName: "statusType", v: "Failed");
```

As you can see, this command is called login, it will take username and password, login the users and save his data and use them in further requests, bean of type user will be around the app so we can use it from anywhere in the app, see the list of commands that the user can use:

```
Collection Commands
       create-collection: Create a Collection
       delete-collection: Delete a Collection
       get-collections: Get all collections

Database Commands
       get-databases: Get all databases
       delete-database: Delete a database
       create-database: Create a database
       get-logs: See Logs

Document Commands
       get-documents: Get all documents
       update-field: Update a field in a document
       add-field: Add a field to a document
       delete-field: Delete a field from a document
       delete-document: Delete a document
       process-json: Process JSON input
       add-document: Add a document
       get-document-by-field: Get a document by field

Login And Register Commands
       login: Contact Bootstrap to login
       sign-up: Sign up, user mode only
```

I will show other commands methods now, method arguments means that these arguments are required to run this command

```java
@ShellMethod("Create a database")
public ObjectNode createDatabase(String dbName) {
    WebClient webClient = WebClient.builder().baseUrl("http://localhost:808"+user.getNode_number()+"/database/create/"+dbName)
            .filter(ExchangeFilterFunctions.basicAuthentication(user.getUsername(), user.getPassword()))
            .build();
    System.out.println(webClient.head());
    Mono<ObjectNode> responseMono = webClient
            .get() RequestHeadersUriSpec<capture of ?>
            .retrieve() ResponseSpec
            .bodyToMono(ObjectNode.class);

    return responseMono.block();
}
```

```java
@ShellMethod("Add a document" )
public ObjectNode addDocument(String dbName, String collectionName, String document) {
    WebClient webClient = WebClient.builder()
            .baseUrl("http://localhost:808" + user.getNode_number() + "/collection/create/" + dbName + "/" + collectionName)
            .filter(ExchangeFilterFunctions.basicAuthentication(user.getUsername(), user.getPassword()))
            .build();
    ObjectMapper objectMapper = new ObjectMapper();
    JsonNode documentNode;
    try {
        documentNode = objectMapper.readTree(document);
        System.out.println(documentNode.toString()+" "+documentNode.isObject()+"--->");
    } catch (JsonProcessingException e) {
        System.out.println("Error parsing JSON: " + e.getMessage());
        e.printStackTrace();
        return (ObjectNode) objectMapper.nullNode();
    }
    Mono<ObjectNode> responseMono = webClient
            .post() RequestBodyUriSpec
            .contentType(MediaType.APPLICATION_JSON) RequestBodySpec
            .bodyValue(documentNode) RequestHeadersSpec<capture of ?>
            .retrieve() ResponseSpec
            .bodyToMono(ObjectNode.class);

    return responseMono.block();
```

TESTING

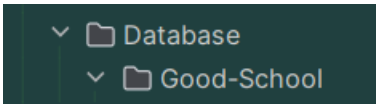
To test our database, I want to test all the functionalities of the database, I will be creating a database called Good-School, it includes collections, each collection represents a class, and we have 3 collections to indicate 3 classes, let's see:

1- Login: logging in is passed, see test:

```
shell:>login
Enter username:
admin
Enter password:
admin
{"statusType":"Success"}
shell:>
```

2- Creating Database: successfully passed, see test and file system:

```
shell:>create-database Good-School
{"statusType":"Success","message":"Database created successfully"}
shell:>
```



3- Creating collection: I created a collection with a simple schema called Math to make sure that the schema works, so let's see:

```
shell:>create-collection Good-School Math
Do you want to include a schema? (y/n)
y
Enter the schema:
{
    "type": "object",
    "properties": {
        "_id": {
            "type": "integer"
        },
        "name": {
            "type": "string"
        },
        "mathScore": {
            "type": "number",
            "minimum": 0,
            "maximum": 100
        },
        "attendance": {
            "type": "boolean"
        }
    },
    "required": ["_id", "name", "mathScore"]
}

{"statusType":"Success","message":"Collection created successfully"}
shell:>
```

4- Now after we have created a collection, we want to test two cases:

- First case that would it allow adding documents if they are applicable to the schema, see case

```
shell:>add-document Good-School Math '{"_id":1, "name":"Hassan", "mathScore":50}'
add-document Good-School Math '{"_id":1, "name":"Hassan", "mathScore":50}{"statusType":"Success","message":"Document added successfully"}
shell:>
```

As you can see, it works, but lets try to violate the schema in two ways, first is that we violate the score, and second is that we don't include the name:

```
shell:>add-document Good-School Math '{"_id":2, "name":"Moh", "mathScore":110}'
add-document Good-School Math '{"_id":2, "name":"Moh", "mathScore":110}{"statusType":"Failure","message":"Schema Violation"}
shell:>add-document Good-School Math '{"_id":2, "mathScore":10}'
add-document Good-School Math '{"_id":2, "mathScore":10}{"statusType":"Failure","message":"Schema Violation"}
shell:>
```

As you can see, the schema is violated and the document is not inserted, but what if someone wants to know the schema?

5- Getting the Schema: If the user wants to know the schema, he will do this:

```
shell:>get-schema Good-School Math
{
  "type" : "object",
  "properties" : {
    "_id" : {
      "type" : "integer"
    },
    "name" : {
      "type" : "string"
    },
    "mathScore" : {
      "type" : "number",
      "minimum" : 0,
      "maximum" : 100
    },
    "attendance" : {
      "type" : "boolean"
    }
  },
  "required" : [ "_id", "name", "mathScore" ]
}
shell:>
```

6- Getting a document by id:

```
shell:>get-document-by-index Good-School Math 1
{
  "_id" : 1,
  "name" : "Hassan",
  "mathScore" : 50
}
shell:>
```

7- Getting a document by some field:

```
shell:>get-document-by-field Good-School Math mathScore 50
[
{
  "_id" : 1,
  "name" : "Hassan",
  "mathScore" : 50
}
]
shell:>
```

This can be useful because suppose you want to get all students who got a grade of 50.

8- Add a field to a document:

```
shell:>add-field Good-School Math 1 messege Hello
{"statusType":"Success","message":"Field added successfully"}
shell:>get-document-by-index Good-School Math 1
{
  "_id" : 1,
  "name" : "Hassan",
  "mathScore" : 50,
  "messege" : "Hello"
}
shell:>
```

9- Delete a field from a document:

```
shell:>delete-field Good-School Math 1 messege
{"statusType":"Success","message":"Field deleted successfully"}
shell:>get-document-by-index Good-School Math 1
{
  "_id" : 1,
  "name" : "Hassan",
  "mathScore" : 50
}
shell:>
```

10- Update Field:

```
shell:>update-field Good-School Math 1 mathScore 90
{"statusType":"Success","message":"Field updated successfully"}
shell:>get-document-by-index Good-School Math 1
{
  "_id" : 1,
  "name" : "Hassan",
  "mathScore" : 90
}
shell:>
```

11- Delete a collection:

```
shell:>delete-collection Good-School Math
{"statusType":"Success","message":"Collection deleted successfully"}
shell:>get-collections Good-School
{"statusType":"Success","message":""}
shell:>
```

12- Delete a database:

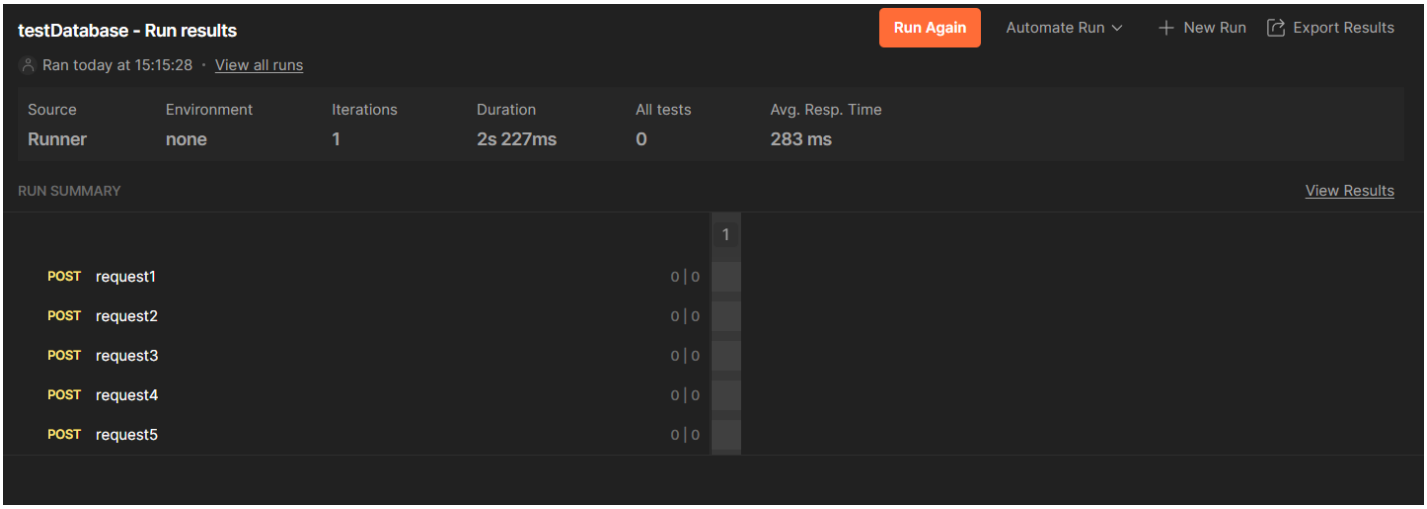
```
shell:>delete-database Good-School
{"statusType":"Success","message":"Database deleted successfully"}
shell:>get-databases
{"statusType":"Success","message":""}
shell:>
```

Notice that the documents, collections and databases are the same across all the nodes, this can be tested easily by logging in with other users, because users are in different nodes, I have used the admin account to create the data, and I have used the user account to read the data that the admin wrote, this way, we can make sure that the nodes are consistent with the data provided.

Load Balanced

To test that the load is balanced across the nodes, I will show two ways:

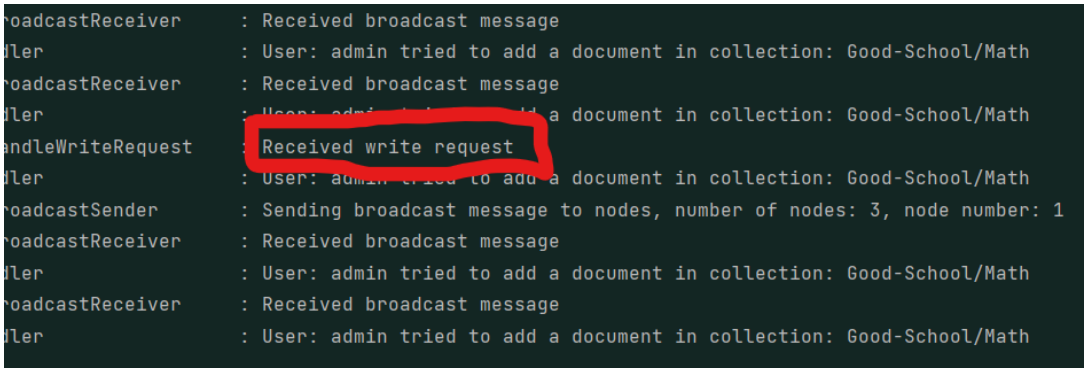
- The writing of the documents is load balanced, I have tested this with Postman, I have created many documents using requests, and these requests sent at the same time, lets see:

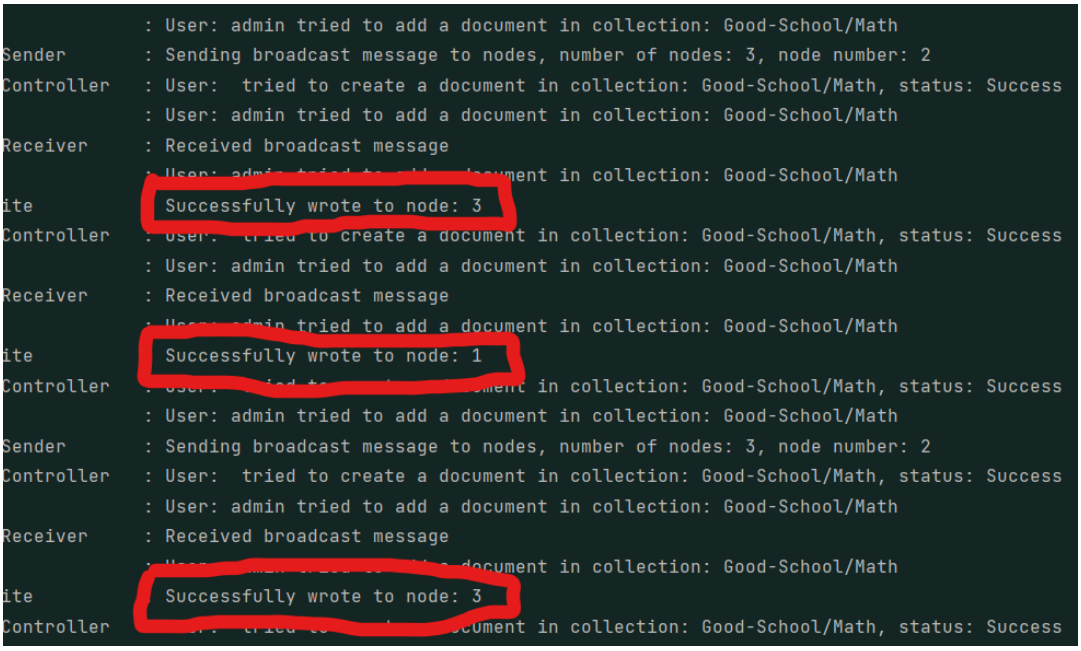


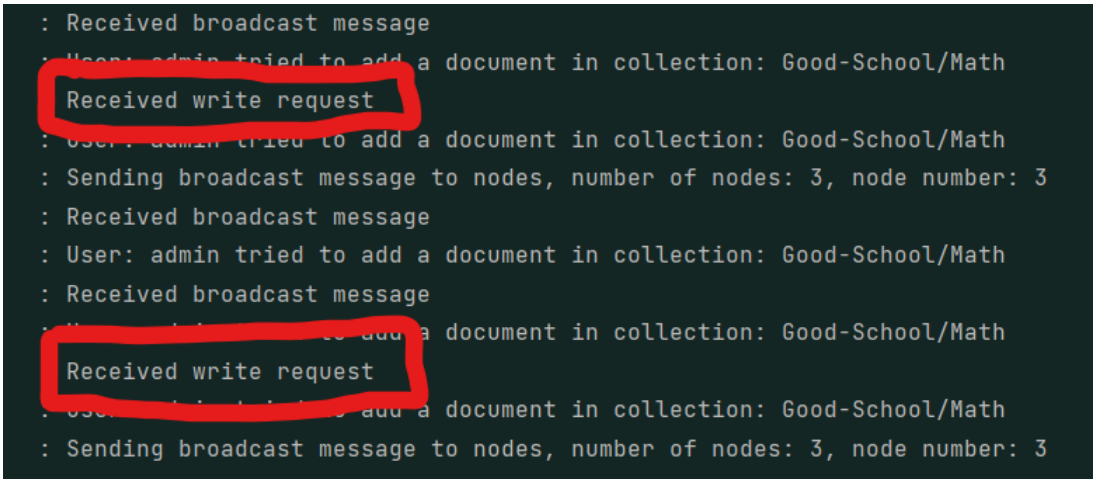

```
shell:>get-documents Good-School Math
{
  "_id" : 5,
  "name" : "Malek",
  "mathScore" : 99
}
{
  "_id" : 4,
  "name" : "Fadi",
  "mathScore" : 90
}
{
  "_id" : 3,
  "name" : "Moh",
  "mathScore" : 10
}
{
  "_id" : 2,
  "name" : "Salem",
  "mathScore" : 40
}
{
  "_id" : 1,
  "name" : "Hassan",
  "mathScore" : 30
}

shell:>
```

These requests are arrived as you see, and we want to check now how these requests is handled in the nodes,

I have 3 nodes, and I want to show their logs:







The round robin method works in this way: it starts from node-2, then node-3, then node-1, if we have 5 documents, and we start from node-2, it will be like this node-2→ node-3→node-1→node-2→ node-3, and as you can see, the node-3 received two write requests, while node 1 received one, and node-2 executed 2 queries, broadcasting is done for the node that writes, and then it broadcasts.

Notice how there are 5 requests, and 5 messages in each node, it might be, RECEIVED BROADCAST, RECEVED WRITE REQUEST, or if the node itself is writing it will announce that it is broadcasting.

Unit Testing

I wrote some tests using JUNIT 5, notice that the tests are from the client side, the green mark in the test method indicates that the test is passed

```
@Test
void loginTest() {
    String input = "admin\nadmin\n";
    testIn = new ByteArrayInputStream(input.getBytes());
    System.setIn(testIn);
    ObjectNode mono = loginAndRegisterCommands.login();
    System.setIn(originalIn);
    assertEquals( expected: "Success", mono.get("statusType").asText());
}
@Test
void signUpTest() {
    String responseEntity = loginAndRegisterCommands.signUp( username: "hassan",  password: "123",  email: "hassan123@gmail.com");
    assertEquals( expected: "Successfully Registered, now you need to login", responseEntity);
}
@Test
void createDatabaseTest() {
    String input = "admin\nadmin\n";
    testIn = new ByteArrayInputStream(input.getBytes());
    System.setIn(testIn);
    loginAndRegisterCommands.login();
    ObjectNode objectNode = databaseCommands.createDatabase( dbName: "test1");
    StepVerifier.create(Mono.just(objectNode)) FirstStep<ObjectNode>
            .expectNextMatches(objectNode1 -> objectNode1.get("statusType").asText().equals("Success")) Step<ObjectNode>
            .verifyComplete();
}
```

```java
    @Test
    void createCollectionTest() throws JsonProcessingException {
        String input = "admin\nadmin\n";
        testIn = new ByteArrayInputStream(input.getBytes());
        System.setIn(testIn);
        ObjectNode mono = loginAndRegisterCommands.login();
        System.setIn(originalIn);
        input = "n\n";
        testIn = new ByteArrayInputStream(input.getBytes());
        System.setIn(testIn);
        ObjectNode objectNode = collectionCommands.createCollection( dbName: "test1",  collectionName: "test");
        StepVerifier.create(Mono.just(objectNode)) FirstStep<ObjectNode>
                .expectNextMatches(objectNode1 -> objectNode1.get("statusType").asText().equals("Success")) Step<ObjectNode>
                .verifyComplete();
    }
    @Test
    void addDocumentTest() {
        String input = "admin\nadmin\n";
        testIn = new ByteArrayInputStream(input.getBytes());
        System.setIn(testIn);
        ObjectNode mono = loginAndRegisterCommands.login();
        System.setIn(originalIn);
        ObjectNode objectNode = documentCommands.addDocument( dbName: "test1",  collectionName: "test",  document: "{\"_id\":1,\"namee\"
        StepVerifier.create(Mono.just(objectNode)) FirstStep<ObjectNode>
                .expectNextMatches(objectNode1 -> objectNode1.get("statusType").asText().equals("Success")) Step<ObjectNode>
                .verifyComplete();
    }
```

```java
    @Test                                                                                    ⚠ 12  ✓ 3
    void findDocumentTest() {
        String input = "admin\nadmin\n";
        testIn = new ByteArrayInputStream(input.getBytes());
        System.setIn(testIn);
        ObjectNode mono = loginAndRegisterCommands.login();
        System.setIn(originalIn);
        List<JsonNode> objectNode = documentCommands.getDocumentByField( dbName: "test1",  collectionName: "test",  field: "namee",  valu
        assertEquals(objectNode.get(0).toString(),  actual: "{\"_id\":1,\"namee\":\"hassan\"}");
    }
    @Test
    void deleteCollectionTest() {
        String input = "admin\nadmin\n";
        testIn = new ByteArrayInputStream(input.getBytes());
        System.setIn(testIn);
        ObjectNode mono = loginAndRegisterCommands.login();
        System.setIn(originalIn);
        ObjectNode objectNode = collectionCommands.deleteCollection( dbName: "test1",  collectionName: "test");
        StepVerifier.create(Mono.just(objectNode)) FirstStep<ObjectNode>
                .expectNextMatches(objectNode1 -> objectNode1.get("statusType").asText().equals("Success")) Step<ObjectNode>
                .verifyComplete();
    }
    @Test
    void deleteDatabaseTest() {
        String input = "admin\nadmin\n";
        testIn = new ByteArrayInputStream(input.getBytes());
        System.setIn(testIn);
        ObjectNode mono = loginAndRegisterCommands.login();
        System.setIn(originalIn);
        ObjectNode objectNode = databaseCommands.deleteDatabase( dbName: "test1");
```

CLEAN CODE

Design rules

1. Keep configurable data at high levels: in my design, I have separated configurations and higher level of communication and logging, layers of database design.
2. Prefer polymorphism to if/else or switch/case: in the Query handler, I could handle the queries just by if else or switch statements, but instead, I used Polymorphism to create types of Quary Handlers.
3. Use dependency injection: as we have seen earlier, we have relied in spring to inject beans to our methods or constructors.
4. Follow Law of Demeter. A class should know only its direct dependencies: all classes just know the needed information for its methods.

Understandability

1. Be consistent. If you do something a certain way, do all similar things in the same way: code is consistent as possible.
2. Use explanatory variables: almost all variables are well-named variables.
3. Encapsulate boundary conditions. Boundary conditions are hard to keep track of. Put the processing for them in one place: all the boundary conditions are placed in one place.
4. Prefer dedicated value objects to primitive type: we used JsonNode objects to deal with values instead of dealing with values like strings or integer, sometimes we dealt with Integer and StringBuilder wrapper objects.
5. Avoid negative conditionals: negative conditional statements are avoided.

Names rules

1- Choose descriptive and unambiguous names.

2- Make meaningful distinction.

3- Used pronounceable names.

Functions

1- Small

2- Most functions do one thing.

3- Prefer fewer arguments: most functions take one or two arguments.

Objects and data structures

1- Hide internal structure: data structures implementations are hidden, like the b+ tree.

2- Prefer non-static methods to static methods: there are no static methods in my code because of the good use of beans in spring.

SOLID PRINCIPLES

Single Responsibility

First principle indicates that a class should have one responsibility or a one reason to change, this means that the class should do one thing in general, and that is clear in my code, see examples:

```java
@Service
@Slf4j
public class BroadcastSender {
    @Autowired
    private WebClient.Builder webClientBuilder;

    @Value("${NUMBER_OF_NODES}")
    private int numberOfNodes;
    @Value("${NODE_NUMBER}")
    private int nodeNumber;
    public void sendBroadcast(Query query) {
      log.info("Sending broadcast message to nodes, number of nodes: " + numberOfNodes + ", node number: " + nodeNumber);
        for (int i = 1; i <= numberOfNodes; i++) {
            if(i==nodeNumber) {
                continue;
            }
            log.info("sending to: "+"node-" + i );
            webClientBuilder.build().post() RequestBodyUriSpec
                    .uri( uri: "http://node-" + i + ":8080/broadcast/Query") RequestBodySpec
                    .contentType(MediaType.APPLICATION_JSON)
                    .body(Mono.just(query), Query.class) RequestHeadersSpec<capture of ?>
                    .exchange() Mono<ClientResponse>
                    .block();
        }
    }
}
```


```java
@Service
@Slf4j
public class BroadcastSender {
    @Autowired
    private WebClient.Builder webClientBuilder;

    @Value("${NUMBER_OF_NODES}")
    private int numberOfNodes;
    @Value("${NODE_NUMBER}")
    private int nodeNumber;
    public void sendBroadcast(Query query) {
      log.info("Sending broadcast message to nodes, number of nodes: " + numberOfNodes + ", node number: " + nodeNumber);
        for (int i = 1; i <= numberOfNodes; i++) {
            if(i==nodeNumber) {
                continue;
            }
            log.info("sending to: "+"node-" + i );
            webClientBuilder.build().post() RequestBodyUriSpec
                    .uri( uri: "http://node-" + i + ":8080/broadcast/Query") RequestBodySpec
                    .contentType(MediaType.APPLICATION_JSON)
                    .body(Mono.just(query), Query.class) RequestHeadersSpec<capture of ?>
                    .exchange() Mono<ClientResponse>
                    .block();
        }
    }
}
```

```java
public class RequestWrite {
    @Autowired
    private WebClient.Builder webClientBuilder;
    public Status requestWrite(Query query, int nodeNumber) {
        Status status;
        int i = 0;
        if (query.getQueryType().equals("UpdateField")) {
            do {
                status = webClientBuilder.build().post() RequestBodyUriSpec
                        .uri( uri: "http://node-" + nodeNumber + ":8080/write/Query") RequestBodySpec
                        .bodyValue(query) RequestHeadersSpec<capture of ?>
                        .exchange() Mono<ClientResponse>
                        .flatMap(response ->
                                response.bodyToMono(Status.class)
                        ).block();
                i++;
            } while (status.getStatusType().equals(Status.StatusType.Failure) && i < 5);
        }
        else
            status = webClientBuilder.build().post() RequestBodyUriSpec
                    .uri( uri: "http://node-" + nodeNumber + ":8080/write/Query") RequestBodySpec
                    .bodyValue(query) RequestHeadersSpec<capture of ?>
                    .exchange() Mono<ClientResponse>
                    .flatMap(response ->
                            response.bodyToMono(Status.class)
                    ).block();
        if(status.equals(Status.StatusType.Failure)){
            log.warn("Failed to write to node: "+nodeNumber);
        }
        else
            log.info("Successfully wrote to node: "+nodeNumber);
        return status;
    }
```

Open for Extension, Closed for Modification

This shows in my code in the Query Handler, I have an interface that anyone can implement, then he will need to implement the method handleQuery, the QueryHandlerFactory map will be updated

```java
@Slf4j
@Component
public class CreateDatabaseHandler implements QueryHandler {
    @Autowired
    private DatabaseService databaseService;
    @Autowired
    private BroadcastSender broadcastSender;
    public Object handleQuery(Query query) {
        if(!query.getBroadcastMessage()) {
            query.setBroadcastMessage(true);
            broadcastSender.sendBroadcast(query);
        }
        Status status = databaseService.createDatabase(query.getDatabaseName());
        log.info("User: "+query.getUsername()+" tried to create a database named:"+query.getDatabaseName()+", status: "+status
        return status;
    }
}
```

```java
@Service
public class QueryHandlerFactory {
    private HashMap<QueryType, QueryHandler> queryHandlerMap;

    @Autowired
    public QueryHandlerFactory(
            GetDatabasesHandler getDatabasesHandler,
            GetCollectionsHandler getCollectionsHandler,
            CreateDatabaseHandler createDatabaseHandler,
            CreateCollectionHandler createCollectionHandler,
            DeleteDatabaseHandler deleteDatabaseHandler,
            DeleteCollectionHandler deleteCollectionHandler,
            AddDocumentHandler addDocumentHandler,
            DeleteDocumentHandler deleteDocumentHandler,
            FindDocumentsHandler findDocumentsHandler,
            GetDocumentsByFieldHandler getDocumentsByFieldHandler,
            FindDocumentHandler findDocumentHandler,
            AddFieldHandler addFieldHandler,
            DeleteFieldHandler deleteFieldHandler,
            UpdateFieldHandler updateFieldHandler
    ) {
        queryHandlerMap = new HashMap<>();
        queryHandlerMap.put(QueryType.GetDatabases, getDatabasesHandler);
        queryHandlerMap.put(QueryType.GetCollections, getCollectionsHandler);
        queryHandlerMap.put(QueryType.CreateDatabase, createDatabaseHandler);
        queryHandlerMap.put(QueryType.CreateCollection, createCollectionHandler);
        queryHandlerMap.put(QueryType.DeleteDatabase, deleteDatabaseHandler);
        queryHandlerMap.put(QueryType.DeleteCollection, deleteCollectionHandler);

        queryHandlerMap.put(QueryType.AddDocument, addDocumentHandler);
        queryHandlerMap.put(QueryType.DeleteDocument, deleteDocumentHandler);
```

As you can see, he will just add to this map, this method takes many arguments unfortunately, but I had to do this so the spring context can do the injection and inject the beans into the object.

Liskov

This is clear in the QueryHandler implementation by many subclasses, there are no violations for this principle.

Interface Segregation

This also shows in the QueryHandler and how we separated the interface into sub interfaces.

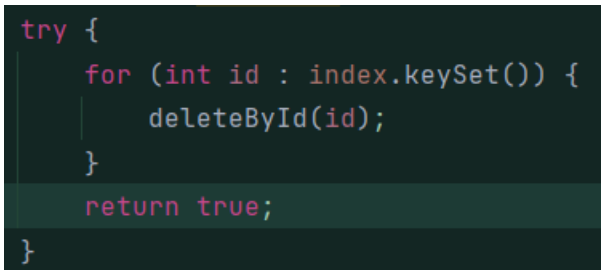
Dependency Inversion

We have seen and talked about Dependency injection and how the spring context controls the beans and injections.

EFFECTIVE JAVA

- Use BUILDERS when faced with many constructors(Item 2): I used builder to create Queries instead of many un-necessary constructors.
- Enforce the singleton property with a private constructor or an enum type(Item 3): I used the Singleton property for SchemaValidator, and other classes are components in my system, so they are singleton by nature, because the spring context will hold one bean of that component.
- Enforce noninstantiability with a private constructor(Item 4): this is applied in the Schema Validator.
- Prefer dependency injection to hardwiring resources(Item 5): this is applied everywhere in the code because I'm using Spring.
- Avoid creating unnecessary objects(Item 6): every object that is created in my code is being used.
- Minimize the accessibility of classes and members(Item 15): members in my classes are private.
- In public classes, use accessor methods, not public fields(Item 13): I used accessor methods (getters) and mutators (setters).
- In public classes, use accessor methods, not public fields(Item 16): all classes have accessors.
- Minimize mutability(Item 17): Every filed in my classes that can be final is final.
- Favor composition over inheritance(Item 18): I used this idea a lot in my classes instead of inheritance, because the inheritance violates the encapsulation in some way.
-

- Prefer interfaces to abstract classes(Item 20): because Java allow single inheritance this applies to abstract classes, so the best way is to use interfaces if you have many implementations, and I avoided using abstract classes.
- Use interfaces only to define types(Item 19): I have used interface to define a type which is a QueryHandler.
- Prefer class hierarchies to tagged classes(Item 20): tagged classes means that classes that have many uses, which clearly violates the single responsibility principle.
- Prefer lists to arrays(Item 28): lists are more comfortable to work with because they are variable sized.
- Consistently use the Override annotation: while this can work without the annotation, but it's important so anyone who are reading the code will be able to read the code.
- Check parameters for validity: in the Collection indexer, when adding the document, we first check if the database exists, then the collection, of they are valid, we will insert that document, otherwise we will throw an exception and handle it.
- Design method signatures carefully(Item 51): all methods signatures are well defined, and their names are self explanatory.
- Return empty arrays or collections, not nulls: when the user requests documents with some field, if the list is empty, he will be faced with empty list.
- Prefer for-each loops to traditional for loops(Item 58): we used for each loop instead of normal loops.



```
for (int id : ids) {
    try {
        jsonNodes.add(read(id));
```

- Know and use libraries(Item 59): I have used many librarires including but not limited to: Google Guava, Github Fge, Java lang, Java util.
- Beware the performance of string concatenation: because each time you concatenate to a string, new object is created, this was avoided by using the StringBuilder in the code whenever needed.
- Refer to objects by their interface(Item 64): just like the Query Handler interface usage.
- Use exceptions only for exceptional conditions(Item 69): I used the exceptions when needed, so for example, when creating a document, if the database does not exist or the collection, an exception is thrown and handled.
- Favor the use of standard exceptions(Item 72): all the thrown exceptions in my code are standards and not custom exceptions.
- Don’t ignore exceptions(Item 77): never ignored exceptions in the code.
- Synchronize access to shared mutable data(Item 78): just like what we did for the B+ tree by using the ReadWriteLock and the Multimap to make them thread safe.

HOW TO RUN THE CODE

To run the code, you will need to run the bootstrap jar file by this command:

java -jar BOOTSTRAP --num.of.nodes=3

num of nodes is optional, if you leave it without deciding it will operate at 5 nodes, notice that you must have docker turned on your pc.

After that, you need to run the client code to start working with the database, type help to see options, if you prefer to run the jar and see cli on cmd, you can run the Client jar file of the client by: java -jar Client

TOOLS AND DEVOPS PRACTICES

I used GitHub to upload and update my code whenever needed.

I used GitHub copilot and it made me more productive by 30% at least, it suggests code completion, and that helps a lot.

To turn on the nodes, the bootstrap must be outside the docker network, the figure explains everything.