



Final Project
NoSQL Database
Dr. Motasem Diab
Dr. Fahed Jubair
Mohammad Amjad AlQuraan

Table of content

Acknowledgement	3
Chapter one: Introduction	4
Chapter two: Design	5
Chapter three: Implementation	7
3.1 Technology	7
3.2 File system hierarchy in the node.....	7
3.3 Indexing.....	8
3.4 Master	9
3.5 Load Balancer	12
3.6 Node	13
3.7 Cash	16
3.8 Client	17
3.9 Implemented Protocol	19
3.10 Multithreading	20
3.11 Data structures	20
3.12 Scalability/ Consistency issues in the DB	20
3.13 Security issues in the DB	21
3.14 Solid Principles	21
3.15 Effective Java	21
3.16 Clean Code.....	27
3.17 Design patterns	31
3.18 DevOps	31
Chapter four: Demo Application	32

Acknowledgement

I joined Atypon training program on 25/1/2022 and till this moment I'm still learning and getting new experience every day, I want to thanks Dr.Motasem and Dr.Fahed for their efforts, great experience and knowledge they gave me through this great chapter in my life, I'm thankful for everything and I'm truly honored to be one of your students.

Chapter 1 Introduction

NoSQL databases are non-relational databases which means they store data in JSON objects rather than storing them as rows in relational tables.

NoSQL database provides fast queries and can handle big data because they can expand horizontally, which is a big advantage over relational databases.

There are four types of NoSQL databases:

- Document databases.
- Key-value stores.
- Column-oriented databases.
- Graph databases.

Figure 1.1 shows popular NoSQL databases



Figure 1.1: NoSQL databases

Chapter 2 Design

In this chapter I will talk about the system design of the NoSQL database.

The system consists of three applications:

- Client
- Master
- Node

And I will talk about them in details in the following pages.

Client will connect to the master and authenticate itself; the master will return port number of a node if client is authenticated.

After connection is complete the client will start read and write to a specific node only, later I will describe the duplication of the data to other nodes.

Figure 2.1 shows the client workflow.

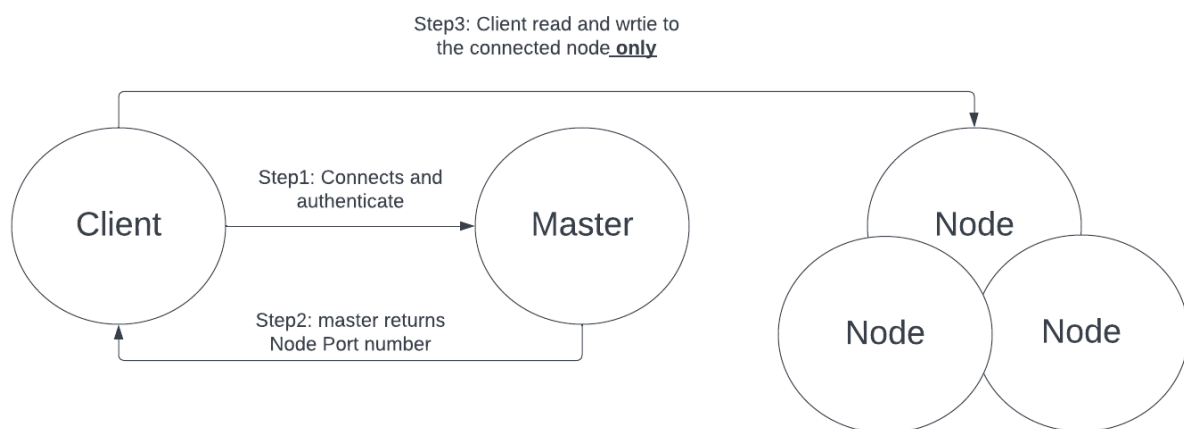


Figure 2.1: Client workflow

Master is the main component of the design it links the client with the nodes, but also it has important functionality, which is duplicating the data in all nodes,

When a node gets create, update, delete operation from the client, it notifies the master, and the master notifies all other nodes.

Figure 2.2 shows the master workflow with nodes.

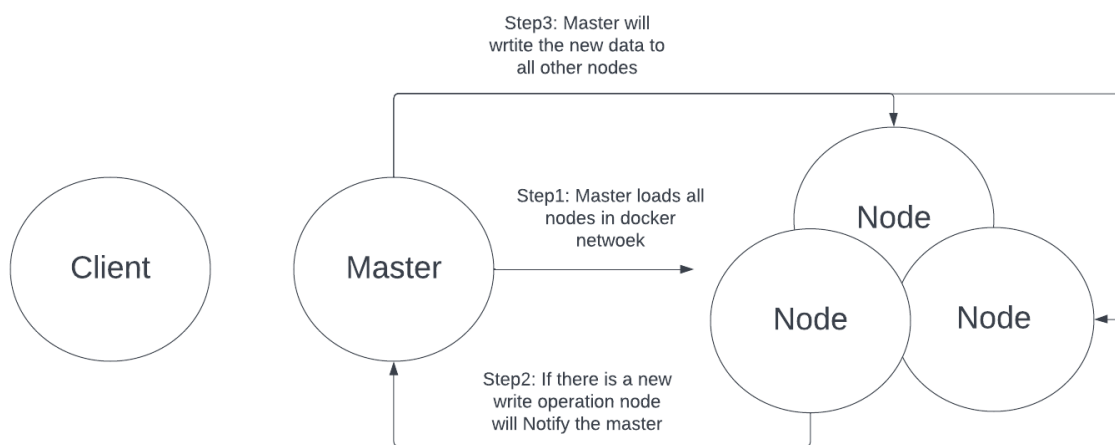


Figure 2.2 Master workflow

Nodes are the server that store schemas, types, and JSON objects of the database in the file system as represented in Figure 2.3.

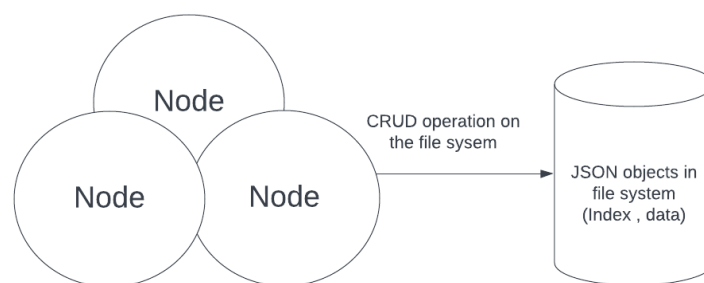


Figure 2.3 Nodes CRUD to files system

Chapter 3 Implementation

In this chapter I will go through the implementation of my solution for the NoSQL design in details.

3.1 Technology

I implemented the client, master and node using Spring boot.

Each one of them will listen on different port number, they will communicate with each other by calling API based on a protocol I will explain later in the report.

3.2 File system hierarchy in the node

The hierarchy of database file will be as follows

schema directory has a list of type directories, inside the type there are:

- Indexes directory contains:
 - o List of Indexed property name and they contain:
 - JSON object of property value store list of IDs.
- JSON object named with it unique IDs.
- JSON object represents the type schema inside it.

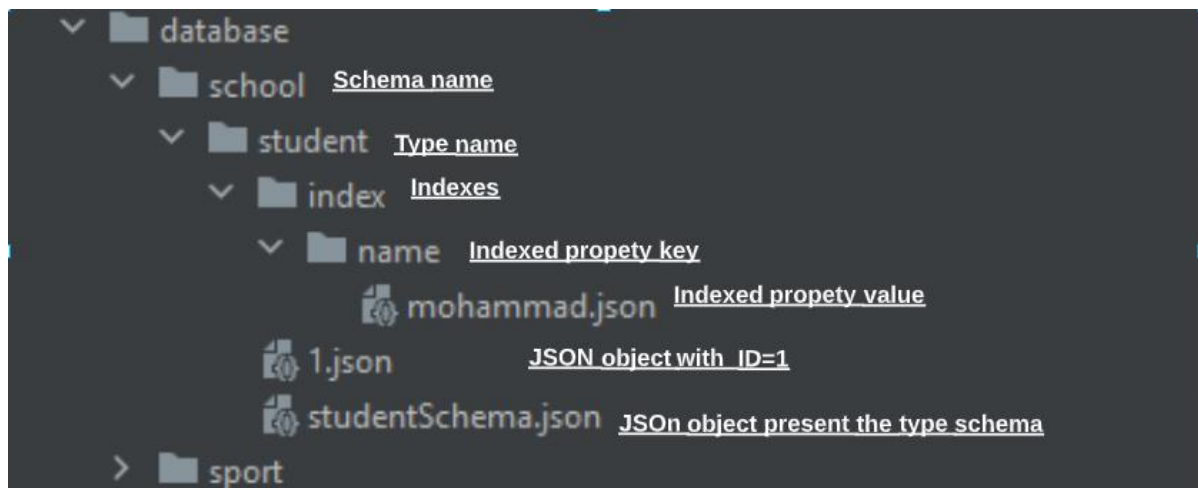


Figure 3.1: File System hierarchy

3.3 Indexing

For the indexing I adopt the concept of cluster and non-clustered indexing.

First, I made the ID property indexed automatically by naming the JSON object in the file system storage with the ID number as shown in Figure 3.1 above.

Second the user can index another property, it could be any property and it shouldn't be unique to be indexed.

For example, if the user wants to index the name property, and there are two objects with name "mohamamd" it will store the two objects ID inside the index and use them later in read, update, and delete any object with name=mohamamd

Figure 3.2 shows index example.

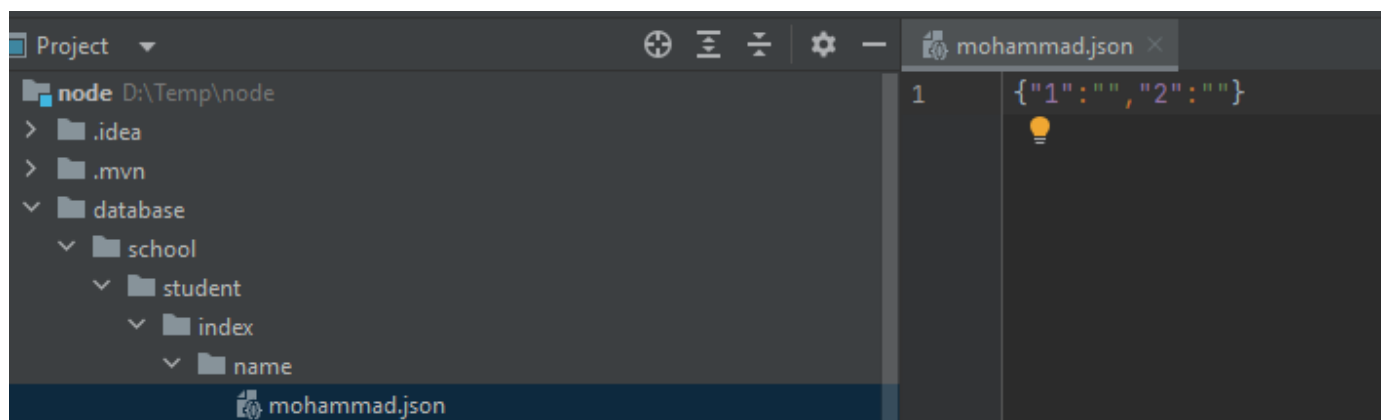


Figure 3.2 Index example

3.4 Master

The master as I mentioned previously is the link between node and client, and it notifies all nodes if some changes made in one node.

Figure 3.2 shows the master classes

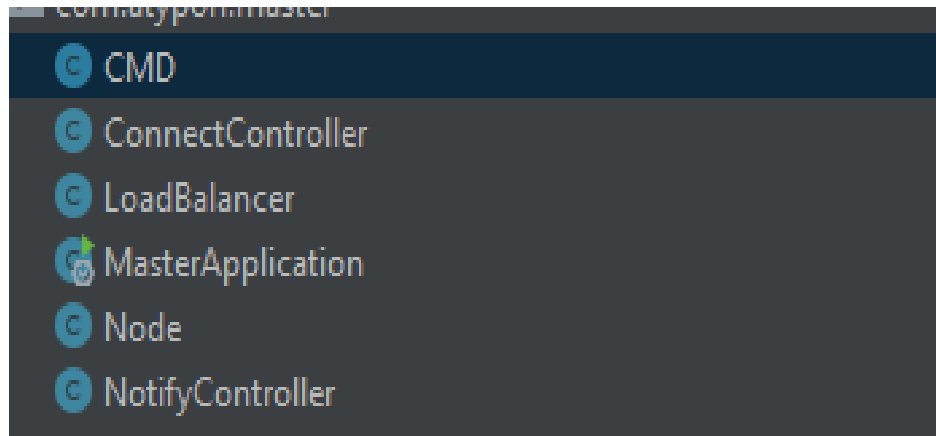


Figure 3.2: master classes.

CMD class used to call methods that can execute windows commands.

I used it to run docker container as shown in Figure 3.3.

```
public class CMD {  
    public static void runNodeContainer(int nodePort) {  
        try {  
            Runtime.getRuntime().exec( command: "docker run -p "+nodePort+":8085 node");  
            Node node = new Node(nodePort);  
        } catch (IOException e) {  
            throw new RuntimeException(e);  
        }  
    }  
}
```

Figure 3.3: CMD class

Connect Controller is a class used to handle the connecting request the comes from the client as shown in Figure 3.4.

It is responsible to authenticate the client and return a node port number from the load balancer.

```
@GetMapping(path = "/connect")
int isValidLogin(@RequestParam("username") String username,
                @RequestParam("password") String password){
    int portNumber = LoadBalancer.getCurrentNodeTurn();
    if (users.get(username).equals(password))
        return portNumber;
    else
        return 0;
}
```

Figure 3.4 connect controller.

Node class is the class that represents node entity in the master as shown in Figure 3.5.

```
14 usages
public class Node {
    2 usages
    private final int portNumber;
    2 usages
    private static final ArrayList<Node> listOfNodes = new ArrayList<>();
    2 usages
    Node(int nodeNumber){
        this.portNumber = nodeNumber;
        listOfNodes.add(this);
    }
    5 usages
    public int getPortNumber() { return portNumber; }
    1 usage
    public static ArrayList<Node> getListOfNodes() { return listOfNodes; }
    5 usages
    public void callNode(String uri){...}
}
```

Figure 3.5: Node class.

Notify Controller is a controller that handles the request comes from the node.

When a node gets updated, it should notify the master with its update, and the master will write the updates to all other nodes. Figure 3.6 shows the Notify Controller class.

```
@RestController
public class NotifyController {

    5 usages
    private static final ArrayList<Node> listOfNodes = Node.getListOfNodes();
    5 usages
    String getParams(HashMap<String,String> params){...}

    @GetMapping(path = "/notify/create/schema")
    void notifyNodesSchemaCreat(@RequestParam HashMap<String,String> params){...}

    @GetMapping(path = "/notify/create/type")
    void notifyNodesTypeCreat(@RequestParam HashMap<String,String> params){...}

    @GetMapping(path = "/notify/create/object")
    void notifyNodesObjectCreat(@RequestParam HashMap<String,String> params){...}

    @GetMapping(path = "/notify/update")
    void notifyNodesUpdate(@RequestParam HashMap<String,String> params){...}

    @GetMapping(path = "/notify/delete")
    void notifyNodesDelete(@RequestParam HashMap<String,String> params){...}
```

Figure 3.6 Notify Controller class.

3.5 Load Balancer

Load balancer is the class that is responsible for balancing the load on the nodes, each time client connects to a node, the load balancer will return current node port number turn.

The Load balance implements the round robin algorithm, round robin assigns clients to nodes equally in circular order and treats all nodes with same priority.

Figure 3.7 shows the implementation of Load Balancer class.

```
public class LoadBalancer {  
    2 usages  
    private static final int firstPort = 8085;  
    2 usages  
    private static final int lastPort = 8090;  
    4 usages  
    private static int currentNodeTurn = 8085;  
    1 usage  
    public static void loadNodes(){  
        // loop through nodes port numbers and run container  
        for (int i = firstPort; i <= lastPort ; i++) {  
            Node node = new Node(i);  
            CMD.runNodeContainer(i); // run a node in docker container  
        }  
    }  
    1 usage  
    public static int getCurrentNodeTurn() {  
        int nodePortNumber = currentNodeTurn;  
        if (currentNodeTurn+1 > lastPort)  
            currentNodeTurn=firstPort;  
        else  
            currentNodeTurn++;  
        return nodePortNumber;  
    }  
}
```

Figure 3.7 Load Balancer

3.6 Node

The node class represents the server that handle crud API's and writes the changer to its own file system. Figure 3.8 shows the packages and classes inside the node application, and we will talk about them in details.

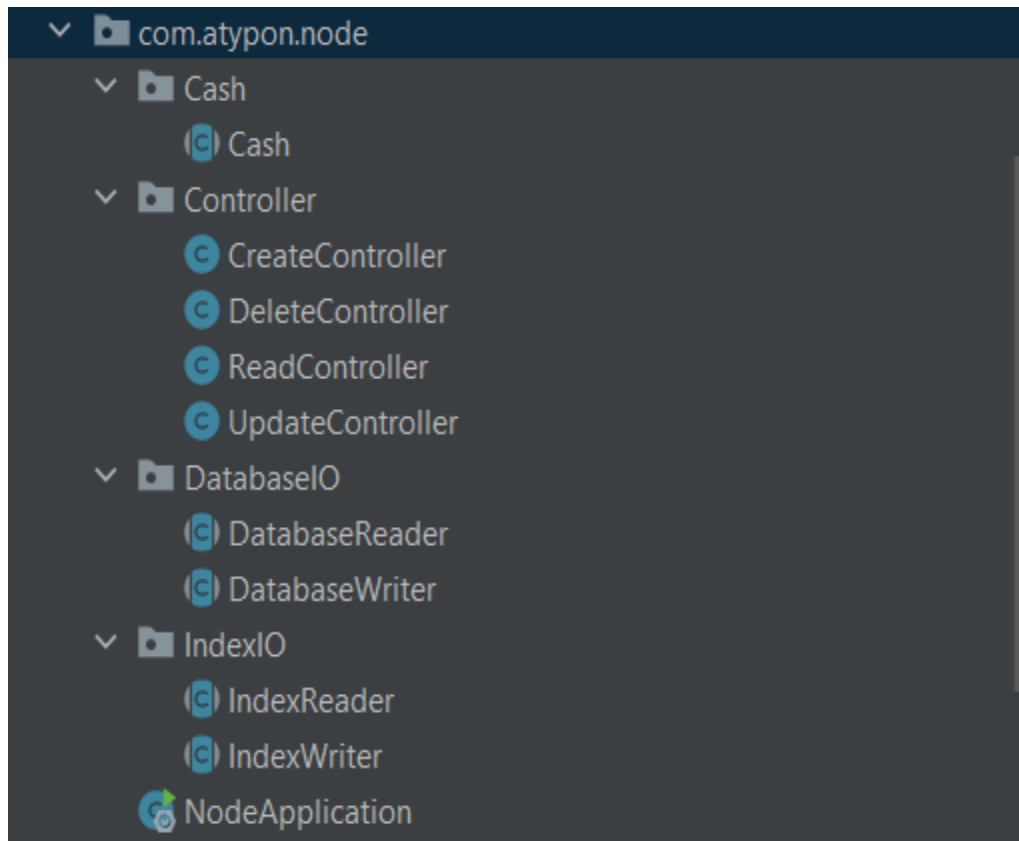


Figure 3.8 Node application packages and classes

Controller package handles all CRUD calls that come from client I will talk about the protocol between them later in the report.

Database IO packager container the database reader and writer classes.

Reader class is responsible for methods that deals with reading data from the file system as shown in Figure 3.9.

```

14 usages
public abstract class DatabaseReader {
    15 usages
    static BufferedReader reader;
    2 usages
    public static Boolean isFileExist(String path){...}
    1 usage
    public static Boolean isTypeSchema (String path, JSONObject jsonObject){...}
    5 usages
    public static File[] getListOfFiles(String path){...}
    3 usages
    public static ArrayList<Map<String,Object>> readObject(String condition ,String path){...}
    2 usages
    public static Map <String,Object> readObjectByID(String path){...}
    1 usage
    public static ArrayList<Map<String,Object>> readAllObject(String path ){...}
}

```

Figure 3.9: Database reader

Writer class is responsible for methods that deals with writing data to the file system as shown in Figure 3.10.

```

public abstract class DatabaseWriter {
    6 usages
    static BufferedWriter writer;
    4 usages
    public static boolean createDirectory(String path){...}
    2 usages
    public static boolean creatJsonObject (String path , String jsonObject ){...}
    1 usage
    public synchronized static boolean updateJsonObject(String typePath , String condition , String property){...}
    1 usage
    public synchronized static boolean deleteJsonObject(String condition , String typePath){...}
}

```

Figure 3.10 Database writer.

When dealing with indexes I was implementing its methods inside Database readers and writer but then I noticed that I'm breaking the single responsibility design principle, so I decided to create their own reader and writers

Index Writer class is responsible for methods that deals with writing index data to the file system as shown in Figure 3.11.

```

6 usages
public abstract class IndexWriter {
    3 usages
    static BufferedWriter writer;
    4 usages
    private static void creatIndexJsonObject (String path , String jsonObject ){...}
    1 usage
    public static void checkObjectHasIndex(HashMap<String,String> params , String typePath){...}
    2 usages
    public static void insertToIndexObject(String id , String typePath , String property){...}
    1 usage
    public static void removeFromIndexObject(String id , String typePath , String property){...}
    1 usage
    public static void removeFromAnyIndex(String id , String typePath){...}
}

```

Figure 3.11 Index writer

Index reader class is responsible for methods that deals with reading index data to the file system as shown in Figure 3.12.

```

public abstract class IndexReader {
    4 usages
    static BufferedReader reader;
    // Return index json object
    3 usages
    public static String getIndexObject(String path){...}
    // return if the condition is indexed
    4 usages
    public static boolean isConditionHasIndex(String condition , String path){...}
    // return all the IDs in index object
    1 usage
    public static Set<String> getIndexObjectIDs(String condition, String path){...}
    // Read all objects based on indexes
    3 usages
    public static ArrayList<Map<String,Object>> readObjectByIndex(String condition, String path ){...}
}

```

Figure 3.12 Index reader.

3.7 Cash

Cash provides fast reads from the main memory instead of reading from the file system storage, it is simply based on Hash Map, and it is fixed size, also it changes along with the database changes.

The hash map of the cash is between the path of specific JSON object and its own data. So, instead of reading the data from the actual path we get the data from the Hash Map. Figure 3.13 shows the cash class implementation.

```
10 usages
public abstract class Cash {
    1 usage
    private static final int CASH_MAX_SIZE = 10;
    5 usages
    private static final HashMap<String,String> cash = new HashMap<>();
    1 usage
    public static String readFromCash(String objectPath){...}
    2 usages
    public static void writeToCash(String objectPath , String jsonObject){...}
    1 usage
    public static void removeFromCash(String objectPath){...}
    3 usages
    public static boolean isInCash(String objectPath) { return cash.containsKey(objectPath); }
    2 usages
    public static boolean isFull() { return cash.size()==CASH_MAX_SIZE; }
}
```

Figure 3.13 Cash class

3.8 Client

Client class is responsible for connection to the node and call the CRUD operations API. But first it should connect to the controller to get a node port number, start communicating with it. Figure 3.14 show the client classes.

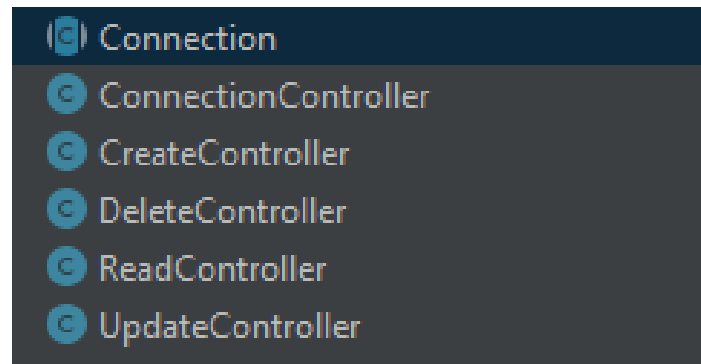


Figure 3.14 client classes

Connection represents the connection status in the client. Figure 3.15 shows the implementation of connection class.

```
16 usages
public abstract class Connection {
    2 usages
    private static int nodePort;
    2 usages
    private static boolean connected = false;
    1 usage
    public static void connectToMaster(String username, String password) {...}

    8 usages
    public static int getNodePort() { return nodePort; }

    1 usage
    public static void setNodePort(int nodePort) { Connection.nodePort = nodePort; }

    6 usages
    public static boolean isConnected() { return connected; }

    1 usage
    public static void setConnected(boolean connected) { Connection.connected = connected; }
}
```

Figure 3.15: Connection class.

The CRUD controllers will handle requests coming of CRUD operations and send it to the node.

I will talk about the protocol of communication later in the report.

Connect Controller is responsible of handle the authentication request and send it to the master to get the port number If it is authenticated.

Figure 3.16 shows the implementation of connect controller.

```
@GetMapping(path = "/connect")
public int setConnection(@RequestParam("username") String username,
                        @RequestParam("password") String password){
    Connection.connectToMaster(username,password);
    return Connection.getNodePort();
}
```

Figure 3.16 Connect controller

3.9 Implemented Protocol between client and server

In this section I will talk about the implementation of communication protocol between client and node for all the CRUD operation and it's based on HTTP protocol.

Note: **red** color represents variable.

Creating schema:

localhost:**nodeport**/create/schema/?schemaName=**schemaName**

Creating Type:

localhost:**nodeport**/create/schema/?schemaName=**schemaName**
&typeName=**typeName**&typeSchema= **id,name,age..etc**
&typeIndex=**propertyName**

Creating Object:

localhost:8080/create/object/?schemaName= **schemaName** &typeName=
typeName &id=1&name=mohammad&...etc.

Reading Object:

localhost:8080/read/?schemaName= **schemaName** &typeName= **typeName**
&condition=**id,4**

Deleting Object:

localhost:8080/delete/?schemaName= **schemaName** &typeName= **typeName**
&condition=**id,4**

Update Object:

localhost:8080/delete/?schemaName= **schemaName** &typeName= **typeName**
&condition=**id,4**&property=**name,ahmad**

3.10 Multithreading

The Spring boot application is thread safe in terms of connection, spring boot creates new thread for every new connection.

Also, I had to make the database writing methods (Update and delete) synchronized, so when each thread wants to delete or update object based on some condition, it will be sure that this is the last update of data in the file system.

3.11 Data structures

In my implementation the most data structure is being used is the HashMap, I used it when dealing with the request parameters and when converting to/from JSON Object.

Also, I used Array List to store multiple Hash Maps each one representing document, the Array list is used when return the documents in the reading operation.

3.12 Scalability/ Consistency issues in the DB

I used round robin algorithms in the load balance so that when there are many connections they will be balanced equally. But the problem is the nodes are fixed number and if there are too many requests it cannot scale automatically.

For Consistency while number of nodes dose not scale and increase automatically there is no problem, and all the data will be the same in all nodes. But if I want to change the code and make it automatically create new nodes, the new nodes will have the new data only without the old one.

3.13 Security issues in the DB

I didn't use any encryption algorithm and the data stored as plain text in the file system.

The authentication information exposed in the http request parameters.

3.14 SOLID Principles

The Single Responsibility Principle (SRP)

The idea behind the SRP is that every class, module, or function in a program should have one responsibility/purpose in a program.

I tried to break and analyze all entities of the problem Controllers, Database IO, Index IO, Load Balance, node, cash ...etc. so each entity has single responsibility of itself, same for the methods inside entities.

Open–closed principle

"software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification"; that is, such an entity can allow its behavior to be extended without modifying its source code.

Most of the classes are open for extinction you can add new functionality to them, for example the Database IO you can add new features like deleting types and schemas for example. Or create new functionality of the Cash, Load Balancer, Node, classes and so on. But the modification of the existing code is closed.

Liskov Substitution Principle (LSP) states that objects of a superclass should be replaceable with objects of its subclasses without breaking the application.

There are no subclasses in my project implementation since most of my methods are controllers and final classes have static methods.

The Interface Segregation Principle (ISP) states that a client should not be exposed to methods it doesn't need.

I didn't use interfaces in my project and all the classes implemented there needed classes only.

The Dependency Inversion Principle (DIP) states that high-level modules should not depend on low-level modules; both should depend on abstractions.

Most of my classes are controllers and final classes have static methods. So, there is no dependency on low-level modules.

3.15 Effective Java

Creating and Destroying Object:

- **Item: Consider static factory methods instead of constructors.**

A class can provide a public static factory method, which is simply a static method that returns an instance of the class.

I used static factory methods in my classes as shown in Figure.

```
private Node(int nodeNumber){
    this.portNumber = nodeNumber;
    listOfNodes.add(this);
}
1 usage
public static Node newInstance(int nodeNumber){
    return new Node(nodeNumber);
}
```

- **Item: Enforce noninstantiability with a private constructor**

When writing classes that are just a grouping of static method and fields, it is recommended to make them non instantiable.

I enforced noninstantiability with a private constructor in Database IO and Index IO classes for example, Figure shown on of them.

```
6 usages
public final class IndexWriter {

    3 usages
    static BufferedWriter writer;
    private IndexWriter(){}
```

- **Item: Avoid creating unnecessary object.**

It is often appropriate to reuse a single object instead of creating a new functionally equivalent object each time it is needed

I avoided creating unnecessary objects by using factory method.

- **Item: Avoid finalizers and cleaners.**

Finalizers are unpredictable, often dangerous, and generally unnecessary. Their use can cause erratic behavior, poor performance, and portability problems.

I never used finalizers in my solution.

- **Item: Prefer try with resource to try finally.**

Always use try-with-resources in preference to try-finally when working with resources that must be closed.

I didn't use try-finally when working with resources that must be closed as shown in Figure.

```
try {  
    writer = new BufferedWriter(new FileWriter( fileName: "database/"+path));  
    writer.write(jsonObject);  
    writer.close();  
    return true;  
} catch (IOException e) {  
    return false;  
}
```

Classes and Interfaces:

- **Item: Minimize the accessibility of classes and members.**

A well-designed component hides all its implementation details, cleanly separating its API from its implementation.

I tried to achieve that as much as I can by making fields private and methods if it is not

used. A well good example is the connection class in Figure.

```
public final class Connection {
    2 usages
    private static int nodePort;
    2 usages
    private static boolean connected = false;

    private Connection(){}
    1 usage
    public static void connectToMaster(String username, String password) {...}

    8 usages
    public static int getNodePort() { return nodePort; }

    1 usage
    private static void setNodePort(int nodePort) { Connection.nodePort = nodePort; }

    6 usages
    public static boolean isConnected() { return connected; }

    1 usage
    private static void setConnected(boolean connected) { Connection.connected = connected; }
}
```

- **Item: In public classes, use accessor methods, not public fields.**
I used accessor methods (getters) and (setters) only to access classes fields in my project.
- **Item: Minimize Mutability**
An immutable class is simply a class whose instances cannot be modified. All the information contained in each instance is fixed for the lifetime of the object, so no changes can ever be observed.
No immutable classes used in my implementation of the project.
- **Item: Favor composition over inheritance**
Inheritance in this case is when a class extends another. Inheritance violates encapsulation
In my project I didn't use inheritance at all and used composition only when it needed, one example as shown in Figure.


```
public class NotifyController {
    5 usages
    private static final ArrayList<Node> listOfNodes = Node.getListOfNodes();
}
```

Generics:

- **Item: Don't use raw types.**

Raw type is the generic type's definition without type parameters.

I didn't need to use raw types in my code.

- **Item: Eliminate unchecked warnings.**

Eliminate every unchecked warning that you can, if you can't use Suppress-Warnings annotation on the smallest scope possible.

I tried to eliminate all the unchecked warning as much as I can in my project.

- **Item: Prefer lists to arrays.**

I tried to use lists instead of arrays as much as I can in my project, one example as shown in Figure.

```
5 usages
private static final ArrayList<Node> listOfNodes = Node.getListOfNodes();
5 usages
```

Methods:

- **Item: Return empty arrays or collections, not nulls**

There is no reason ever to return null from an array- or collection-valued method instead of returning an empty array or collection Return an immutable empty array instead of null.

In my project I always return empty list instantiated at the begin of the method as shown in Figure when there are no documents to insert in it.

```
public static ArrayList<Map<String,Object>> readAllObject(String path ){
    ArrayList<Map<String,Object>> objects = new ArrayList<> ();
```

General Programming:

- **Item: Minimize the scope of local variables.**

Declare local variable where it is first used. Most local variable declaration should contain an initializer.

You can see that I applied this in most of the methods in my project.

- **Item: Prefer for-each loop to traditional for loops.**

The Enhanced for loop officially known “for-each” gets rid of the clutter and the opportunity for error by hiding the iterator or index variable.

I applied this item in my project each time I must loop over itearables as shown in Figure.

```
for (String id : IDs){  
    objects.add(DatabaseReader.readObjectByID( path: path+"/"+id+".json"));  
}  
  
return objects;
```

- **Item: Know and use the libraries.**

By using a standard library, you take advantage of the knowledge of the experts who wrote it and the experience of those who used it before you.

In my project I widely used libraires, in different places like IO library and JSON library and so on.

```
import org.json.JSONObject;  
import java.io.BufferedReader;  
import java.io.File;  
import java.io.FileReader;  
import java.io.IOException;  
import java.util.ArrayList;  
import java.util.HashSet;  
import java.util.Map;  
import java.util.Set;
```

- **Item: Prefer primitive types to boxed primitives.**

Primitives: int, double, boolean

Boxed Primitives: Integer, Double, Boolean

I used primitives in my project and only used boxed primitives when there is a need for it.

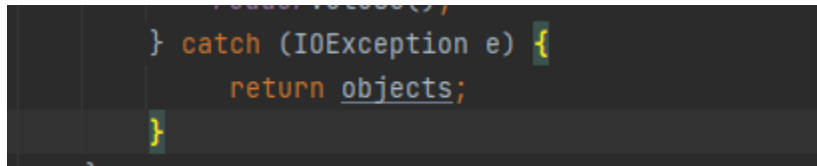
Exceptions:

- **Item: Use exceptions only for exceptional conditions.**

Exceptions are for exceptional conditions in my project I didn't use exceptions for ordinary control flow.

- **Item: Favor the use of standard exceptions.**

In the project I used only the standard exceptions. As shown in Figure.



```
} catch (IOException e) {  
    return objects;  
}
```

- **Item: Don't ignore exceptions.**

All catch blocks in my project have some logic and they are not empty.

3.16 Clean Code Principle

- Comments

1. I always tried to explain myself in the code not in the comments.
2. I always remove my comments when it gets old in my project.
3. There is no code commented, commented code always removed.
4. There is no noise in the code coming from comments.
5. Comments in my project explain the intent of what code snippet will do.

- Functions

1. Too many arguments:

Most of the methods in my project has no more than two arguments, and few has three arguments only. Figure shows example

```
public static Boolean isFileExist(String path){...}
1 usage
public static Boolean isTypeSchema (String path, JSONObject jsonObject){...}
5 usages
public static File[] getListOfFiles(String path){...}
3 usages
public static ArrayList<Map<String, Object>> readObject(String condition ,String path){...}
3 usages
public static Map <String, Object> readObjectByID(String path){...}
1 usage
public static ArrayList<Map<String, Object>> readAllObject(String path ){...}
```

2. Flag arguments:
There are no flag arguments for the methods in my project because flag arguments declares that the method do more than one thing.
3. Dead function:
All methods in my project are used, and the methods that are not used always got deleted.

- General

1. Multiple language in one file:
I didn't use more than one language in the source file.

2. Obvious Behavior Is Unimplemented:
All methods and classes implement their expected behavior
3. Incorrect Behavior at the Boundaries:
I tried to deal with all corner cases in my project to prevent any crashes and exceptions in the code.
4. Dead Code:
There is no dead code is not executed in my project
5. Vertical Separation:
I tried to make declared variables above their first usage as much as possible in my project as shown in Figure.

```
String schemaName = params.get("schemaName");
String typeName = params.get("typeName");
String condition = params.get("condition");

String typePath = schemaName+"/"+typeName;
// if there is no condition return all objects in type
if(condition == null || condition.split(regex: ",").length != 2){
    return DatabaseReader.readAllObject(typePath);
}
else{
    return DatabaseReader.readObject(condition,typePath);
}
```

6. Inconsistency:
I tried to do the same logic in my project between similar things as example all the CRUD operation have the same style.
7. Clutter:
I didn't leave any clutter code in my project like not used methods and variables and meaningless comments.
8. Selector Arguments:
I didn't use selector arguments to make the methods only do one thing.
9. Function Names Should Say What They Do:
I tried to carefully pick my methods names to make them explain their self as much

as possible.

10. Replace Magic Numbers with Named Constants:

I didn't use magic numbers in my code I refer to numbers in constants like the cash max size and the min and max port numbers for the nodes.

11. Functions Should Do One Thing:

I tries to force my methods to follow the single responsibility principle as much as I can in my project and make them do one thing.

- **Names:**

I choose names that describe my variables and methods very carefully.

I tried to make meaningful distinctions between variables and methods.

I tried to make pronounceable names as much as I can.

Class names have nouns.

Methods names have verbs.

3.17 Design patterns

Singleton is a creational design pattern that lets you ensure that a class has only one instance.

I used singleton design patter in the cash because I have only one cash in the node.

I used the Observer design pattern So when the master wants to notify the nodes it will act as a (subject) and loop through list of the nodes(observer) to update them with the new data.

I used Spring boot to build the database parts, which is MVC framework, MVC design pattern used to separates the data model, presentation information, and control information from each other.

3.18 DevOps

I used Docker to create new nodes in the docker network. Figure 3.17 shows the docker file of the node application.

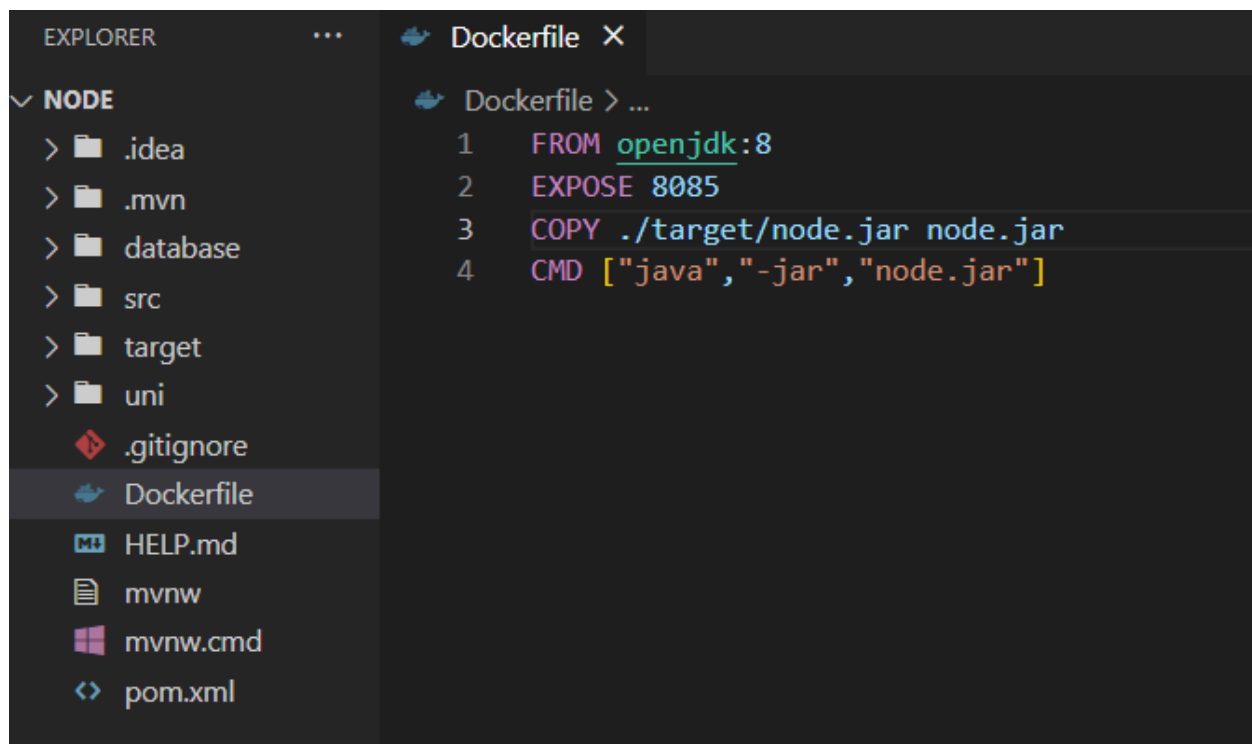


Figure 3.17: Node docker file.

Chapter 4 Demo Application.

I created demo application. The demo is web app used to make crud operations on school schema to add and manipulate student information. Figure 4.1 shows the classes and views I made for the demo application

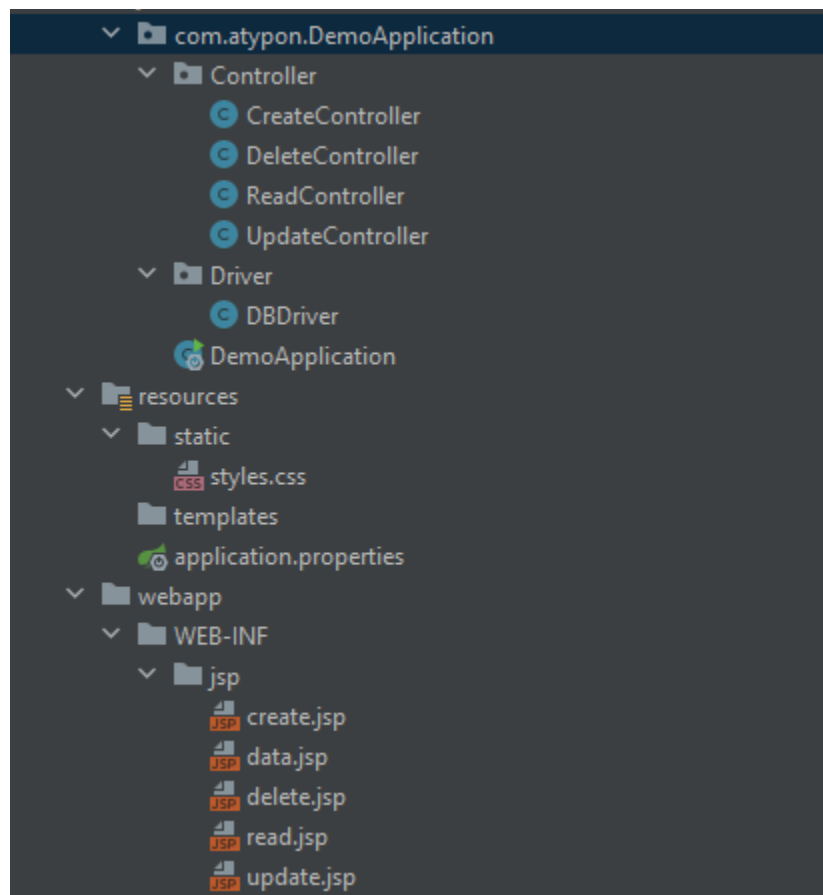


Figure 4.1 Demo Application

The DB Driver class is used to implements API calls to the database client. So, whenever new client is instantiated from the driver class, it can manipulate and connect with database.

Figure 4.2 show the implementation of DB Driver.

```

12 usages
public class DBDriver {
    4 usages
    public DBDriver(String username, String password){
        String uri = "http://localhost:8081/connect/?username="+username+"&password="+password;
        RestTemplate restTemplate = new RestTemplate();
        Integer result = restTemplate.getForObject(uri, Integer.class);
    }

    4 usages
    public String createSchema(String schemaName){...}
    4 usages
    public String createType(String schemaName , String typeName){...}
    1 usage
    public boolean createObject(String schemaName , String typeName ,HashMap<String,String> properties) {...}
    1 usage
    public boolean updateObject(String schemaName , String typeName, String condition , String property){...}
    1 usage
    public boolean deleteObject(String schemaName , String typeName, String condition){...}
    1 usage
    public ArrayList readObject(String schemaName , String typeName, String condition){...}

```

Figure 4.2 DB Driver implementation

And the Figure 4.3 show creating new database client object.

```

3 usages
DBDriver client = new DBDriver( username: "admin", password: "admin");

```

Figure 4.3: database client object

And for the views I created a view for each CRUD operation as shown in Figures 4.4, 4.5, 4.6,.

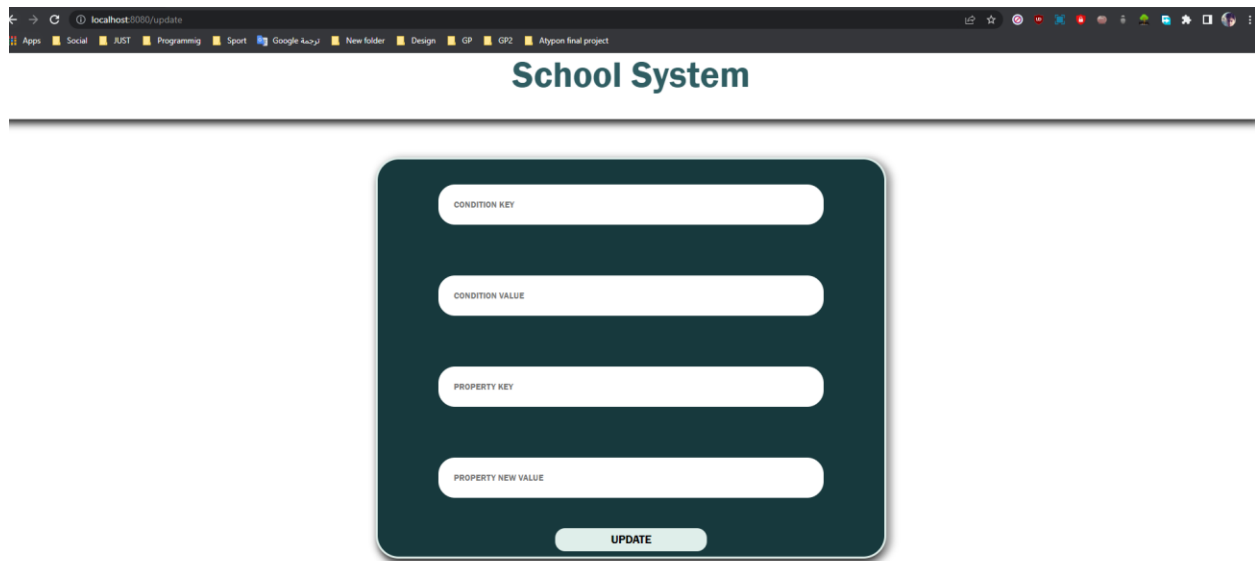
The screenshot shows a web browser window with the address bar at `localhost:3080/create`. The page title is "School System". Below the title is a dark green rounded rectangle containing a form with five white input fields stacked vertically, labeled "STUDENT ID", "STUDENT NAME", "STUDENT MAJOR", "STUDENT YEAR", and "STUDENT GPA". At the bottom of this rectangle is a light green button labeled "INSERT".

Figure 4.4 create student

The screenshot shows a web browser window with the address bar at `localhost:3080/read`. The page title is "School System". Below the title is a dark green rounded rectangle containing a form with two white input fields stacked vertically, labeled "CONDITION KEY" and "CONDITION VALUE". At the bottom of this rectangle is a light green button labeled "READ".

Figure 4.5 Read student

Delete view will have the same UI as read view.



The screenshot shows a web browser window with the address bar displaying 'localhost:3000/update'. The browser's tab bar includes several open tabs: 'App', 'Social', 'JST', 'Programming', 'Sport', 'Google', 'New folder', 'Design', 'GP', 'GP2', and 'Atypen final project'. The main content area of the browser displays a web page titled 'School System' in a large, bold, dark green font. Below the title is a horizontal line. Centered on the page is a dark green rounded rectangle containing four white input fields stacked vertically. The labels for these fields are 'CONDITION KEY', 'CONDITION VALUE', 'PROPERTY KEY', and 'PROPERTY NEW VALUE'. At the bottom of this green rectangle is a light green button with the text 'UPDATE' in dark green capital letters.

Figure 4.5 update student