

# ATYPON

**Atypon java and devOps training Final Project**

## **NoSQL database**

Name: Aws Saleh AbuObeid

Email: [awsahmad157@gmail.com](mailto:awsahmad157@gmail.com)

## Contents

Introduction .....	3
Requirements.....	4
Project Structure.....	5
Database implementation & data Structure.....	6
Data and schema format .....	7
Data access.....	8
Caching.....	9
Indexing.....	11
Queries, and query handling.....	12
Writing to and editing the Database (Read Server Side) .....	13
Writing and editing the Database (Controller Side).....	14
Node Cluster Implementation .....	14
Observer design pattern .....	15
Database Config page .....	16
Scalability/ Consistency issues in the DB .....	17
Multithreading and race condition .....	18
Security .....	19
Protocol.....	20
WebApp using the database .....	21
Clean Code principles.....	24
S.O.L.I.D Design principles .....	26
Effective Java.....	27
Design Patterns.....	32
Testing.....	32
DevOps practices .....	32
Final words.....	34

## ***Introduction***

### **NoSQL database**

A NoSQL database stores data in documents rather than relational tables. They are called NOSQL which means "not only SQL", but They can also be very versatile and can be used for a lot of applications with different the schemas and or requirements, NoSQL databases can be very flexible and not as rigid as standard SQL like MYSQL, the data is stored in JSON format and can be queried and accessed based on attributes of the document.

### **Project description**

And this project I am supposed to build and NoSQL database that saves the data in documents in JSON format the database gets mainly read requests and has a specific structure to support those requests.

it consists of a main controller which controls the security of the databases by handling the users and admins of each one, the controller also controls a cluster of read servers a user will be redirected to when he sends a read request on a database, it also acts as a load balancer, by controlling where the read requests are redirected to, it also handles all the write requests, and is responsible for keeping all the nodes in the cluster synchronized.

The read servers house the data, and server the queries on the databases, with different functions, such as query with index, or multiple indexes, or by the generated document id.

## ***Requirements***

- How you implemented the DB,
- The data structures used,
- Multithreading the locks,
- Scalability/ Consistency issues in the DB
- Security issues in the DB
- The protocol you implemented between the client and the server.
- Defending your code against the Clean Code principles (Uncle Bob).
- Defending your code against “Effective Java” Items (Joshua Bloch)
- Defending your code against the SOLID principles
- Design Patterns
- DevOps practices

This report should include all the above.

## ***Project Structure***

- The database consists of two parts, the read server, which holds the data, and is responsible for handling all the read requests, and the Main Controller, which handles the cluster of read Servers, and all the write commands.
- Both are spring boot applications, using spring helped by handling all the grunt work while I can focus on adding more features.
- The data is saved only on the read servers, when an update needs to be made to the database, the controller will package it into a JSON object and send it to the read server where it is quickly executed.
- When a user sends a read request, he is given an API key, which he uses to authenticate with the chosen read server.
- The communication between the Controller and the cluster is simple for the most part, the Controller sends updates for writes and API keys for users, and the server responds with a status and if asked, the load on the node.

## ***Database implementation & data Structure***

### **JSON Library used**

I decided to use the Jackson library, as it has a lot of functionality that I can use to implement any schema, the main JSONObject in it is the ObjectNode, with a superclass of JsonNode.

```
<dependency>
  <groupId>com.fasterxml.jackson.core</groupId>
  <artifactId>jackson-databind</artifactId>
</dependency>
```

Figure 1-jackson library

### **File system**

The database is implemented in JSON documents saved in .json files, each database has a folder with its name, the folder holds one schema file which contains the full database schema saved as a JSON format, where each attribute holds the schema of a collection in the database, it also contains a file for each collection where it holds the documents in a JSON array, and finally, for indexing, each indexed parameter has its own index table which is saved on a file named "collection\_indexName.index".

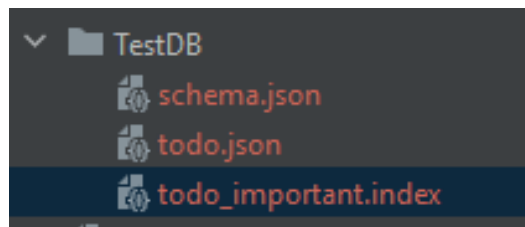


Figure 2-database files

## ***Data and schema format***

The schema is saved in a standard JSON schema, I took inspiration from MongoDB, each attribute has a type, has an optional required and default parameter, a parameter “\_id” which holds the object id generated by the database is added to each collection schema.

```
{
  "todo": {
    "properties": {
      "do this": {
        "type": "string",
        "required": true
      },
      "and this": {
        "type": "string"
      },
      "important": {
        "type": "boolean"
      },
      "_id": {
        "type": "string"
      }
    },
    "index": ["important"]
  }
}
```

Figure 3-schema format

```
{
  "id": 127096,
  "name": "Aws AbuObied",
  "password": "123456789",
  "_id": "93f7c19b-9996-4c12-8ba1-85d7959a0dc1"
}
```

Figure 4-data format

## Data access

I've created 2 layers to access the data, a **FileService** it handles all the interaction with the files and holds the directory of the data.

```
public interface FileService {
    File getCollectionFile(String DB,String colName);
    File getSchemaFile(String DB);
    void deleteCollectionFile(String DB,String colName);
    void deleteDB(String DB) throws IOException;
}
```

Figure 5-data format

And a **DocumentDao**, which interacts with the File service, and is responsible for the data storage, creation, deletion, and retrieval.

```
public interface DocumentDao {
    ObjectNode getCollection(String DB, String colName) throws IOException;
    Hashtable<JsonNode, List<String>> getIndexTable(String DB, String colName, String indexName) throws IOException;
    void setCollection(String DB, String colName, ObjectNode collection) throws IOException;
    void deleteCollection(String DB, String colName) throws IOException;
    void addCollection(String db, String collection, JsonNode schema) throws IOException;
    ObjectNode getSchema(String DB) throws IOException;
    void setSchema(String DB, ObjectNode schema) throws IOException;
    void addDocument(String DB, String colName, ObjectNode document) throws IOException;
    void deleteDocument(String DB, String colName, String doc_ID) throws IOException;
    void deleteDatabase(String DB) throws IOException;
}
```

Figure 6-data format

The interface has two data retrieval functions, **getDocuments** and **getSchema** and **getIndexTable**, the schema is returned as an ObjectNode, and the collection is returned as a ObjectNode where the key is the Object id for fast access on queries by id, and the indexTable is returned as a hashTable which links the value of the index with a list of object id's.

The rest of the functions are write functions that are invoked only by the controller, they are self-explanatory.



## Caching

After testing out with data structures for caching, It all seemed too crude to me, and wouldn't scale well, so I decided to use an already made library **EhCache**, which in hand with spring boot, caches functions annotated with `@Cacheable`, and so I applied a proxy design pattern together with the Dao for the caching.

It includes the **CachedDao**, the **FileDao**, and the DocumentDao interface.

**CachedDao** is an implementation of the DocumentDao interface, it represents the Proxy part of the proxy pattern, it uses the spring cache annotations which in turn use **Ehcache** to cache the output of the functions in memory, the annotations are set up so that writes evict the parts of the cache that have become outdated.

I've set cache to implements LRU algorithm for the heap, with a max size of 500m.

```
<cache name="collections"
    maxBytesLocalHeap="500m"
    eternal="false"
    memoryStoreEvictionPolicy="LRU"
    transactionalMode="off">
    <persistence strategy="none" />
</cache>
```

Figure 7-cache config

```
@Override
@Cacheable(value = "collections", key = "{#DB, #colName}")
public HashMap<String, ObjectNode> getDocuments(String DB, String colName) throws IOException {
    return dao.getDocuments( DB, colName);
}
```

Figure 8-cache config

I've the proxy pattern caches full collections in the memory, to make sure that all queries are Quick, not just id queries.

**FileDao** is an implementation of the DocumentDao interface, it represents the RealSubject part of the proxy pattern, it uses a file service to access the DBs files and do the read and write

operations, In the future, it would be simple to add caching for queries, as from my research, most famous databases cache specific queries instead of whole collections.

To make sure that users have access to the old data while it is being updated, I've made it so that that data is cached first, then updated, then the cache is evicted, meaning that a copy of the data being edited is always present in the cache.

```
private void addDocument(String db, String collection, JsonNode document)
{
    dao.getDocuments(db, collection);
    dao.addDocument(db, collection, (ObjectNode) document);
}
```

Figure 9-caching before writing.

Calling the get documents function will cache the data, and when the add function is done, that data is evicted.

## Indexing

I've implemented single attribute indexing by storing a table which links the value of an index to a list of object IDs that have that value, this table is stored inside a file, in a JSON format.

```
[
  {
    "value": true,
    "objects": [
      "73bfe01d-6f31-4008-abf5-8de21301ab07"
    ]
  },
  {
    "value": false,
    "objects": [
      "0685d620-ff1a-4c8a-9560-19fe98196474",
      "a5885857-5630-4fa7-98f9-29c0582d56f5"
    ]
  }
]
```

Figure 9-index file

At query time, the index table is loaded from the cache if available, the list of object IDs is taken from the table and used to load all the documents from the collection.

To achieve multi attribute indexing, for each attribute, the list of object IDs is loaded, and then the intersection is used to get the corresponding objects.

```
while (it.hasNext()) {
    String index = it.next();
    List<String> objects = dao.getIndexTable(DB, colName, index).get(query.get(index));
    allObjects.retainAll(objects);
}
```

Figure 9-index file

## Queries, and query handling

I've designed it so that the queries come in a specific structure, which would entail one of 4 possible query types :

- **QueryAll**, gets all the documents in a collection, it is done through a get request with the path `"/query/collectionName"`:
- **IndexQuery**, gets documents on an index or multiple indexes, it is done through a post request with the path `"/indexQuery/collectionName"`, with a body containing the indexes in a JSON format.
- **IdQuery**, fast query by document ID, it is done through a get request with the path `"/idQuery/collectionName/doc_id"`.
- **searchQuery**, gets documents on a non-indexed attribute or multiple attributes, it is done through a post request with the path `"/searchQuery /collectionName"`, with a body containing the attributes in a JSON format.

I've implemented a **Validation service** which implements the `QueryValidator` interface and validates the queries against the schema, to prevent indexing of nonexistent or removed attributes, or searching with a wrong data type, and checks the query format to prevent errors.

```
public interface QueryValidator {  
  
    String isValidIndexQuery(String DB, String collection, JsonNode query) throws IOException;  
  
    boolean isValidFormat(JsonNode query) throws IOException;  
  
    boolean resourceExists(String DB, String collection) throws IOException;  
  
}
```

Figure 8-QueryValidator

As the first step between the controller and the service, the **ReadHandler** service is responsible for handling and sending all the queries to the query service, it uses a **QueryValidator** instance

to check the if it's a valid query, then sends it to the query service to be executed and returns the queried data.

## ***Writing to and editing the Database (Read Server Side)***

There are a lot of operations the can be done on the database, and in the case of the read server, all of them will come from the controller, therefore no handling and checking of the data validity is required, the operations are represented in the Operation Enum.

```
public enum Operation {  
    ADD_DOCUMENT, DELETE_DOCUMENT, DELETE_COLLECTION,  
    SET_SCHEMA, DELETE_DATABASE, ADD_ATTRIBUTE, ADD_COLLECTION  
}
```

Figure 9- Operation Enum

For a write operation to be executed, a JSON formatted message is received from the controller, Ex:

```
{  
  "op": "ADD_DOCUMENT",  
  "collection": "student",  
  "document": {  
    "id": 127096,  
    "name": "Aws AbuObied",  
    "password": "123456789"  
  }  
}
```

The write commands have straight forward execution, the write handler is called, where it gets the needed data from the JSON messages received from the controller, the handler uses the Dao and the data is updated.

## ***Writing and editing the Database (Controller Side)***

The controller handles the write requests in a RESTful way, with put, delete, mappings, When a write request is sent to the controller, it is sent to the **Write handler** where it is validated and executed on the local schema file, then packaged and sent to the read servers to be executed.

## ***Node Cluster Implementation***

### **Docker**

Each read server is created and maintained inside a docker container, I decided on using the docker API to manage the containers, by sending request to the Docker rest API I can manage all the nodes with relative ease, for example here is how I run a container:

```
String response = WebClient.create().post().uri(url).contentType(MediaType.APPLICATION_JSON)
    .body(BodyInserters.fromValue(s))
    .retrieve().bodyToMono(String.class).block();
JsonNode resp = new ObjectMapper().readTree(response);

if (!resp.has( fieldName: "Id"))
    return false;
containerId = resp.get("Id").asText();

url = DB_URL + ":" + DOCKER_API_PORT + "/containers/" + READ_SERVER_NAME + "_" + id + "/start";
WebClient.create().post().uri(url).contentType(MediaType.APPLICATION_JSON)
    .body(BodyInserters.fromValue( body: ""))
    .retrieve().bodyToMono(String.class).block();
return true;
```

Figure 10- running container using Docker API

Where the string “s” holds the command to the docker API in a JSON format.

The rest of the operations like stopping and committing a container are in the same format.

## Observer design pattern

The controller needs to send update to all the read servers when write is preformed, and therefore an observer design pattern would be perfect for this role.

I've set up two main interfaces that represent the design pattern:

```
public interface Observer {  
    void update(ObjectNode message);  
}  
  
public interface Subject {  
    void addObserver(Observer observer);  
    void removeObserver(Observer observer);  
    void notifyObservers(ObjectNode message);  
}
```

Figure 11- Observer interfaces

Those are then implemented by the **ReadServerNode** class, which represents a read server, holding all values that represent it, such as:

- int id.
- containerId.
- int port.
- boolean dirty, if the data in the node is dirty.
- Int load, the number of sessions currently running on the server.

And the ReadServerSubject, which only has the purpose of updating the nodes.

## ReadServerManager

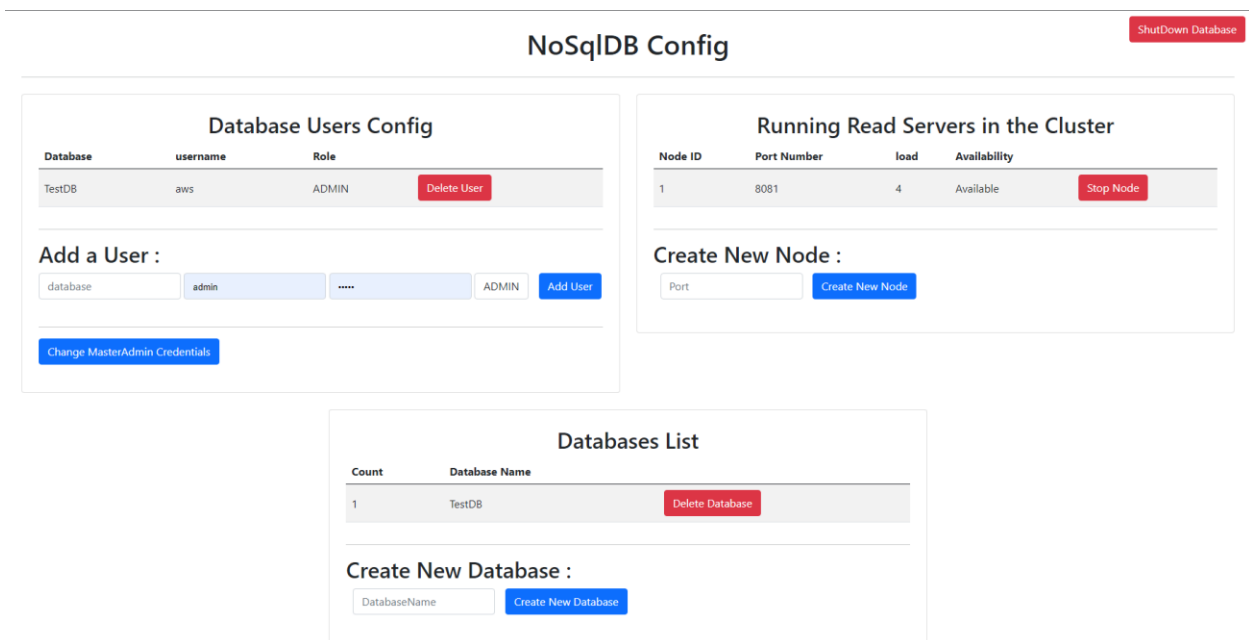
The **ReadServerManager** class manages all the nodes in the cluster, it holds all of them, it does operations such as, start a new node, or stop one. it uses the Subject of the Observers to send updates to the servers, also keeps track of the IDs of each one, and does the exception and error handling for when a container misbehaves.

It houses functions such as sendReadToNode(), where it chooses what node give to the reader based on the load on that node, and sends it the generated API key .

After every write to the database, a container will be committed to the local image, for when a new Node needs to be created, it will be built using the latest image, that has the most up to date data, The manager also checks for when a port is free or not.

The manager is used by the Admin controller, which in turn executes requests from the admin page .

## ***Database Config page***



The screenshot displays the 'NoSqlDB Config' admin interface. At the top right is a 'ShutDown Database' button. The main content is divided into three sections:

- Database Users Config:** Contains a table with columns 'Database', 'username', 'Role', and a 'Delete User' button. The table has one entry: 'TestDB', 'aws', 'ADMIN'. Below the table is an 'Add a User :' form with fields for 'database', 'username' (filled with 'admin'), 'password' (masked with '\*\*\*\*'), 'Role' (filled with 'ADMIN'), and an 'Add User' button. A 'Change Master/Admin Credentials' button is at the bottom.
- Running Read Servers in the Cluster:** Contains a table with columns 'Node ID', 'Port Number', 'load', 'Availability', and a 'Stop Node' button. The table has one entry: '1', '8081', '4', 'Available'. Below is a 'Create New Node :' form with a 'Port' field and a 'Create New Node' button.
- Databases List:** Contains a table with columns 'Count' and 'Database Name', and a 'Delete Database' button. The table has one entry: '1', 'TestDB'. Below is a 'Create New Database :' form with a 'DatabaseName' field and a 'Create New Database' button.

Figure 12- Admin Page

The admin page has all the functionality that the database admin can do, which are:

- Manage Databases.
- Manage Database users and permissions.
- Manage Nodes in the cluster.
- Change the database admin credentials.

Which are all managed by the admin controller.



## ***Scalability/ Consistency issues in the DB***

as mentioned before, the database is scalable through the admin page, by providing a port, the database cluster can scale horizontally.

Having multiple containers holding the same data may cause data inconsistency, which is why I've implanted features to keep the data consistent.

I've implanted a dirty flag, which is made true when a node doesn't respond to an update, it is flagged as dirty, the Manager will try to fix it by stopping the container and recreating a new one with the latest data, if it fails also, it is deleted.

If none of all of the servers fail to respond correctly, the write command is canceled and an error message recommending a server restart is printed.

```
public synchronized boolean updateReadServers(ObjectNode message) {
    logger.info("Sending update to " + count + " nodes in the cluster");
    subject.notifyObservers(message);

    if (!isUpdateSuccessful()) {
        for (ReadServerNode node : readServerNodes)
            node.cleanNode();
        return false;
    }

    for (ReadServerNode node : readServerNodes)
        if (node.isDirty()) {
            logger.info("attempting to fix read server id : " + node.getId() + " -restarting...");
            node.stopContainer();
            if(commitNeeded) commitContainer();
            if (!node.runContainer()) {
                logger.error("failed to restart stopping server id : " + node.getId());
                stopNode(node.getId());
            } else node.cleanNode();
        }
    }
    commitNeeded=true;
    return true;
}
```

Figure 13- updateReadServers

## ***Multithreading and race condition***

As spring already handles most of the threading, I only needed to make sure that some of my services are thread safe and don't cause problems, such as setting the execute Write as Synchronized in the ReadServer, so that 2 threads access the files at the same time.

```
@Service
public class WriteHandler {

    @Autowired
    DocumentDao dao;

    public synchronized void executeWrite(ObjectNode body) throws IOException {
        Operation op= Enum.valueOf(Operation.class,body.get("op").asText());
```

Figure 14-Synchronized executeWrite

All other write commands on the read server and on the controller is set to be thread safe.

## Security

The database is fairly secure, as I have used spring security to secure both the controller and the read server.

The controller is secured by username and password, and is needed for logging into the admin page, and for accessing the write and read privileges for each database, each user is configured using a **DBUser** object, which implements **UserDetails** from spring security, all of the users are then saved into the file system, and accessed using the Controller Dao.

```
@Data
public class DBUser implements UserDetails {

    private String database;
    private String role;
    private String username;
    private String password;
```

Figure 15- DBUser class

I also implemented an **InMemoryUserDetailsManager** bean so I can edit the users during the runtime of the program.

```
@Bean
public InMemoryUserDetailsManager inMemoryUserDetailsManager() {
    InMemoryUserDetailsManager manager = new InMemoryUserDetailsManager(createAdmin());

    for(DBUser i : dao.getUsers())
        manager.createUser(i);

    return manager;
}
```

Figure 16- InMemoryUserDetailsManager

And finally, I implemented a Constants Class to save all the data needed by controller to run.

```
public class Constants {  
    public static final int CONTROLLER_PORT = 8080;  
    public static final int DOCKER_API_PORT = 2375;  
    public static final int IMAGE_INTERNAL_PORT = 5050;  
    public static final int READ_SERVER_STARTING_PORT = 8081;  
    public static final String IMAGE_TAG = "awssaleh/nosql-read-server";  
    public static final String CONTROLLER_API_KEY = "Controller_API_key";  
    public static final String READ_SERVER_NAME = "read_server";  
    public static final String HOST_URL = "http://localhost";  
    public static final String DB_URL = "http://localhost";  
}
```

Figure 17- Constants class

## ***Protocol***

I implemented a pretty simple protocol, where a user connects to the Controller first, using a username and password, after that if he wants to read request, he connects to “/read” where he will be given a link to the chosen read server, and a onetime use API key, he must then connect to “/authenticate” on the read server using the key, and then he can query at “/query”.

if the session holder stays idle at the read server for more than 15 minutes, his session will end, and he will have to re authenticate.

## WebApp using the database

In order to save time, decided on using the student grading system application from a previous assignment, but instead of using MYSQL, I will be using mine.

I've developed a few tools that a user of the database can import into their application, and use it to connect and access the database, it handles the protocol and works as a black box, the user only needs to provide a few initial parameters regarding the database.

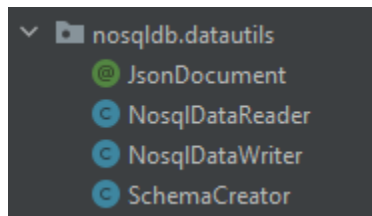


Figure 18- Nosqlldb.datautils

To set up the schema for the database, the user needs to annotate classes with `JsonDocument`, which informs the schema creator that this is a schema collection, it also takes an optional string array parameter, which represents the indexed fields.

```
@Data
@JsonDocument(index = {"cid", "sid"})
public class CourseMark {

    @JsonProperty(required = true)
    private String cname;
    @JsonProperty(required = true)
    private String cid;
    @JsonProperty(required = true)
    private int sid;
    @JsonProperty(required = true)
    private float mark;

    public CourseMark(String cname, String cid, int sid, float mark) {
        this.cname = cname;
        this.cid = cid;
        this.sid = sid;
        this.mark = mark;
    }

    public CourseMark() {
    }
}
```

Figure 19- @JsonDocument

After that, the user only needs to set up some configurations for the reader and the writer objects, such as a username and password created at the admin page.

```
StudentDatabaseDao(){
    try {
        ObjectNode schema=new SchemaCreator().
            createSchema( basePackage: "com.studentGrading.springapp");

        nosqlDataReader=new NosqlDataReader();
        nosqlDataWriter=new NosqlDataWriter();
        nosqlDataWriter.setUsername("studentWebsite");
        nosqlDataWriter.setPassword("123456789");
        nosqlDataWriter.setControllerURL("http://localhost:8080");
        if(!nosqlDataWriter.setSchema(schema))
            System.out.println("failed to connect to controller");

        nosqlDataReader.setUsername("studentWebsite");
        nosqlDataReader.setPassword("123456789");
        nosqlDataReader.setControllerURL("http://localhost:8080");

        if(!nosqlDataReader.connect())
            System.out.println("failed to connect to read server");
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

Figure 20- initial nosqlldb config

After that the reading and writing is straight forward.

```
public void addStudent(String name, int id, String password) {
    try {
        JsonNode student = mapper.readTree(mapper.writeValueAsString(new Student(name, id, password)));
        nosqlDataWriter.addDocument( colName: "Student",student);
        System.out.println("Add Complete");
    } catch (Exception e){
        System.out.println("Add Failed.. ");
    }
}
```

```
public Student getStudent(int id) {
    ObjectNode index= mapper.createObjectNode();
    index.put( fieldName: "id",id);
    JsonNode queried=nosqlDataReader.query( colName: "Student",index).get(0);
    ((ObjectNode)queried).remove( propertyName: "_id");
    return mapper.convertValue(queried,Student.class);
}
```

Figure 21- reading and writing to the database

The reader and writer communicate with database using a web client, the reader logs through the controller and then is directed to a read server, and the writer has a connection to the controller.

This demo doesn't take advantage of all the possibilities of the database, I didn't include them to save time.

## ***Clean Code principles***

Writing a code that works is significant; fixing a bug is more significant. But being able to understand what you have developed after a few months is the actual goal. Most of the clean code principles focus on having an understandable code that whoever reads it after the author will get it. Nevertheless, I also believe since this code had only one developer that being able to understand your code after a while is a valid argument too. In this section, I'll go through the principles I have used to keep my code clean for me and others equally.

### **Comments**

Code comments follow these rules:

The code is clean from the comment-out code.

- Comments are neoteric, appropriate, well written, and to the point.
- Comments are written to emphasize vague code parts.

### **Environment**

Although the code is distributed, has multiple components, and runs different areas concurrently, the building process is held as one step. Distribution is in the background, and the code can be considered one block.

### **Functions**

Functions follow these rules:

- The code has no dead functions. If there is a method in any class, then it's used somewhere in the code.
- Each method does one and only one task, and it's relatively small.
- There are no flag or output arguments.
- Each method has up to three arguments.

### **General**

- The code is written in JAVA, and comments are written in English.
- Function's behavior is clear, correct, and other programmers would expect it.



- According to the **Duplication principle**: general methods, variables, and objects are written once and called as needed in different code areas.
- Following the **Dead Code principle**: each part of the code has usage and is reachable from other parts.
- As mentioned in the **Vertical Separation principle**: variables and functions are defined close to where they are used. Local variables are declared just above their first usage and have a small vertical scope.
- As in the **Inconsistency principle**: All parts of the code are written similarly.
- To satisfy the **Artificial Coupling principle**: functions, constants, and variables are separated as required.
- For the **Feature Envy principle**: The class methods work inside their scope.
- There are no clutter methods or variables.
- The code is free from selector arguments, Hungarian notation, and magic numbers.
- The code is precise, encapsulates its conditions, has no negative conditions, and follows the standard conventions.
- Try catch blocks are written as needed, the same thing for exceptions.
- Method's name is representative of its functionality.

## JAVA

- Wildcards are used if there is no ambiguity to avoid a long list of imports.
- Constants are public and accessible to avoid constant's inheritance.
- Enums and constants are used based on the required functionality.

## Names

0. Names in general are descriptive, unambiguous, and at the appropriate level of abstraction.

1. Long Names are used for Long Scopes.
2. There are no encodings.
3. Class's name is a noun, methods are verbs, and variables are short and meaningful.

## ***S.O.L.I.D Design principles***

- **Single Responsibility:**

Each class has one job, and the implemented functionality is around the main task. Class size depends on the task size. If irrelevant or new functionality is needed, it will be written in a new class.

```
public interface JsonSchemaVaildator {  
  
    String validateDocument(JsonNode schema, JsonNode document);  
  
    boolean isValidSchema(JsonNode schema);  
}
```

Figure22- Single Responsibility

- **Open/Closed:**

The code is extendable, open for changes, and adding new behaviors. On the other hand, the current state of the module is close. Any future changes won't affect the current state.

- **Liskov substitution**

Passing a sub-class object to a method that accepts its super class won't crash the execution.

- **Interface segregation**

The module has interfaces as needed. There is no general-purpose interface.

```
Service
public class SchemaValidator implements JsonSchemaVaildator,JsonSchemaApplicator {
```

Figure23- Single Responsibility

- **Dependency inversion principle states**

Every class eventually depends on an interface or abstract class. There are no dependencies between concrete modules.

```
@Service
public class WriteHandler {

    @Autowired
    private ControllerDao dao;
    @Autowired
    private ReadServersManager readServersManager;
    @Autowired
    JsonSchemaVaildator validator;
    @Autowired
    JsonSchemaApplicator applicator;
```

Figure24- Dependency inversion

## ***Effective Java***

Absolutely fantastic book, sift through it any time I'm displeased with my code structure or don't know the best practice for designing a subsystem, in this project I've integrated as many of them as I can.

- Enforce the singleton property with a private constructor or an Enum type, The singleton design pattern was used heavily in my project, mainly by Auto Wiring using spring, sense it defaults to making the scope a singleton.

```

@Autowired
private ControllerDao dao;
@Autowired
private ReadServersManager readServersManager;
@Autowired
private WriteHandler writeHandler;

```

Figure 25- AutoWire

- Enforce noninstantiability with a private constructor, I've implemented this into my code also, but very sparingly.

```

private ControllerDaoImpl() {
    this.mapper = new ObjectMapper();
}

```

Figure 26- private constructor

- **Prefer dependency injection to hardwiring resources**, dependency injection was also used heavily in my project, mainly by Auto Wiring using spring, which has helped a lot in making my code better, more readable, and less coupled.
- **Minimize mutability**, most of my Variables are final and private to provide data protection, most entity classes are immutable, so they are thread-safe and require no synchronization, unless there is a good reason.
- **In public classes, use accessor methods, not public fields**, Encapsulation of data using getters/setters. Class members cannot be accessed directly, a getter should be used, this was usually achieved using **Lombok**.

```

@Data
public class DBUser implements UserDetails {

    private String database;
    private String role;
    private String username;
    private String password;

    @Override
    public String getPassword() { return password; }

    @Override
    public String getUsername() { return username; }
}

```

Figure 27- accessor methods

- **Minimize the accessibility of classes and members**, class members are always private and are encapsulated, since classes with public mutable fields are not thread safe.

```

public class ReadServersManager {

    private int idCount;
    private int count;
    private final Subject subject;
    private final ArrayList<ReadServerNode> readServerNodes;
}

```

Figure 28- accessor methods

- **Always override hash Code when you override equals.**
- **Obey the general contract when overriding equals.**

```

@Override
public boolean equals(Object o){
    if (o == this) {
        return true;
    }
    if (!(o instanceof DBUser)) {
        return false;
    }
    DBUser user=(DBUser)o;
    return user.getUsername().equals(getUsername()) &&
        user.getDatabase().equals(getDatabase());
}

@Override
public int hashCode() {
    int result = 17;
    result = 31 * result + database.hashCode();
    result = 31 * result + username.hashCode();
    return result;
}

```

Figure 29- equals & hashCode

- **Avoid creating unnecessary objects**, no new objects are created when we can reuse an existing one. Common use of one method to increase code reusability.
- **Avoid finalizers and cleaners**, I've avoided them since they cause performance issues.
- **Favor composition over inheritance**, I tried to use composition instead of inheritance when possible, to avoid all of its issues.
- **Prefer interfaces to abstract classes**, as interfaces are more flexible, I've used them extensively in this project, both for the flexibility and the added level of abstraction.

```

public interface JsonSchemaVaildator {

    String validateDocument(JsonNode schema, JsonNode document);

    boolean isValidSchema(JsonNode schema);

    void applySchema(JsonNode document, JsonNode schema);
}

```

Figure 30- interfaces

- **Eliminate unchecked warnings**, I've eliminated as many warnings as possible.
- **Use Enums instead of int constants**, since the number of write operations are limited and constant, I've used an Enum instead of Constant.

```
public enum Operation {  
    ADD_DOCUMENT, DELETE_DOCUMENT, DELETE_COLLECTION, SET_SCHEMA,  
    DELETE_DATABASE, ADD_ATTRIBUTE, ADD_COLLECTION  
}
```

Figure 31-Enum

- **Consistently use the Override annotation**, It's a good practice to use @Override annotation on all methods that override superclass or interface methods, the @Override annotation was used on method declarations that override declarations from interfaces as well as classes.

```
@Override  
public List<DBUser> getUsers(){...}  
@Override  
public synchronized void addUser(DBUser user) throws IOException {...}  
@Override  
public synchronized void deleteUser(String username) throws IOException {...}  
@Override  
public ObjectNode getDatabaseSchema(String DB) throws IOException {...}
```

Figure 32-Override

## ***Design Patterns***

Most or all the designs used in this project were mentioned in other parts.

## ***Testing***

I didn't implement any testing.

## ***DevOps practices***

I've implanted a CICD pipeline using GitHub actions, to automate the process of setting up and deploying the Database.

It is a simple pipeline that takes the code in the repository, build it, package it, and creates a docker image, that image is then pushed into my DockerHub (awssaleh) where it can be pulled from and start the database with a simple command, The figure below is the used yaml file.



```

4   name: Java CI with Maven
5
6   on:
7     push:
8       branches: [ "master" ]
9     pull_request:
10      branches: [ "master" ]
11
12  jobs:
13    build:
14
15      runs-on: ubuntu-latest
16
17      steps:
18        - uses: actions/checkout@v3
19        - name: Set up JDK 1.8
20          uses: actions/setup-java@v3
21          with:
22            java-version: '8'
23            distribution: 'temurin'
24            cache: maven
25        - name: Build with Maven
26          run: mvn -B package --file pom.xml
27
28        - name: Set up QEMU
29          uses: docker/setup-qemu-action@v2
30
31        - name: Set up Docker Buildx
32          uses: docker/setup-buildx-action@v2
33
34        - name: Login to DockerHub
35          uses: docker/login-action@v2
36          with:
37            username: ${ secrets.DOCKERHUB_USERNAME }
38            password: ${ secrets.DOCKERHUB_TOKEN }
39
40        - name: Build and push
41          uses: docker/build-push-action@v3
42          with:
43            context: .
44            push: true
45            tags: awssaleh/nosql-controller

```

Figure33- Single Responsibility

## ***Final words***

This is my implementation of a NoSQL Database that you requested, I implemented it based on my understanding, I did a lot of searches and still needs more modifications and enhancements but for this moment, this code represents me at my best.

Thank you for taking the time to read this report.