

ATYPON

Atypon internship program

Final project

NoSQL Database

Prepared by:

Ahmad Drabkah

February-2022

Table of Contents

1. Introduction.....	2
2. Implementation	4
3. Load Balancing	11
4. LRU Cache	14
5. Scalability and Consistency	19
6. Multithreading	21
7. Clean code	24
8. Effective Java	38
9. SOLID Design principles	49
10. Design patterns	52
11.DevOps	56

1. Introduction

1.1 NoSQL database overview

A document database is a type of NoSQL database, which stores data as JSON documents instead of columns and rows. JSON is a native language used to both store and query data. These documents can be grouped together into collections to form database systems.

Each document consists of a number of key-value pairs, using JSON enables app developers to store and query data in the same document-model format that they use to organize their app's code. The object model can be converted into other formats, such as JSON, BSON and XML.

Document databases provide fast queries, a structure well suited for handling big data, flexible indexing and a simplified method of maintaining the database. It is efficient for web apps and has been fully integrated by large-scale IT companies.

Although SQL databases have great stability and vertical power, they struggle with super-sized databases. Use cases that require immediate access to data, such as healthcare apps, are a better fit for document databases. Document databases make it easy to query data with the same document-model used to code the application.

1.2 Requirements

In this project, we are required to build a NoSQL database application with ability to do all CRUD operations, ability to handle request of multiple users and all the write operation (creation of a schema, add type to a schema, add object of specific type to schema or delete object of specific type from the schema) must be handled and done only by the master controller which is a single point of failure.

In our design, we must take care about OOP principles, SOLID Design principle, Design patterns, and Effective Java and race conditions.

In the following pages, I am going to discuss my design, thoughts and all technologies I used in the project implementation.

2. Implementation

2.1 Introduction

In this section, I will talk about the project implementation, data structures used, database storage way and organization, cache and indexing, load balancing and protocol for communication between user and server.

In general, the project is consist of the separated Java application one for master controller, node and client, which are implemented using web socket, and streams to manage the communication protocol; in the following pages, I will discuss them in details.

2.2 Database architecture

My architecture of the database divide into three main part schema, types and objects. Each schema consist of multiple types, the types consist of set of keys and values and each object is belong to one type in other word it must follow its structure so the object is the actual data off the types (assign values to the key set).

I organized the storage of the database like that:

- Schema will be a directory that contains all types belong to it.
- Each type of belong to the schema is a sub directory inside its directory.
- Each object is a text file inside the type, which belong to it, the file store the actual JSON object and the file name is the object ID.

In this way of organization, I can easily access database folders and files, like even when I need to read data from the HDD when it is not in the cache I do not need to search in folders or files.

2.3 Master Controller

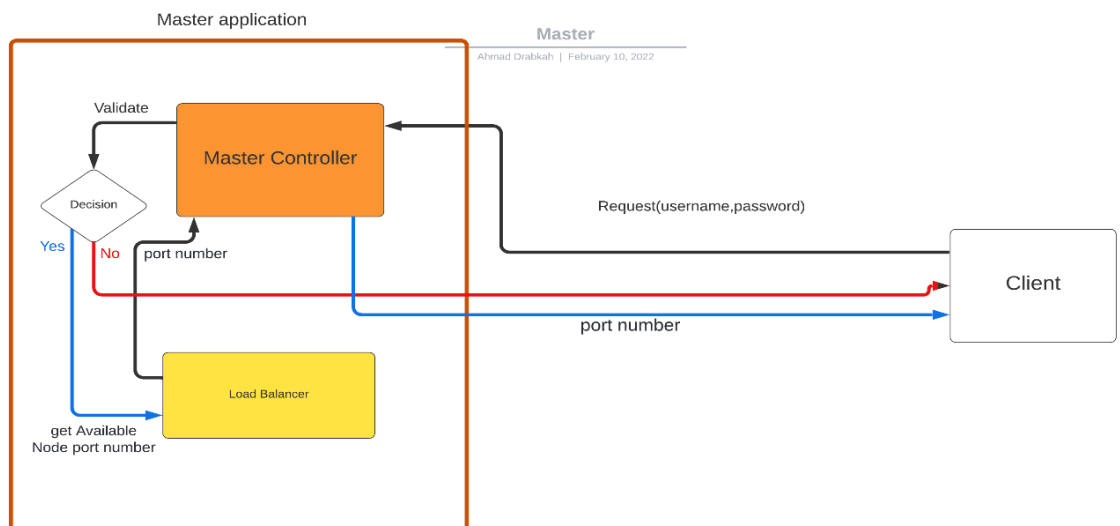
This application is the main point of the whole project it control all the write processes done and connect client to the node.

The application start by opening web socket on a specific port number waiting for a client to connect to it. I used ExecutorService interface to handle multiple client each one in a thread.

```
public MasterController(){
    loadUsers();
    try {
        ServerSocket serverSocket = new ServerSocket( port: 8100);
        ExecutorService executorService = Executors.newCachedThreadPool();
        while (true){
            Socket socket = serverSocket.accept();
            executorService.execute(new Task(socket));
        }
    }
    catch (Exception e){
        e.printStackTrace();
    }
}
```

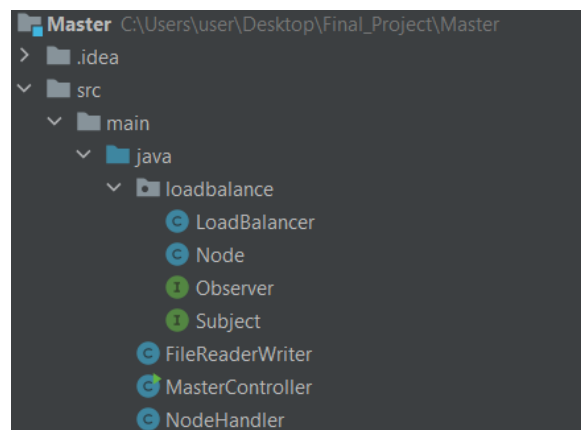
The benefit of used cached thread pool that it will handle create and destroy the threads by its own so there is no need to manage request and create waiting queue.

The **Task** class is a local private class that implement interface Runnable so it can run concurrently, it determined if the connection is from client or from node to direct it to the proper method as described in the following flow chart.



After client connect to master, it will validate from his credentials using a HashMap of predefined users (loaded from file on the HDD when the application start), after that if the client authorized then it will send a port number of available node(get it from load balancer, I will describe it later) to connect to it, in order to handle his request.

The master controller is responsible about load balancing for the clients to the node and all write operation so I managed these operations using the following classes



- I. The **File reader writer** class handle all operation done to a file reading, writing...etc. In addition, this class is build using singleton design patterns (will discuss later in Design patterns section) to get only one object using to by all threads so the management of race condition could be easier.

```
public class FileReaderWriter {
    private final String PATH;
    private static volatile FileReaderWriter dataReaderWriter = null ;

    private FileReaderWriter(String path) { this.PATH=path; }
    public static FileReaderWriter getInstance(String path){...}
    public synchronized List<JSONObject> readDataFromFile(String directoryName, String fileName){...}
    public synchronized void createDirectory(String directoryName){...}
    public synchronized void createSubDirectories(String directoryName , List<String> subDirectoriesNames){...}
    public synchronized void createFile(String directoryName, String fileName) throws IOException {...}
    public synchronized void writeDataToFile(String directoryName, String fileName, String data){...}
    public synchronized void deleteFile(String directoryName, String fileName){...}
}
```

- II. The **Node handler** class is responsible to handle node request which is request the master to do a write operation create schema, add new type to schema ...etc.

The main method of this class is **handleNodeOperation**, which is determined node operation, and call function that is responsible for this operation the **operation** and **data** parameters is read by master when node connect to it.

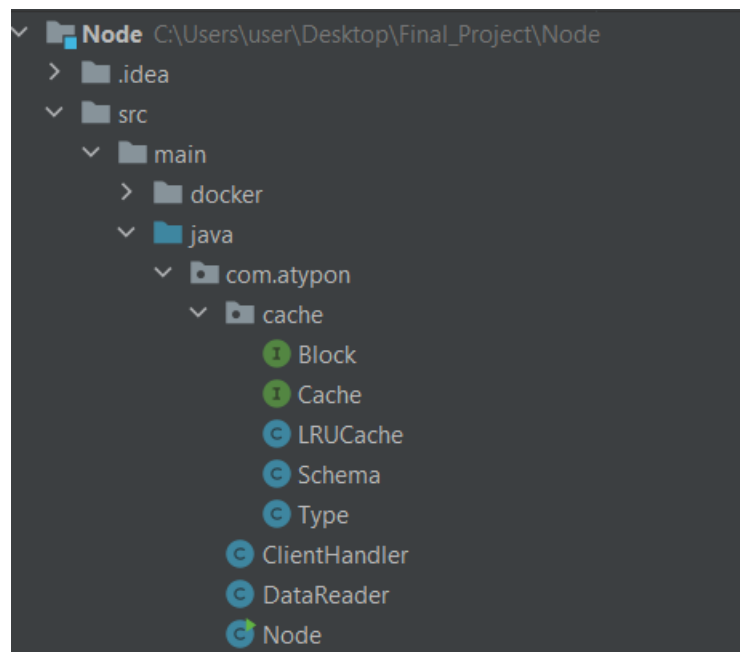
```
public class NodeHandler {  
    private final FileReaderWriter dataReaderWriter = FileReaderWriter.getInstance(  
        "C:\\Users\\user\\Desktop\\Final_Project\\Database"  
    );  
  
    public void handleNodeOperation(String operation, HashMap<String, String> data ) throws IOException {...}  
    private void createSchema(HashMap<String, String> schemaInfo) throws IOException {...}  
    private void addObjectType(HashMap<String, String> schemaInfo ) throws IOException {...}  
    private void createTypesFile(String schemaName, HashMap<String, String> schemaInfo) throws IOException {...}  
    private void addObject(HashMap<String, String> objectInfo ) { createObjectFile(objectInfo); }  
    private void updateObject(HashMap<String, String> objectInfo ){...}  
    private void createObjectFile(HashMap<String, String> objectInfo){...}  
    private void deleteObject(HashMap<String, String> objectInfo) { deleteObjectFile( objectInfo); }  
    private void deleteObjectFile(HashMap<String, String> objectInfo){...}  
}
```

The protocol for communication between node and master is to send first word “node” so the master know it is not a client then sent the operation to be did then send HashMap contain data to be written so the master read it and send to the Node handler.

2.4 Node application

This application represent the backend server, its responsibilities is to handle the client request read and write operations, the read operation is done locally from the cache but if the operation is write, the node collect user information necessary for this operation (like in create schema it need schema name, types names and keys set ..Etc) and then send it to the master controller using the protocol discusses earlier.

This application is replicated and scaled on each server (Docker container) with capability to handle multiple users at the same time.



- I. The main class in this application is **Node** class, which open the socket and accept the client request then redirect it to the handler.

The app start by initiate of open a server socket on a specific port (that will mapped by load balancer when run the container) to handle and accept user request as Master application I used cached thread pool to tack the advantage of creating and destroying threads automatically.

```

public static void main(String[] args){
    System.out.println("Node start !!");
    new Node();
}
public Node(){
    try {
        ServerSocket serverSocket = new ServerSocket( port: 5000);
        ExecutorService executorService = Executors.newCachedThreadPool();
        while (true){
            Socket socket = serverSocket.accept();
            executorService.execute(new Task(socket));
        }
    } catch (IOException e) {
        e.printStackTrace();
    }
}
}

```

- II. Class **Task** is the same as described in the Master application it implement Runnable interface so it can execute concurrent task.
- III. The **FileReader** class is responsible for all file reading operation, it was built using singleton design patterns so the race conditions can manage easily and carefully as this class cloud be used by multiple thread at the same time.

```

public class FileReader {
    private final String PATH ;
    private static volatile FileReader dataReader=null;

    private FileReader(String path) { this.PATH=path; }
    public static FileReader getInstance(String path){...}
    public synchronized File[] getAllSubDirectories(String directoryName){...}
    public synchronized HashMap<String, JSONObject> getAllFilesDataInDirectory(String directoryName){...}
    public synchronized JSONObject readDataFromFile(String directoryName, String fileName){...}
    public synchronized boolean isFileExist(String directoryName,String fileName) {...}
}

```

- IV. The **ClientHandler** class is responsible for handle all clients requested operation read or write, if the operation is read then it will get the data locally from the cache and send it to the user, but if the operation is write then it will collect the necessary data for performing the requested operation in a

HashMap then it will send the operation name and map for the master in order to perform this operation.

The data scalability in the node, caching and update cache status will discuss later in the Cache section.

2.5 Client Side

This is part is interacted with user operation by read data from the user and show him the result of his requested operation it could be ether web application or command line.

This side should be contain a web socket that initiate the connection with master to login and get port number for one of the nodes.

As described in the master application the protocol of communication will be like the following:

- Client create web socket to connect to the master the send a word “Client” so the master can distinguished that it’s a client not node and then send the user name and password.
- Master will validate of client authentication, then if it is authorized master will replay by “true” indicted that the client is authorized then send available node port number, otherwise it will replay by “false” and waiting for enter username and password again.
- After client received the port number, it will close the connection with master and connect to the node.

3. Load balancing

3.1 Introduction

Load balancing is a core networking solution used to distribute traffic across multiple servers in a server farm. Load balancers improve application availability and responsiveness and prevent server overload. Each load balancer sits between client devices and backend servers, receiving and then distributing incoming requests to any available server capable of fulfilling them.

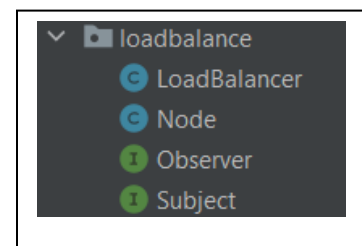
Modern applications and websites cannot function without balancing the load on them. This is for the reason that such applications and sites serve millions of simultaneous requests from the end-users and have returned the correct text and images or the related data asked for, responsively, and reliably. Adding more servers was considered good practice for meeting such high volumes of traffic, so far.

However, the concept of balancing the load with a dedicated Load Balancer unit is a much more economical and effective way of ensuring the peak performance of the website or application and offering the end-user a great experience.

3.2 Implementation

My idea for doing this job was to make the master create a node by run Docker container when it is needed. At any time load balancer will have an available node when there is a request for available node it will check the current node if it is available, if it is then return the current node otherwise it will create a new one.

The master does this job using the following classes and interfaces as **Observer** and **Subject** interface used to implement Observer design pattern, which I will describe later.



The **LoadBalancer** class manage this operation by keep a pointer of type node for the available node **currentAvailableNode**, which is the only node that is not full and can handle a client connection.

```
public class LoadBalancer implements Subject {
    private final List<Observer> nodes = new ArrayList<>();
    private Node currentAvailableNode;

    public LoadBalancer() { createNode(); }
    public synchronized Observer getAvailableNode(){...}
    private void createNode(){...}
    private boolean isCurrentNodeFull(){...}
    @Override
    public void register(Observer observer) { nodes.add(observer); }
    @Override
    public void unregister(Observer observer) { nodes.remove(observer); }
    @Override
    public void notifyUpdate() {...}
}
```

In the method, **getAvailableNode** check if the node reach its maximum capacity, if the node is full then it will create new node using class **Node** (that will describe now) and register it in the node list.

This method called from master after validate the client to send the port number so client can connect to it.

```
public Observer getAvailableNode(){
    if(isCurrentNodeFull())
        createNode();
    currentAvailableNode.incrementNumberOfConnections();
    return currentAvailableNode;
}
private void createNode(){
    Node node = Node.nodeFactory();
    currentAvailableNode = node;
    register(node);
}
```

The **Node** class is represent node in the system by store the node IP, port number and number of connected clients, it is responsible for create the Docker container which run the node (will describe in DevOps section) and notify node when data is write so nodes know that there caches is dirty (will describe more in Cache section).

```
public class Node implements Observer{
    private final String ip = "localhost";
    private int portNumber;
    private int numberOfConnections=0;

    public static Node nodeFactory(){...}
    private static int createContainer(){...}
    private static int generateRandomPortNumber(){...}
    public int getPortNumber() { return portNumber; }
    public void setPortNumber(int portNumber) { this.portNumber = portNumber; }
    public int getNumberOfConnections() { return numberOfConnections; }
    public void incrementNumberOfConnections(){numberOfConnections++;}
    @Override
    public void update() {...}
}
```

The **nodeFactory** method is responsible for create a node it use **createContainer** which in turn create the container which run the node after generate a random port number for it.

I used Java runtime so I can run Docker command by create process and run it.

```
private static int createContainer(){
    String volumePath = "//c/Users/user/Desktop/Final_Project/Database";
    int portNumber = generateRandomPortNumber();
    try {
        String[] command = {"docker", "run", "-t", "--volume",
            volumePath+":/database", "-p", portNumber+":5000", "node:v1"};
        ProcessBuilder pb = new ProcessBuilder(command);
        pb.inheritIO();
        Process proc = pb.start();
    } catch (IOException e) {
        e.printStackTrace();
    }
    return portNumber;
}
```

4. Least recently used(LRU) Cache

4.1 Introduction

Cache is a small memory, fast access local store where we store frequently accessed data. Caching is the technique of storing copies of frequently used application data in a layer of smaller, faster memory in order to improve data retrieval times, throughput, and compute costs.

4.2 How cache work

When a request comes to a system, there can be two scenarios. If a copy of the data exists in cache it has called a **cache hit**, and when the data has to be fetched from the primary data store it has called a **cache miss**. The performance of a cache is measured by the number of cache hits out of the total number of requests.

4.3 How LRU cache work

The Least Recently Used (LRU) cache is a cache eviction algorithm that organizes elements in order of use. In LRU, as the name suggests, the element that has not been used for the longest time will be evicted from the cache.

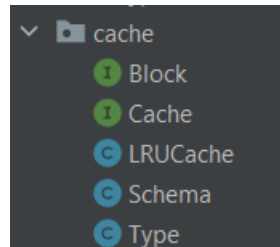
4.4 Implementation

As described in the way LRU working I need to implement it a doubly queue (Deque) and HashMap to store the object and there keys. In my database architecture there is three level of files schema folders, each one contain type's folders and each type folder contain object files so I need three level of HashMap to store this hierarchy of folders.

First the Deque is used to store the blocks keys in order they are used, as if the block access then it will added to the front of the queue, so at last the least used block will be in the last position of the queue.

The HashMap store the blocks name as keys and the block object as value.

I implement this approach using the following classes and interfaces



Cache interface is the supper type of each class that implement any cache algorithm it contains two methods, which must be implemented by any method **getBlock** and **update** which I will discuss them in **LRUCache** class. **Block** interface is the supper type of each value in the HashMap of the cache, in both these interface I considered a high level of abstraction so the system will be open for any extension of the cache for implementing any cache algorithm, and I will describe this more in SOLID design principles.

```
public interface Cache {  
    Block getBlock(String blockName);  
    void update();  
}
```

```
public interface Block {  
    Object getBlockData();  
}
```

LRUCache class is the implementation of the least recently used cache, which I described above. It is implemented **Cache** interface by override the two methods **getBlock** and **update** to apply the LRU algorithm with using of local helper function.

The max number of schemas and objects is represent the amount of data that the cache can hold at a certain time depending on the resource that provided to the application. I implement cache using

singleton design pattern, as it should be only one instance in the whole application.

```
public class LRUCache implements Cache{
    private final int MAX_SCHEMA_NUMBER = 2;
    private final int MAX_Object_NUMBER = 2;
    private boolean isDirty = false;
    private final String PATH = "/database";
    private final Map<String, Block> schemas ;
    private final Deque<String> LRUQueue;
    private final FileReader dataReader = FileReader.getInstance(PATH);
    private final static LRUCache LRU_CACHE = new LRUCache();
    private LRUCache(){...}
    public static LRUCache getInstance() { return LRU_CACHE; }
    @Override
    public Block getBlock(String blockName) {...}
    private void moveBlockToFront(String schemaName){...}
    private synchronized boolean isBlockExist(String schemaName){...}
    private void uploadToCache(String schemaName){...}
    private synchronized Block bringBlock(String blockName){...}
    @Override
    public void update(){...}
    private void dropCache(){...}
}
```

The method **getBlock** is the entry point of the cache as it return the data required from it if it is exist on it or it is exist in HDD. In case it contains the required data, it will move its key to the front of **LRUQueue** using **moveBlockToFront** method as indicate this data has been accessed recently. In case the data is not in the cache, it will search for it in the HDD and bring it using **bringBlock** method otherwise it will return null as indication the data is not exist.

```

public Block getBlock(String blockName) {
    synchronized (schemas) {
        synchronized (LRUQueue) {
            if(schemas.containsKey(blockName)){
                moveBlockToFront(blockName);
                return schemas.get(blockName);
            }
            if(isBlockExist(blockName)){
                uploadToCache(blockName);
                return schemas.get(blockName);
            }
        }
    }
    return null;
}

```

The method **update** will scale data as if the master send to the node that the data has been modified then the class **Handler** class will call it so it will make cache status dirty and drop the data that is in the cache, this operations will discuss more in the Scalability and Consistency which is the next section.

```

public void update(){
    cache.update();
    try {
        inputFromUser.close();
        outputToUser.close();
    } catch (IOException e) {
        e.printStackTrace();
    }
}

```

Schema class is an implementation of **Block** interface so it could be a value in the cache class, it contains a second HashMap that represents each type name as a key and object of **Type** class will discuss next.

```

public class Schema implements Block{
    private final HashMap<String, Type> schemaTypes;

    public Schema(HashMap<String, Type> schemaData){...}
    @Override
    public Object getBlockData() { return schemaTypes; }
    public Type getType(String typeName){...}
    public boolean containsType(String typeName){...}
}

```

Type class is a representation of schema type contains a HashMap of object ID as key and the JSON object as a value. In this class I applied the algorithm of LRU for the objects the idea is that when the client access an object not in the cache only the LRU object will be replaced no the whole schema as the cost of bring object from HDD is much less than cost of bring schema.

```

public class Type {
    private final Map<String, JSONObject> objects = new HashMap<>();
    private final Deque<String> LRUObject = new LinkedList<>();
    private final String SCHEMA_NAME;
    private final String TYPE_NAME;
    private final int MAX_SIZE;

    public Type(String schemaName, String typeName, int maxSize) {...}
    public JSONObject getObject(String id){...}
    private void moveObjectToFirst(String id){...}
    private JSONObject getObjectIfExists(String id){...}
    public void addObject(String id, JSONObject object){...}
}

```

5. Scalability and Consistency

5.1 Introduction

Database scalability is the ability of a database to handle changing demands by adding/removing resources. There is two type of scaling horizontal and vertical we will consider the horizontal.

Data consistency means that the changes made to the different occurrences of data should be controlled and managed in such a way that all the occurrences have the same value for any specific data item.

5.2 Scalability

In this project, I used a horizontal scalability in order to handle the increasing user requests. To achieve this goal I used load balancing technique and Docker as the container represent the servers and they will create and managed by the load balancer in the Master. Any time there worker container is full the load balancer will create a Docker container to handle the new requests as I discussed in the load balancer section, each container will have its own copy of the data in his cached as the data will be replicated to each container(Node).

In this manner, the scalability of the system at any point of time is guaranteed.

5.3 Consistency

As I defined the consistency, the data should be at a valid state at any time and all nodes should read the same data. This is achieved by make all operations that change the database state (write operations) done only from one point which is the Master as it is the only part of the application could change the database state. For the second condition of constancy, which is that all node should read the same data, I achieved it by notify the worker nodes that the load balancer kept its information (IP, port number and number of connected user) in as list of object of type **Node** class in Master.

After any write operation done by the master it call the load balancer method **notifyUpdate** which iterate over the list of nodes and call method **update** from **Node** class, this method will connect to the node and inform it by the update by sending word “update” . The node will know that the database state has been updated so it will drop its local copy of data (cache) and then read any data requested from the disk and store it in the cache.

```
public void nodeOperation() throws IOException, ClassNotFoundException {
    String operation = (String)input.readObject();
    HashMap<String, String> data = (HashMap<String, String>)input.readObject();
    NodeHandler nodeHandler = new NodeHandler();
    nodeHandler.handleNodeOperation(operation,data);
    loadBalancer.notifyUpdate();
}
```

NodeHandler in Master project

```
public void notifyUpdate() {
    for (Observer node: nodes) {
        node.update();
    }
}
```

LoadBalancer in Master project

```
public void update(){
    cache.update();
    try {
        inputFromUser.close();
        outputToUser.close();
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

Handler in Node project

```
public void update() {
    try{
        Socket socket = new Socket(ip,portNumber);
        ObjectOutputStream outputStream
            = new ObjectOutputStream(socket.getOutputStream());
        outputStream.writeObject("update");
        outputStream.close();
        socket.close();
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

Node in Master project

```
public void update(){
    isDirty = true;
    dropCache();
}
```

LRUCache in Node project

6. Multithreading

6.1 Introduction

Multithreading is a model of program execution that allows multiple threads to be created within a process, executing independently but concurrently sharing process resources. Depending on the hardware, threads can run parallel if they are distributed to their own CPU core.

6.2 Implementation

The project is built using multithread technique in order to handle multiple user requests at the same time. Firstly, the **MasterController** class in the master application and **Node** class in the node application is start with initiate cached thread pool, which used to handle user request each one in separated thread.

```
public MasterController(){
    loadUsers();
    try {
        ServerSocket serverSocket
            = new ServerSocket( port 8100);
        ExecutorService executorService
            = Executors.newCachedThreadPool();
        while (true){
            Socket socket = serverSocket.accept();
            executorService.execute(new Task(socket));
        }
    }
    catch (Exception e){
        e.printStackTrace();
    }
}
```

```
public Node(){
    try {
        ServerSocket serverSocket
            = new ServerSocket( port 5000);
        ExecutorService executorService
            = Executors.newCachedThreadPool();
        while (true){
            Socket socket = serverSocket.accept();
            executorService.execute(new Task(socket));
        }
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

The **Task** class is implement Runnable interface, in both classes it handle the client request and direct execute the requested operation.

With cached thread pool there is no need for queueing the requests and threads as the pool itself increase and decrease number of threads automatically, as if it is full and there is new request it will

create new thread to serve it and in case there is a thread that not used for long time it will remove it.

6.3 Synchronization

In order to grantee the result of concurrently execution synchronized are used, as I used it whenever thread change the state of shared data.

First placed to be synchronized is the **FileReaderWriter** class as all of its methods that write to a file, create directory or create file must be mutually exclusion so there is no thread will read from the file that being write to it.

```
public class FileReaderWriter {
    private final String PATH;
    private static volatile FileReaderWriter dataReaderWriter = null ;

    private FileReaderWriter(String path) { this.PATH=path; }
    public static FileReaderWriter getInstance(String path){...}
    public List<JSONObject> readDataFromFile(String directoryName, String fileName){...}
    public synchronized void createDirectory(String directoryName){...}
    public synchronized void createSubDirectories(String directoryName , List<String> subDirectoriesNames){...}
    public synchronized void createFile(String directoryName, String fileName) throws IOException {...}
    public synchronized void writeDataToFile(String directoryName, String fileName, String data){...}
    public synchronized void deleteFile(String directoryName, String fileName){...}
}
```

As shown above all methods that change the state of the database are synchronized.

In the Node application the cache operation that change the **HashMap schemas** and **LRUQueue** must be synchronized so the cache state is consistence at any time.

```
private void moveBlockToFront(String schemaName){
    synchronized (LRUQueue){
        LRUQueue.remove(schemaName);
        LRUQueue.addFirst(schemaName);
    }
}
```

```
private void dropCache(){
    if(isDirty){
        synchronized (schemas){
            synchronized (LRUQueue){
                schemas.clear();
                LRUQueue.clear();
            }
        }
    }
}
```

```
private void uploadToCache(String schemaName){
    synchronized (schemas){
        synchronized (LRUQueue){
            if(schemas.size() == MAX_SCHEMA_NUMBER){
                String leastUsingSchema = LRUQueue.removeLast();
                schemas.remove(leastUsingSchema);
            }
            Schema schema = (Schema) bringBlock(schemaName);
            LRUQueue.addFirst(schemaName);
            schemas.put(schemaName, schema);
        }
    }
}
```

As seen in the above pictures I dealt with any operation change shared data carefully tacking care about the risk of concurrent execution.

7. Clean Code

7.1 Introduction

In this section, I am going to discuss clean code principles and code smells defined by Robert C. Martin.

"Any fool can write code that a computer can understand. Good programmers write code that humans can understand." – Martin Fowler.

From the above quotation, we can know that writing clean, understandable, and maintainable code is a skill that is crucial for every developer to master.

7.2 Code smells

7.2.1 Comments

This section is talk about the comments, it is important to clarify and justify the code, and it is introduce five role for writing the comments like “It is inappropriate for a comment to hold information better held in a different kind of system such as your source code”. That what I did in my code as I give a meaningful name for the variable instead of write comment describe it and implement the algorithm in many steps so the code and functions descript itself instead of writing comments that explain the algorithm.

7.2.2 Environment

1. Building require more than one step
“Building a project should be a single trivial operation. You should not have to check many little pieces out from source code control.”

This rule was applied by using maven to build the project and used to build the Docker image, so that maven will build the whole project.

2. Test require more than one step

“You should be able to run all the unit tests with just one command. In the best case you can run all the tests by clicking on one button in your IDE”

I didn't write any test case or using JUNIT testing as the system is more than one application communicate with each other, instead of that I tested the system manually.

7.2.3 Functions

1. Too many arguments

“Functions should have a small number of arguments one, two or three”

This principle is applied for all methods in the project, as no method tack more than two arguments.

2. Output argument

“If your function must change the state of something, have it change the state of the object it is called on”

This principle is applied for all methods in the project, as I did not use any output argument.

3. Flag argument

“Boolean arguments loudly declare that the function does more than one thing.”

This principle is applied for all methods in the project, as I implement all methods in single responsibility principle so every method do only one job.

4. Dead function

“Methods that are never called should be discarded. Keeping dead code around is wasteful”

This principle is applied for all methods in the project, as all method implemented are used in the code.

7.2.4 General

1. Multiple language in one file

“The ideal is for a source file to contain one, and only one, language”

This principle is applied for all source files in the project, as I did not use more language per file.

2. Obvious behavior unimplemented

“Any function or class should implement the behaviors that another programmer could reasonably expect”

This principle is applied for all classes and methods as they are implement the behavior expected by the client.

3. Incorrect behavior in the boundaries

“Every boundary condition, every corner case, every quirk and exception represents something that can confound an elegant and intuitive algorithm”

This principle is applied for all methods as they are implemented with care about boundaries and unexpected situations.

```
public void deleteObject() throws IOException, ClassNotFoundException {  
    HashMap<String, String> objectInfo = readObjectInfoFromUser();  
    if(objectInfo != null) {  
        sendDataToMaster( operation: "deleteObject", objectInfo);  
    }  
}
```

4. Duplication

“Every time you see duplication in the code, it represents a missed opportunity for abstraction. That duplication could probably become a subroutine or perhaps another class outright”

This principle is applied for all methods as there is no duplication in the code and the code that has been used multiple times it is refactored to a method. Like read from user method below it has been used multiple times in the code.

```
public String readFromUser(String message) throws IOException, ClassNotFoundException {  
    outputToUser.writeObject(message);  
    String input = (String)inputFromUser.readObject();  
    return input;  
}
```

5. Code at wrong level of abstraction

“It is important to create abstractions that separate higher level general concepts from lower level detailed concepts”

This principle is applied in a good manner as I implement abstraction when needed, like the cache interface is the super type of any class that implements cache algorithm and it does not know anything about subtypes details.

```
public interface Cache {  
    Block getBlock(String blockName);  
    void update();  
}
```

6. Base classes depending on their derivatives

“Higher level base class concepts can be independent of the lower level derivative class concepts”

This principle is applied for all super types as they do not know anything about the subtypes, like in cache and block interface they do not know about LRUCache and Schema classes that implement them.

```
public interface Cache {  
    Block getBlock(String blockName);  
    void update();  
}
```

```
public interface Block {  
    Object getBlockData();  
}
```

7. Too many information

“Well-defined modules have very small interfaces that allow you to do a lot with a little.”

This principle is applied for all classes as they implement only the necessary methods and depend on the necessary classes, like load balancer class depend only the Node class that is necessary for creating new node.

8. Dead code

“Dead code is code that isn’t executed. You find it in the body of an if statement that checks for a condition that cannot happen. You find it in the catch block of a try that never throws. You find it

in little utility methods that are never called or switch/case conditions that never occur.”

This principle is applied for the whole project as there is no code that never used or in a block that never reached.

9. Vertical separation

“Variables and function should be defined close to where they are used”

This principle is applied for the whole project as all methods defined in the nearest distance from they are used.

```
public void update(){  
    isDirty = true;  
    dropCache();  
}  
private void dropCache(){...}
```

10. Inconsistency

“If you do something a certain way, do all similar things in the same way”

This principle is applied for the whole project as I used the same name for the same variable declared in different method and use the same name for the methods do similar job in different class.

11. Clutter

“Variables that aren’t used, functions that are never called, comments that add no information, and so forth”

This principle is applied for the whole project as there is no variable or function that never used and I do not used comments.

12. Artificial coupling

“Things that don’t depend upon each other should not be artificially coupled.”


I tried my best to applied this principle for all classes in the project and take time of thinking where to declare the variables and which classes depend on each other.

13. Features envy

“The methods of a class should be interested in the variables and functions of the class they belong to, and not the variables and functions of other classes”

I did not figure out where should I used this principle, but in general most of the methods deal with the methods and variable of their class, in case it need to use another class it use a specific method from it. Like the following show object method is in class Client handler and it used object cache to get a specific block.

```
public void showObject() throws IOException, ClassNotFoundException {
    HashMap<String, String> objectInfo = readObjectInfoFromUser();
    if(objectInfo != null) {
        String schemaName = objectInfo.get("schemaName");
        String typeName = objectInfo.get("typeName");
        String id = objectInfo.get("id");
        Schema schema = (Schema)cache.getBlock(schemaName);
        JSONObject object = schema.getType(typeName).getObject(id);
        outputToUser.writeObject(object.toJSONString());
    }
}
```



14. Selector arguments

“Each selector argument combines many functions into one. Selector arguments are just a lazy way to avoid splitting a large function into several smaller functions”

This principle is applied for all methods, as there is no method implement with argument that control its behavior.

15. Miss responsibility

“Sometimes we get “clever” about where to put certain functionality. We’ll put it in a function that’s convenient for us, but not necessarily intuitive to the reader”

I tried my best to applied this principle for all classes and methods and to be compatible with single responsibility principle.

16. Inappropriate static

“In general you should prefer nonstatic methods to static methods. When in doubt, make the function nonstatic. If you really want a function to be static, make sure that there is no chance that you’ll want it to behave polymorphically”

I applied this principle for the most of methods, as they are non-static and I used static method only when I really need it. Like the following I need node factory method to be static as it will not call by any instance of node.

```
public static Node nodeFactory(){  
    Node node = new Node();  
    int portUsed = createContainer();  
    node.setPortNumber(portUsed);  
    return node;  
}
```


17. Use explanatory variables

“More explanatory variables are generally better than fewer. It is remarkable how an opaque module can suddenly become transparent simply by breaking the calculations up into well-named intermediate values.”

I applied this principle for all place that need to explain it instead of writing comment.

18. Function names should say what they do

“If you have to look at the implementation (or documentation) of the function to know what it does, then you should work to find a better name”

I applied this principle very carefully as all method names describe what they are do. Like the following method, it check if the schema exist in the disk.

```
public boolean isValidSchemaName(String schemaName){  
    return dataReader.isFileExist( directoryName: null, schemaName);  
}
```

19. Replace magic number with constant

“In general it is a bad idea to have raw numbers in your code. You should hide them behind well-named constants.”

This principle does not applied many times as there is no magic number in the code but I used it when needed. Like in the folloing instead of put a hard coded number for schema count, I declare it as a constant.

```
public class LRUCache implements Cache{  
    private final int MAX_SCHEMA_NUMBER = 2;  
    private final int MAX_Object_NUMBER = 1;
```

20. Be precise

“When you make a decision in your code, make sure you make it *precisely*. Know why you have made it and how you will deal with any exceptions.”

I applied this principle in all situation I need to make a decision. Like the following, I synchronized the objects I will modify to guarantee the result and the other threads will read modified objects.

```
private void dropCache(){
    if(isDirty){
        synchronized (schemas){
            synchronized (LRUQueue){
                schemas.clear();
                LRUQueue.clear();
            }
        }
    }
}
```

21. Encapsulate the condition

“Boolean logic is hard enough to understand without having to see it in the context of an if or while statement. Extract functions that explain the intent of the conditional.”

I applied this principle for the most condition as mapped them to a function.

```
public boolean isFileExist(String directoryName,String fileName) {
    File[] allFolders = dataReader.getAllSubDirectories(directoryName);
    List<String> allDirectories=Arrays.asList(allFolders)
        .stream().map(File::getName)
        .collect(Collectors.toList());
    return allDirectories.contains(fileName);
}
```

22. Avoid negative conditions

“Negatives are just a bit harder to understand than positives. So, when possible, conditionals should be expressed as positives”

I applied this principle for the most condition as treated as positive. Like the following

```
if(objects.containsKey(id)) {  
    moveObjectToFirst(id);  
    return objects.get(id);  
}
```

23. Function should do one thing

“It is often tempting to create functions that have multiple sections that perform a series of operations. Functions of this kind do more than *one thing*, and should be converted into many smaller functions, each of which does *one thing*.”

I applied this principle for all methods as they do only one simple thing. Like the following

```
private void deleteObject(HashMap<String, String> objectInfo){  
    deleteObjectFile( objectInfo);  
}
```

24. Hidden temporal coupling

“Temporal couplings are often necessary, but you should not hide the coupling. Structure the arguments of your functions such that the order in which they should be called is obvious”

I applied this principle for all situation I need temporal dependent for another class like:

```

public void showObject() throws IOException, ClassNotFoundException {
    HashMap<String, String> objectInfo = readObjectInfoFromUser();
    if(objectInfo != null) {
        String schemaName = objectInfo.get("schemaName");
        String typeName = objectInfo.get("typeName");
        String id = objectInfo.get("id");
        Schema schema = (Schema)cache.getBlock(schemaName);
        JSONObject object = schema.getType(typeName).getObject(id);
        outputToUser.writeObject(object.toJSONString());
    }
}

```

25. Avoid transitive navigation

“In general we don’t want a single module to know much about its collaborators. More specifically, if A collaborates with B, and B collaborates with C, we don’t want modules that use A to know about C”

I tried my best to apply this principle for all classes I implemented in order to decoupling them from each other.

7.2.5 Java

1. Avoid long import list

“If you use two or more classes from a package, then import the whole package”

I applied this principle for the whole project like the following:

```

import org.json.simple.*;
import org.json.simple.parser.*;

import java.io.IOException;
import java.util.*;

```

2. Do not inherit constant

“Puts some constants in an interface and then gains access to those constants by inheriting that interface”

I applied this principle for all interfaces and there subtypes as interface contains only methods.

7.2.6 Names

1. Choose descriptive name

“Don’t be too quick to choose a name. Make sure the name is descriptive. The power of carefully chosen names is that they overload the structure of the code with description”

I applied this principle for all methods and variables so that they describe their jobs.

2. Choose name at appropriate level of abstraction

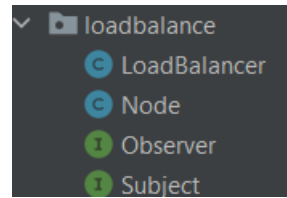
“Don’t pick names that communicate implementation; choose names that reflect the level of abstraction of the class or function you are working in.”

I applied this principle for all classes, methods and variables so that they describe their jobs, with taking care about Java naming conventions.

3. Use slandered nomenclature where possible

“Names are easier to understand if they are based on existing convention or usage. For example, if you are using the DECORATOR pattern, you should use the word Decorator in the names of the decorating classes”

I applied this principle when implementing Observer design patterns



4. Use long name for long scope

“The length of a name should be related to the length of the scope. You can use very short variable names for tiny scopes, but for big scopes you should use longer names.”

I applied this principle for all methods and variables. Like the following:

5. Avoid encodings

“Names should not be encoded with type or scope information. Prefixes such as m_ or f are useless in today’s environments.”

I did not use any encoding in the naming of variables or methods.

6. Name should describe side effect

“Names should describe everything that a function, variable, or class is or does. Don’t hide side effects with a name.”

I applied this principle when it need like the following:

```
public void createSchema()  
public void addObjectType()  
public void addObject() th
```

7.2.7 Testing

As the project consist of multiple application, I couldn’t test it using testing tool, so I test it manually.

8. Effective Java

8.1 Introduction

In this section, I will discuss the pricelessness of effective java determined by Joshua Bloch in his book and where I used them in the project.

8.2 Creating and destroying objects

1. 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 applied this principle for most of the classes, as most of them is build using singleton design pattern. Like the following:

```
public static Node nodeFactory(){  
    Node node = new Node();  
    int portUsed = createContainer();  
    node.setPortNumber(portUsed);  
    return node;  
}
```

2. Consider a builder when faced with many constructor parameter

“Static factories and constructors share a limitation: they do not scale well to large numbers of optional parameters”

I did not apply this principle as I mention in the above principle that most of classes implement using singleton design patterns.

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

“There are two common ways to implement singletons. Both are based on keeping the constructor private and exporting a public static member to provide access to the sole instance”

I implement singleton design pattern using the approach of static factory method tacking care about thread safety.

```
private static volatile FileReaderWriter dataReaderWriter = null ;  
private FileReaderWriter(String path) { this.PATH=path; }  
public static FileReaderWriter getInstance(String path){...}
```

4. Prefer dependency injection to hardwiring resource

“Many classes depend on one or more underlying resources. Do not use a singleton or static utility class to implement a class that depends on one or more underlying resources whose behavior affects that of the class, and do not have the class create these resources directly”

I applied this principle to pass reference input and output streams that refer to the object streams of the socket to the Client handler class.

```
inputFromUser = new ObjectInputStream(clientSocket.getInputStream());  
outputToUser = new ObjectOutputStream(clientSocket.getOutputStream());  
ClientHandler clientHandler = new ClientHandler(inputFromUser,outputToUser);  
clientHandler.handleClientOperation();
```

5. 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 applied this principle by using factory method and declared the reusable object to be field of the class.

6. 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 the finalizer in the project.

7. 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 did not use this principle, as all the resource I used should be declared as field of the classes.

8.3 Methods common to all object

This chapter of the book discuss the override of methods from Object class and the comparable interface which in the project I did not need any of them.

8.4 Classes and interfaces

1. Minimize the accessibility of the class

“A well-designed component hides all its implementation details, cleanly separating its API from its implementation. Make each class or member as inaccessible as possible”

I applied this principle for all classes by making their internal filed inaccessible and reduce the number of methods, which is public as possible. Like the following:

```
public Block getBlock(String blockName) {...}
private void moveBlockToFront(String schemaName){...}
private boolean isBlockExist(String schemaName){...}
```

2. In public classes, use accessor methods, not public fields

“Public classes should never expose mutable fields. If a public class exposes its data fields, all hope of changing its representation is lost because client code can be distributed far and wide.”

I applied this principle along with OOP encapsulation principle by making all classes filed accessible only through setter and geter methods. Like the following:

```
public String getIp() {...}
public void setIp(String ip) {...}
public int getPortNumber() {...}
public void setPortNumber(int portNumber) {...}
```

3. Minimize mutability

“An immutable class is simply a class whose instances cannot be modified. All of the information contained in each instance is fixed for the lifetime of the object, so no changes can ever be observed”

There is no immutable class in the project.

4. Favor composition over inheritance

“Inheritance is powerful, but it is problematic because it violates encapsulation. It is appropriate only when a genuine subtype relationship exists between the subclass and the superclass.”

In the project I did not use inheritance instead of it I used interfaces and composition so the design will be more fixable. Like the following:

```
public class Node {
    private final Cache cache = LRUCache.getInstance();
```

5. Prefer interface to abstract

“Java has two mechanisms to define a type that permits multiple implementations: interfaces and abstract classes. Because Java permits only single inheritance, this restriction on abstract classes severely constrains their use as type definitions. Any class that defines all the required methods and obeys the general contract is permitted to implement an interface, regardless of where the class resides in the class hierarchy.”

In general, interface is more preferable for extending the types as it give more flexibly for the class to implement more than one interface so in the project I used interface instead of abstract class. Like the following:

```
public interface Cache {  
    Block getBlock(String blockName);  
    void update();  
}
```

```
public interface Block {  
    Object getBlockData();  
}
```

6. Design interfaces for posterity

“The declaration for a default method includes a default implementation that is used by all classes that implement the interface but do not implement the default method. It is not always possible to write a default method that maintains all invariants of every conceivable implementation”

According to this principle, I did not implement any default method in any interface in the project; also, I kept the interface suitable for any new type may be implement them.

7. Use interfaces only to define types

“If the constants are strongly tied to an existing class or interface, you should add them to the class or interface”

Referring to the interface I used I did not declare a constant inside them, like the following interface Cache and LRUCache class that

implement it instead of declare max number of schema and object in the interface I declare them in the class.

```
public interface Cache {  
    Block getBlock(String blockName);  
    void update();  
}
```

```
public class LRUCache implements Cache{  
    private final int MAX_SCHEMA_NUMBER = 2;  
    private final int MAX_Object_NUMBER = 1;
```

8. Favor static member classes over nonstatic

“If you declare a member class that does not require access to an enclosing instance, always put the static modifier in its declaration”

I used member class in declaring a class that execute threads and implement Runnable interface in both Master and Node project, as I need each thread to be an instance of this class I did not make it static.

9. Limit source files to a single top-level class

“Never put multiple top-level classes or interfaces in a single source file”

There is no tow classes or interfacing in the same source file in the project.

8.5 Generics

1. Don't use raw types

“Using raw types can lead to exceptions at runtime, so don't use them. They are provided only for compatibility and interoperability with legacy code that predates the introduction of generics.”

I did not use any row type in the project.

2. Eliminate unchecked warnings

“Unchecked warnings are important. Do not ignore them. Every unchecked warning represents the potential for a `ClassCastException` at runtime.”

I tried to eliminate all unchecked warnings.

3. Prefer lists to arrays

“Arrays and generics have very different type rules. Arrays are covariant and reified; generics are invariant and erased. As a consequence, arrays provide runtime type safety but not compile-time type safety, and vice versa for generics.”

In the project, I used list instead of array and when I enforced to use array as following, I convert it to list.

```
public boolean isFileExist(String directoryName,String fileName) {  
    File[] allFolders = dataReader.getAllSubDirectories(directoryName);  
    List<String> allDirectories=Arrays.asList(allFolders)  
        .stream().map(File::getName)  
        .collect(Collectors.toList());  
    return allDirectories.contains(fileName);  
}
```

8.6 Enums and Annotations

This chapter is discuss the using of enum and annotation and when they should be used. An enumerated type is a type whose legal values consist of a fixed set of constants, such as the seasons of the year,the advantages of enum types over int constants are compelling. Enums are more readable, safer, and more powerful.

As I did not used any enum type or framework that need annotations, I did not apply these principles in this project.

8.7 Lambdas and Streams

1. Prefer method references to lambdas

“Method references often provide a more succinct alternative to lambdas. Where method references are shorter and clearer, use them; where they aren't, stick with lambdas.”

I applied this principle as the following:

```
public boolean isFileExist(String directoryName,String fileName) {  
    File[] allFolders = dataReader.getAllSubDirectories(directoryName);  
    List<String> allDirectories=Arrays.asList(allFolders)  
        .stream().map(File::getName)  
        .collect(Collectors.toList());  
    return allDirectories.contains(fileName);  
}
```

8.8 Methods

1. Check parameters for validity

“Each time you write a method or constructor, you should think about what restrictions exist on its parameters. You should document these restrictions and enforce them with explicit checks at the beginning of the method body. It is important to get into the habit of doing this.”

I applied this principle for the most methods that read data from the client like read schema name or type name, I checked them to make sure they are exist.

```
public boolean isValidSchemaName(String schemaName){  
    return dataReader.isFileExist( directoryName: null,schemaName);  
}
```

2. Design method signatures carefully

“Names should always obey the standard naming conventions. Every method should “pull its weight.” ”

This principle has been discussed in the in the clean code and explain where it used.

8.9 General Programming

1. Minimize the scope of local variables

“The most powerful technique for minimizing the scope of a local variable is to declare it where it is first used”

I applied this principle for all methods in the project, as I declared each local variable to the nearest place from where it is firstly used. Like the following:

```
HashMap<String, String> objectInfo = readObjectInfoFromUser();
if(objectInfo != null) {
    String schemaName = objectInfo.get("schemaName");
    String typeName = objectInfo.get("typeName");
    String id = objectInfo.get("id");
```

2. Prefer for-each loops to traditional for loops

“The for-each loop provides compelling advantages over the traditional for loop in clarity, flexibility, and bug prevention, with no performance penalty. Use for-each loops in preference to for loops wherever you can.”

In the project I used for-each instead traditional for loop whenever I need loop.

3. 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 the project I used the most popular libraries like java.io, java.lang, java.concurrent ..etc.

4. Refer to objects by their interfaces

“If appropriate interface types exist, then parameters, return values, variables, and fields should all be declared using interface types.”

I applied this principle in the project whenever it need to give the code the flexibility. Like the following:

```
List<String> allDirectories=Arrays.asList(allFolders)
```

8.10 Exceptions

1. Use exceptions only for exceptional conditions

“Exceptions are designed for exceptional conditions. Don’t use them for ordinary control flow, and don’t write APIs that force others to do so”

I applied this principle in the project whenever I need to use exception I used them.

2. Favor the use of standard exceptions

“Reusing standard exceptions has several benefits. Chief among them is that it makes your API easier to learn and use because it matches the established conventions that programmers are already familiar with”

I applied this principle in the project whenever I need to use exception I used standard exception. Like the following:


```
try {...} catch (IOException | ClassNotFoundException e) {  
    e.printStackTrace();  
}
```

8.11 Concurrency

1. Synchronize access to shared mutable data

“When multiple threads share mutable data, each thread that reads or writes the data must perform synchronization”

As mention in multiple thread section, I synchronized each data that shared between threads.

2. Prefer executors, tasks, and streams to threads

“Choosing the executor service for a particular application can be tricky.”

In this project, I used cached thread pool instead of threads.

```
ExecutorService executorService = Executors.newCachedThreadPool();
```

3. Use lazy initialization judiciously

“You should initialize most fields normally, not lazily. If you must initialize a field lazily in order to achieve your performance goals or to break a harmful initialization circularity, then use the appropriate lazy initialization technique.”

I applied this principle with technique “If you need to use lazy initialization for performance on an instance field, use the double-check idiom” when implement singleton design pattern.

```
public static FileReaderWriter getInstance(String path){  
    if(dataReaderWriter == null){  
        synchronized (FileReaderWriter.class){  
            dataReaderWriter = new FileReaderWriter(path);  
        }  
    }  
    return dataReaderWriter;  
}
```

9. SOLID design principles

9.1 Introduction

Simply put, Martin and Feathers' **design principles encourage us to create more maintainable, understandable, and flexible software**. Consequently, **as our applications grow in size, we can reduce their complexity** and save ourselves many headaches further down the road!

9.2 S : Single responsibility

“A class should only have one responsibility. Furthermore, it should only have one reason to change.”

This principle is applied for all classes and methods as they have only one job to do. Classes are built carefully tacking care about each one and its responsibility. Like **LoadBalancer** which its responsibility describe by its name which load the balance between all node and **FileReaderWriter** which responsible for all reading and writing operation from and to a file.

```
public class LoadBalancer implements Subject {
    private final List<Observer> nodes = new ArrayList<>();
    private Node currentAvailableNode;

    public LoadBalancer() { createNode(); }
    public synchronized Observer getAvailableNode(){...}
    private void createNode(){...}
    private boolean isCurrentNodeFull(){...}
    @Override
    public void register(Observer observer) { nodes.add(observer); }
    @Override
    public void unregister(Observer observer) { nodes.remove(observer); }
    @Override
    public void notifyUpdate() {...}
}
```

9.3 O : Open/close principle

“Classes should be open for extension but closed for modification”

An example for this principle is **LRUCache** class as it depend on a **Block** interface not on its subtype so if we want the cache to store a new type it just need to implement **Block** interface and **LRUCache** class code will not modify.

```
public class LRUCache implements Cache{
    private final int MAX_SCHEMA_NUMBER = 2;
    private final int MAX_Object_NUMBER = 1;
    private boolean isDirty = false;
    private final String PATH = "/database";
    private final HashMap<String, Block> schemas ;
    private final Deque<String> LRUQueue;
    private final FileReader dataReader = FileReader.getInstance(PATH);
    private final static LRUCache LRU_CACHE = new LRUCache();
}
```

```
public interface Block {
    Object getBlockData();
}
```

9.4 L : Liskov Substitution

“Super types can be substituted by subtypes without affecting correctness. In other words, a subclass should not change the expected behavior inherited from a superclass”

An example from the project is **Cache** interface and **LRUCache** class, as LRUCache class implements Cache interface and they are substituted in a place of each other and they do not affect the correctness.

```
private final Cache cache = LRUCache.getInstance();
```

9.5 I : interface segregation

“Larger interfaces should be split into smaller ones. By doing so, we can ensure that implementing classes only need to be concerned about the methods that are of interest to them”

There is no larger interface in the project, all interface contains only the methods that should be implemented by their subtypes.

9.6 D : Dependency injection

“A class should not depend on low-level concrete classes; instead it should depend on abstractions. In other words, client classes should depend on an interface or abstract class, rather than a concrete resource”

An example is the classes depends on the cache, actually they are depend on Cache interface instead of that, so add new type of caching or modifying **LRUCache** class will not affect them as they are depend on abstraction.

```
public class ClientHandler {  
    private final ObjectInputStream inputFromUser;  
    private final ObjectOutputStream outputToUser;  
    private Socket masterSocket;  
    private ObjectInputStream inputFromMaster;  
    private ObjectOutputStream outputToMaster;  
    private final Cache cache = LRUCache.getInstance();  
    private final FileReader dataReader = FileReader.getInstance("/database");  
}
```

10. Design patterns

10.1 Introduction

Design patterns represent the best practices used by experienced object-oriented software developers. Design patterns are solutions to general problems that software developers faced during software development. These solutions were obtained by trial and error by numerous software developers over quite a substantial period.

There are three types of design patterns:

- **Creational Patterns:** These design patterns provide a way to create objects while hiding the creation logic, rather than instantiating objects directly using the new operator. This gives the program more flexibility in deciding which objects need to be created for a given use case.
- **Structural Patterns:** These design patterns concern class and object composition. The concept of inheritance is used to compose interfaces and define ways to compose objects to obtain new functionalities.
- **Behavioral Patterns:** These design patterns are specifically concerned with communication between objects.

In this section, I will discuss the design pattern I used in implementation of the project.

10.2 Singleton

This pattern is one of creational patterns; it is ensure a class has only one instance that is publicly accessible. In other words, a class must ensure that only single instance should be created and all other classes can use single object.

I used this pattern in the classes that dealing with resource like **FileReaderWriter** in master application and **FileReader** class in node application, as I the all classes should use the same object to read or write from or to a file, so data will be consistent among all threads. In these two implementation, I used lazy instantiation, as I want to pass a value for the final parameter **PATH** so I need to create the object when one of threads request the object.

```
private final String PATH;
private static volatile FileReaderWriter dataReaderWriter = null ;
private FileReaderWriter(String path) { this.PATH=path; }
public static FileReaderWriter getInstance(String path){
    if(dataReaderWriter == null){
        synchronized (FileReaderWriter.class){
            dataReaderWriter = new FileReaderWriter(path);
        }
    }
    return dataReaderWriter;
}
```

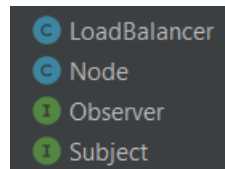
The second used is when implement **LRUCache** class in node project, as there should be only one cache in node. Unlike the previous implementation, here I used early instantiation.

```
private final static LRUCache LRU_CACHE = new LRUCache();
private LRUCache(){...}
public static LRUCache getInstance() { return LRU_CACHE; }
```

10.3 Observer design pattern

This pattern is one of behavioral patterns, Observer Pattern says that "just define a one-to-one dependency so that when one object changes state, all its dependents are notified and updated automatically".

This pattern has been used in implement load balancing algorithm, as I need when the master change the state of the database (do a write operation) all other nodes to be notified, the operation in details is discussed in the load balancing section.



In order to achieve the above goal I implement this design pattern, as **LoadBalancer** class implement **Subject** interface so it contains list of nodes, with ability to register new node to the list and unregister a node from the list.

```
public interface Subject {
    void register(Observer observer);
    void unregister(Observer observer);
    void notifyUpdate();
}
```

```
public class LoadBalancer implements Subject {
    private final List<Observer> nodes = new ArrayList<>();
    private Node currentAvailableNode;

    public LoadBalancer() { createNode(); }
    public synchronized Observer getAvailableNode(){...}
    private void createNode(){...}
    private boolean isCurrentNodeFull(){...}
    @Override
    public void register(Observer observer) { nodes.add(observer); }
    @Override
    public void unregister(Observer observer) { nodes.remove(observer); }
    @Override
    public void notifyUpdate() {...}
}
```

Node class implement **Observer** interface, so that when the master make any changes to the database will notify it through **LoadBalancer** , then it will connect to the actual node to inform it with the status of the database.

```
public interface Observer {  
    void update();  
}
```

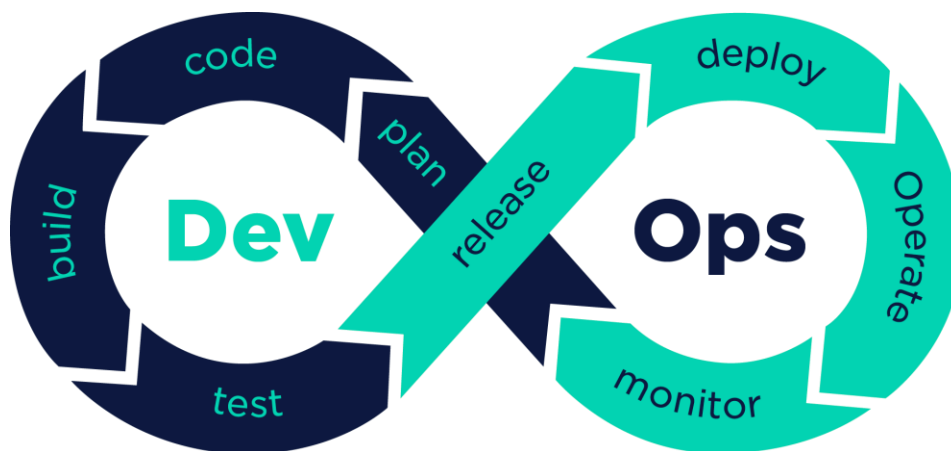
```
public class Node implements Observer{  
    private final String ip = "localhost";  
    private int portNumber;  
    private int numberOfConnections=0;  
  
    public static Node nodeFactory(){...}  
    private static int createContainer(){...}  
    private static int generateRandomPortNumber(){...}  
    public int getPortNumber() { return portNumber; }  
    public void setPortNumber(int portNumber) { this.portNumber = portNumber; }  
    public int getNumberOfConnections() { return numberOfConnections; }  
    public void incrementNumberOfConnections(){numberOfConnections++;}  
    @Override  
    public void update() {...}  
}
```


11. DevOps practices

11.1 Introduction

DevOps is the combination of cultural philosophies, practices, and tools that increases an organization's ability to deliver applications and services at high velocity: evolving and improving products at a faster pace than organizations using traditional software development and infrastructure management processes.

In this section, I will discuss the DevOps tools I used and the practices I do.



11.2 IntelliJ

Is the most popular IDE for development Java application, I used it during implementing this project.



11.3 Maven

Maven is a build automation tool used primarily for Java projects. I used it to manage the dependency and plugins used in the project and build the application as maven has its own life cycle.

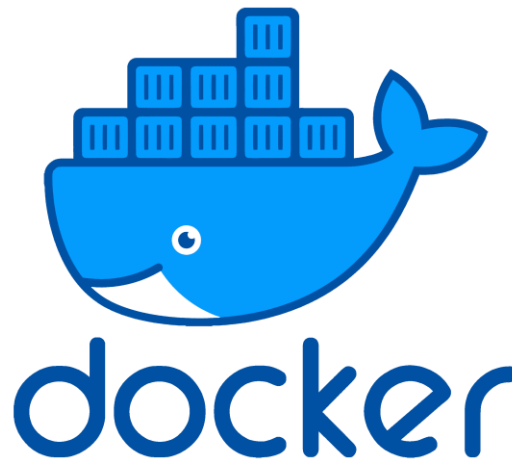


Along with Docker, I used maven to generate build a Docker image for the application (node) and its dependencies, as seen in the following images.

```
PS C:\Users\user\Desktop\Final_Project\Node> mvn clean package
[INFO] Scanning for projects...
[WARNING]
[WARNING] Some problems were encountered while building the effective model for
[WARNING] 'build.plugins.plugin.(groupId:artifactId)' must be unique but found d
[WARNING]
[WARNING] It is highly recommended to fix these problems because they threaten t
[WARNING]
[WARNING] For this reason, future Maven versions might no longer support buildin
[WARNING]
[INFO]
[INFO] -----< com.atypn:Node >-----
[INFO] --- maven-dependency-plugin:3.0.1:copy-dependencies (default) @ Node ---
[INFO] Copying junit-4.10.jar to C:\Users\user\Desktop\Final_Project\Node\target\lib\junit
[INFO] Copying json-simple-1.1.1.jar to C:\Users\user\Desktop\Final_Project\Node\target\li
[INFO] Copying hamcrest-core-1.1.jar to C:\Users\user\Desktop\Final_Project\Node\target\li
[INFO]
[INFO] --- maven-jar-plugin:3.0.2:jar (default-jar) @ Node ---
[INFO] Building jar: C:\Users\user\Desktop\Final_Project\Node\target\Node.jar
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time: 6.478 s
[INFO] Finished at: 2022-02-12T06:28:41+02:00
[INFO] -----
```

11.4 Docker

Docker is a software platform that allows you to build, test, and deploy applications quickly. Docker packages software into standardized units called containers that have everything the software needs to run including libraries, system tools, code, and runtime. Using Docker, you can quickly deploy and scale applications into any environment and know your code will run.



I used Docker in packaging the projects by building Docker image for the project and run containers so I can run multiple node in my laptop.

```
Microsoft Windows [Version 10.0.19043.1466]
(c) Microsoft Corporation. All rights reserved.

C:\Users\user>docker image ls
REPOSITORY    TAG       IMAGE ID       CREATED        SIZE
node          v1        7c5cd20312dc   27 hours ago   85.3MB

C:\Users\user>docker container ps
CONTAINER ID   IMAGE     COMMAND                  CREATED        STATUS        PORTS                               NAMES
7dc3913ed1e4   node:v1   "/usr/bin/java -jar ..." 14 seconds ago Up 9 seconds   0.0.0.0:788->5000/tcp   stupefied_satoshi
```

The Docker images is built using Docker file that I declare its structure (commands which will run) with variable that will be set using maven after building the project.

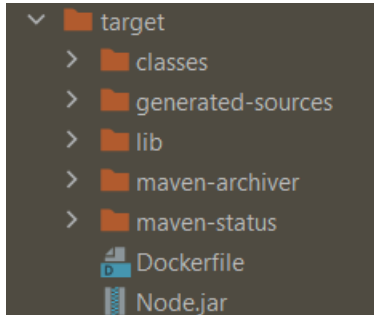
This is the Docker file before maven build the project:

```
FROM openjdk:8-jre-alpine

WORKDIR ${application.workdir}
COPY ${application.dependencies} ${application.dependencies}
COPY ${project.build.finalName}.jar ${project.build.finalName}.jar

ENTRYPOINT ["/usr/bin/java", "-jar", "${project.build.finalName}.jar"]
EXPOSE 5000
```

The generated Docker file will be in the target directory as I specified in the pom.xml file



```
FROM openjdk:8-jre-alpine

WORKDIR application
COPY lib lib
COPY Node.jar Node.jar

ENTRYPOINT ["/usr/bin/java", "-jar", "Node.jar"]
EXPOSE 5000
```








Docker also used to build Docker network and add all the container of master and worker node to it, so they cloud communicate with each other and managed easily. But in my case I couldn't create a network as the load balancer create the node container so if I containerized master controller which contain the load balancer it will not be able to create the node container.

11.5 GitHub

GitHub is a code-hosting platform for version control and collaboration. It lets you and others work together on projects from anywhere. It is one of the best Version control system with multiple features like continues integration and deployment.



I used GitHub for managing both projects and save version of codes, but the most benefit of using, it was using the GitHub actions for creating a pipeline for CI/CD. As I create repository for node project and create .yml file for declaring a workflow (pipeline) as shown in the following images:

 ahamddrabkah	Last version	✓ 196cd90 1 hour ago	🕒 21 commits
 .github/workflows	Latest update		2 hours ago
 .idea	Latest update		2 hours ago
 src/main	Last version		1 hour ago
 target	Last version		1 hour ago
 pom.xml	Initial commit		6 hours ago
 script.sh	Initial commit		6 hours ago

```








name: Node_CI
on:
  push:
    branches: [ master ]
  pull_request:
    branches: [ master ]
jobs:
  build:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v2
      - name: Set up JDK 8
        uses: actions/setup-java@v2
        with:
          java-version: '8'
          distribution: 'adopt'
          cache: maven
      - name: Maven packaging
        run: mvn -B clean package --file pom.xml
      - name: Log in to Docker Hub
        uses: docker/login-action@f054a8b539a109f9f41c372932f1ae047eff08c9
        with:
          username: ${ secrets.DOCKER_USERNAME }
          password: ${ secrets.DOCKER_PASSWORD }
      - name: Build and push Docker image
        run: |
          cd target
          docker build -t node:v1 .
          docker tag node:v1 adrabkah/node:v1
          docker push adrabkah/node:v1

```

As show in the above picture the pipeline will trigger in any push or pull request to the repository, the pipeline runs on Ubuntu as a runner (OS) and install JDK 8 in order to be able to execute the project. Then it will build and package the project using Maven then deploy the project as image into the Docker hub.

The following images show the work of pipeline:

1. Before push update from the client side

 ahamddrabkah Last version ✓ 196cd90 1 hour ago 21 commits		
 .github/workflows	Latest update	2 hours ago
 .idea	Latest update	2 hours ago
 src/main	Last version	1 hour ago
 target	Last version	1 hour ago
 pom.xml	Initial commit	6 hours ago
 script.sh	Initial commit	6 hours ago

2. After push update

ahamddrabkah Last modify			70b8cdf 11 seconds ago	🕒 22 commits
📁 .github/workflows	Last modify		11 seconds ago	
📁 .idea	Latest update		3 hours ago	
📁 src/main	Last modify		11 seconds ago	
📁 target	Last version		1 hour ago	
📄 pom.xml	Initial commit		6 hours ago	
📄 script.sh	Initial commit		6 hours ago	

3. Trigger the pipeline

Summary

Jobs

● build

build

Started 19s ago

> ✓ Set up job

> ✓ Run actions/checkout@v2

> ✓ Set up JDK 8

> ✓ Maven packaging

> ✓ Log in to Docker Hub

> ● Build and push Docker image

24 b6abafe0f63: Pull complete

25 Digest: sha256:f362b165b870ef129cbe730f29065ff37399c0aa8bcab3e44b51c302938c9193

26 Status: Downloaded newer image for openjdk:8-jre-alpine

27 --> f7a292bbb70c

28 Step 2/6 : WORKDIR application

29 --> Running in 95c640eb48c1

30 Removing intermediate container 95c640eb48c1

31 --> 57bc5ed6a036

32 Step 3/6 : COPY lib lib

33 --> fc1973e0c603

34 Step 4/6 : COPY Node.jar Node.jar

4. Push the image into Docker hub :

adrabkah

Search by repository name

adrabkah / node

Last pushed: 3 minutes ago

Now when the master want to create the container it will pull the image from the Docker hub as it will not find it locally then will run the container.