

ATYPON

DocumentDB

Atypon Final Project

By

Mohammad Daoud

Java and DevOps

Fall **2021** Section #2

Date Submitted: **February 10, 2022,**

PREFACE

DocumentDB is one of the most useful NoSQL databases; this report will talk about the journey behind the implementation of DocumentDB.

So, the discussion will be about several things, starting with document databases, the theory behind them, and how we can apply them to our project. Also, the report will discuss the operation of reading and writing, clustering, indexing, thread-locks, protocol implemented, a data structure that will be implemented, scalability, consistency, security issues, and challenges I went through.

Also, the report will discuss the clean code principles according to **Uncle Bob**, Effective Java according to **Joshua Bloch**, **SOLID** principles, Design patterns that have been implemented, and **DevOps** practices in this project.

Before we go through this journey, let's see the requirements for our project?

We need to build a documentDB which have:

1. Reads are the majority of transactions (only through the replica).
2. The writes are done only through the controller node.
3. It has to be JSON structure, not XML
4. The indexing should be in an efficient manner.
5. It is scaling horizontally.
6. Export and import.
7. The administrator should have the ability to manage the users, .i.e, changing roles.

Messenger of Allah (ﷺ) said: "Whoever takes a path upon which to obtain knowledge, Allah makes the path to Paradise easy for him."

PREFACE	1
1.0 PREREQUISITE	7
1.1 Introduction	7
1.2 What is a Document?	7
1.3 DocumentDB Terms	8
1.4 Indexing, Clustering.	10
1.5 What Should We Implement?	11
2.0 DocumentDB implementation	12
2.1 Introduction	12
2.2 Common Implementation	12
2.2.1 Indexing	12
2.2.2 Security	12
2.2.3 Database, Collection, Document Structure	13
2.2.4 Caching and I/O	13
2.3 MasterNode Implementation	13
2.3 ReplicaNode Implementation	15
2.4 NamingServer	17
2.5 API Gateway	18
2.6 Clustering	19
3.0 Multithreading the locks	22
3.1 LRU cache	22
3.2 ThreadLocks With I/O	23
4.0 Desing patterns	25
4.1 Builder Pattern	25
4.2 Singleton Pattern	26
4.3 Observer Pattern	27
4.4 API Gateway Pattern	28
5.0 Data structures	29
5.1 Primitive Data Structures	29
5.2 Non-primitive Data Structure	29
5.2.1 ArrayList	29
5.2.2 Map	29
5.2.3 BTree	29
6.0 Clean Code	32
6.1 Introduction	32
6.2 Meaningful Names	32

Intention-Revealing Names	32
Avoid Disinformation	32
Make Meaningful Distinctions	32
Use Searchable Names	33
One Word per Concept	33
Don't Pun	33
Class Names	33
Method Names	33
6.3 Functions	34
Small, Do One Thing	34
Use Descriptive Names	34
Function Arguments	35
Don't Repeat Yourself (DRY)	36
Try/Catch Blocks	36
6.4 Comments	37
6.5 Objects and Data Structures	37
6.6 Error Handling	38
Use Exceptions Rather Than Return Codes and	
Define Exception Classes in Terms of a Caller's Needs	38
Don't Return Null	38
Don't Pass Null.	38
6.7 Classes	39
Classes Should Be Small, Maintaining Cohesion Results in Many Small	
Classes	39
7.0 SOLID Principles	40
7.1 Introduction	40
7.2 Single Responsibility Principle (SRP)	40
7.3 Open/Closed Principle (OCP)	41
7.4 Liskov Substitution Principle (LSP)	41
7.5 Interface Segregation Principle (ISP)	41
7.6 Dependency Inversion Principle (DIP)	42
8.0 Effective Java	43
8.1 Creating and Destroying Objects	43
Item 1: Consider static factory methods instead of constructors	43
Item 2: Consider a builder when faced with many constructor parameters	43
Item 3: Enforce the singleton property with a private constructor or an enum	
type	44
Item 4: Enforce noninstantiability with a private constructor	44

Item 5: Prefer dependency injection to hardwiring resources	44
Item 6: Avoid creating unnecessary objects	45
Item 8: Avoid finalizers and cleaners	45
Item 9: Prefer try-with-resources to try-finally	45
8.2 Methods Common to All Objects	46
Item 12: Always override toString	46
Item 14: Consider implementing Comparable	46
8.3 Classes and Interfaces	47
Item 15: Minimize the accessibility of classes and members	47
Item 16: In public classes, use accessor methods, not public fields.	47
Item 17: Minimize mutability	48
Item 18: Favor composition over inheritance	48
Item 20: Prefer interfaces to abstract classes	48
Item 24: Favor static member classes over nonstatic	49
Item 25: Limit source files to a single Top-level class	49
8.4 Generics	50
Item 26: Don't use raw types	50
Item 28: Prefer lists to arrays	50
Item 29: Favor generic types	50
Item 30: Favor generic methods	51
8.5 Enums and Annotations	52
Item 34: Use enums instead of int constants	52
Item 40: Consistently use the Override annotation	52
8.6 Lambdas and Streams	53
Item 42: Prefer lambdas to anonymous classes	53
Item 43: Prefer method references to lambdas	53
8.7 Methods	54
Item 49: Check parameters for validity	54
Item 51: Design method signatures carefully	54
Item 52: Use overloading judiciously	54
8.7 General Programming	55
Item 57: Minimize the scope of local variables	55
Item 58: Prefer for-each loops to traditional for loops	55
Item 59: Know and use the libraries	55
Item 60: Avoid float and double if exact answers are required	56
Item 64: Refer to objects by their interfaces	56
8.8 Exceptions	57
Item 69: Use exceptions only for exceptional conditions	57

Item 70: Use checked exceptions for recoverable conditions and runtime exceptions for programming errors	57
Item 71: Avoid unnecessary use of checked exceptions	58
Item 72: Favor the use of standard exceptions	58
Item 77: Don't ignore exceptions.	58
8.9 Concurrency	59
Item 78: Synchronize access to shared mutable data	59
Item 79: Avoid excessive synchronization	59
Item 84: Don't depend on the thread scheduler	59
9.0 DocumentDB Issues	60
9.1 Introduction	60
9.2 Security Issues	60
9.3 Scalability/Consistency Issues	63
9.3.1 Scalability Issues	63
9.3.2 Consistency Issues	63
REFERENCES	64

1.0 PREREQUISITE

1.1 Introduction

This chapter will discuss the theory behind DocumentDB its architects, and in the last section, we will see how we can apply all of that to our project.

1.2 What is a Document?

Dan Sullivan defines the document as “*a document is a set of key-value pairs. Keys are represented as strings of characters. Values may be basic data types (such as numbers, strings, and Booleans) or structures (such as arrays and objects). Documents contain both structure information and data. The name in a name-value pair indicates an attribute, and the value in a name-value pair is the data assigned to that attribute. JSON and XML are two formats commonly used to define documents.*”

Example for JSON document;

```
{  
    "id" : 11,  
    "name": "Mohammad"  
}
```

But what is the benefit of using documents such as JSON or XML and not using the RDBMS? You will know the answer at the end of this chapter.

1.3 DocumentDB Terms

Each DocumentDB should contain a three-part Database, Collection, and Document.

Let us define these bottom-up first the document; as mentioned in the previous section, a document is a multi key-value store, and documents may implement it in JSON or XML. Because the document is a key-value store, it has a fast search. The main benefit of a document is that the document is schemaless, which means more flexibility and more responsibility.

Now let us define the collection; a collection is a group of documents.

Documents within a collection should usually be related to the duplicate subject entry, such as employees, products, ... etc.

Figure 1 below represents the collection and how it relates to the document:

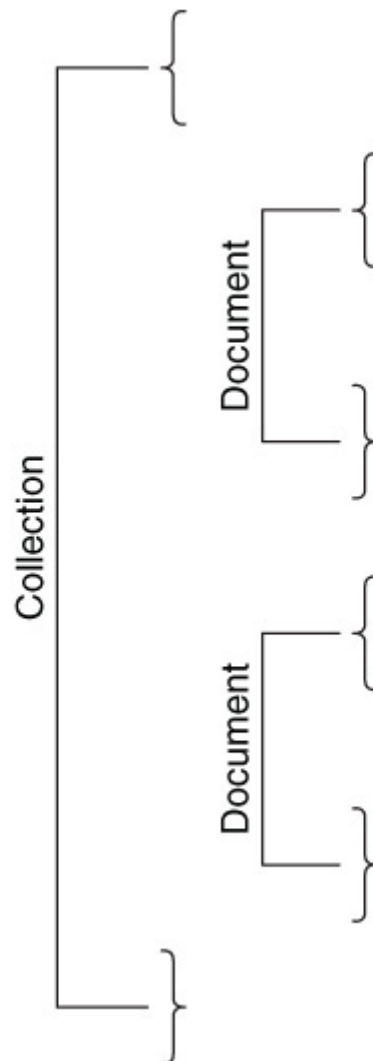


Figure 1: Collection and documents

And for the last thing database; Database is the container for multiple collections. So basically, you can imagine a database, collections, and documents as categories and subcategories of three layers DB, Col, Doc.

Figure 2 below may help you imagine how DocumentDB may be represented as folders examples.

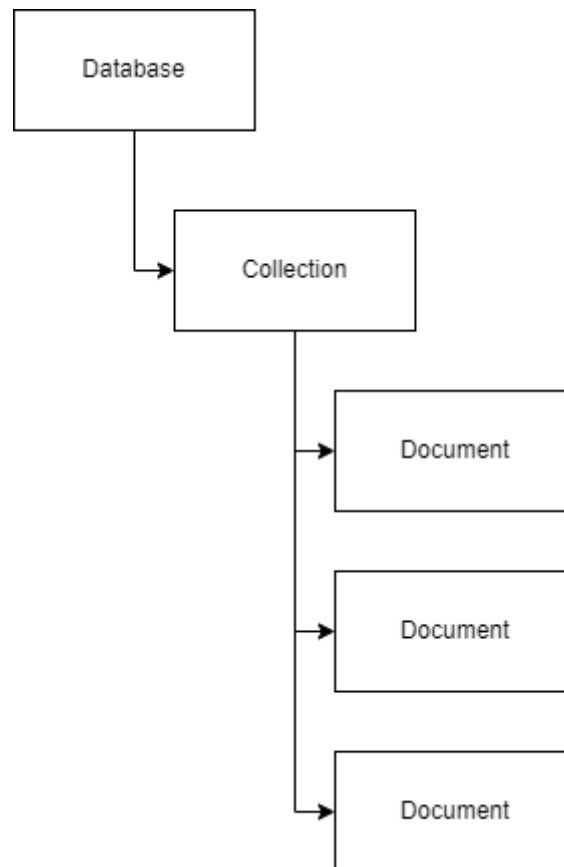


Figure 2: DocumentDB relations

To summarize the above section, if you want to develop a DocumentDB, your system should have a database containing a collection. The collection should have a group of related documents.

1.4 Indexing, Clustering.

When we say index, the first thing that comes to our head is book index, the same thing applies to DocumentDB, and the benefit for indexing is only one thing: a *fast search*.

Multiple data structures implement index algorithms such as reverse index lookup, tree index, ... etc.

Clustering, in theory, is dividing the population or data points into several groups. Data points in the same groups are more similar to other data points in the same group than those in other groups. In simple words, the aim is to segregate groups with similar traits and assign them into clusters.

The process of clustering is vast, so for DocumentDB

Each server (node) must have both read/write operations, and the primary node or leader should be voted automatically on the first takeoff, and if the leader down or some error went to him, the voting should be applied to vote the new master.

There are a lot of books talking about clustering terminology but this report forced on the project needs and the theory related to it.

One more thing to mention is scaling; there are two types of scaling; the first one is *vertical scaling* and *horizontal scaling*;

horizontal scaling refers to adding additional nodes, while *vertical scaling* describes adding more power to your current machines.

1.5 What Should We Implement?

After a lot of thinking, searching, reading, asking experts, coding, getting errors, and repeating the process again and again (the following chapters will talk about many of these challenges for this project), the final Idea to implement our DocumentDB project will be as follow:

1. Q: How will clients talk to the server for read/write operations?

A: REST API using spring-boot *MasterNode* for writing and *ReplicaNode* for reading.

2. Q: Indexing?

A: HashMap and BTree.

3. Q: How will the clustering be?

A: The correct way to do that is to make each node has read/write operations, but this was so complicated (clustering will discuss this challenge later on); So the clustering will be as one primary node that only writes operation will be on it, and the read operation will be through replica, so our application will provide microservices structure.

4. Q: Security?

A: BasicAuth.

5. Q: Caching?

A: LRU cache for each replica node.

“ These are the key points for our DocumentDB”

“OUR JOURNEY BEGINS NOW”

2.0 DocumentDB implementation

2.1 Introduction

The DocumentDB will be built as a microservices program that includes four parts (service):

MasterNode, *ReplicaNode*, *NamingServer*, and *APIGateway*, then the *Clustering* part

This chapter will discuss how I implemented these services in detail, what challenges I faced, practices through the development cycle.

But before starting, let me talk a little about *DevOps* practices used in this project.

- **GitHub:** version control, the second thing as a backup, is a great place to store your project and back to an older version if any error comes after the wrong decision.
- **Apache Maven:** as a build automation tool, the primary purpose of using maven is to allow a developer to comprehend the complete state of a development effort in the shortest period, regardless of the IDE used for the development.
- **AWS:** trying to use AWS for clustering using KAFKA & Zookeeper, but unfortunately, it was too complicated to use (I'll explain it more in the clustering section).

2.2 Common Implementation

This section comes because of common structures or features with read/write operations.

2.2.1 Indexing

The indexing mechanism used in this project is *BTree* data structure and *HashMap*. You can read more about it in the data structure section.

2.2.2 Security

The security authentication is *BasicAuth* at both replica and master nodes, not in API-Gateway, and the reason is that spring cloud does not provide basic auth.

See this link for more details; visit.

Security details will be discussed in the security section.

2.2.3 Database, Collection, Document Structure

These are the foremost common things in the replica and master nodes, so it is good to mention that before clarifying pieces.

2.2.4 Caching and I/O

Also, caching and I/O will use in replica/master nodes.

2.3 MasterNode Implementation

Before going in, let's aim at the objective tasks for master

1. Only the administrative job will be through the master/controller node, .i.e, write operation.
2. The master node must notify the other replica of any update.
3. Only the master node shall write in directory and files.

The structure will be as follow.

Each database will be in hashmap; the key will be the database name, and the value will be the collection group.

Each collection will also be in hashmap; the key will be the collection name, and the value will be the document group.

Now for documents will be implemented as BTree; the key will be the index property, and the value will be the JSON object schema.

Note: BTree explained in the Data structure part.

To ensure that every action on the controller node is from the role of admin, I've implemented the basic auth with spring boot REST API.

Each user has a username password hashed by **SHA512** and the user's role.

And that was a challenge for me; spring boot love to make things unclear, and I don't know why!

Spring boot doesn't clarify an error. I mean, by this, you have a null pointer exception. Still, where, you are the one who should know! And that wasn't very pleasant. It takes me four days to make it go right, but the good thing about programming and developing is waiting for **AHA MOMENT**.

An aphorism says the work you think will take 5 min will take three days.

For I/O actions, it will be through something more simple than a storage engine in Database theory; all I do is implement a writer and remove classes for adding/deleting files or folders.

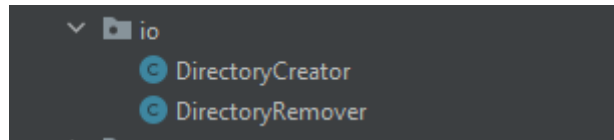


Figure 3: I/O

Now, the calling for some services is through the API: each database, collection, document, and user service and a REST controller, and for security, I've added custom PreAuthorize annotation to be sure that only the admin role can do this.

```
@PostMapping("/master/add-database")
@PreAuthorize("hasAnyAuthority('ROLE_ADMIN')")
public synchronized void addDatabase(@RequestBody Database database){...}

@DeleteMapping(path = "/master/delete/{databaseName}")
@PreAuthorize("hasAuthority('ROLE_ADMIN')")
public synchronized void deleteDatabase(@PathVariable String databaseName){...}
```

Figure 4: REST Controller example (database controller class)

```
public List<User> getAllUser() { return loadUsers(USER_GROUP); }

private List<User> loadUsers(List<User> userGroup) {...}

public synchronized void addUser(String username,
                                String password, Role role) {...}

public synchronized void deleteUser(String username) {...}

public User getUser(String username) {...}

public boolean isUserExist(String username) {...}

public synchronized void changeUserPassword(String username, String newPassword) {...}

public synchronized void changeUserRole(String username, Role role) {...}

public synchronized void changeUserUsername(String oldUsername, String newUsername) {...}
```

Figure 5: Service Example (User service class)

2.3 ReplicaNode Implementation

Let us do the same as MasterNode section, aim the objective:

1. Only the read transaction will be through replicas.
2. Reading should be fast (LRU CACHE).

The implementation of ReaplicaNode is familiar as MasterNode; same data structure, indexing. The difference is that a replica is read first from a file then uploaded to cache, and because it is a replica, it is a duplicate of a master; thus, let's talk more about caching in this section.

The caching mechanism I've implemented in this project is the ***Least Recently Used Cache (LRU CACHE)***.

After reading more about LRU cache, a cache eviction algorithm that organizes elements in order of use, in LRU, as the name suggests, the piece that hasn't been used for the longest time will be evicted from the cache.

For example, if we have a cache with a capacity of three items:

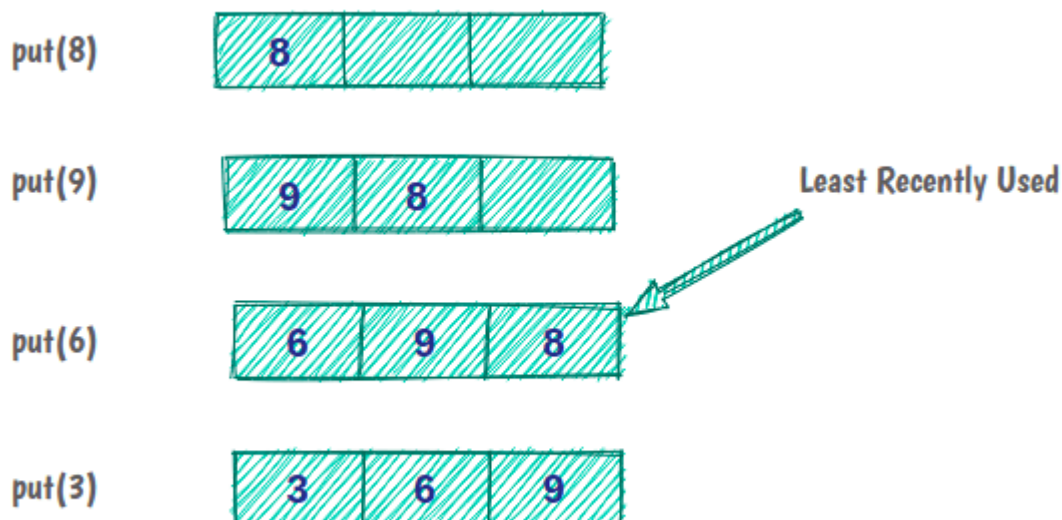


Figure 6: LRU CACHE.

Initially, the cache is empty, and we put element 8 in the cache. Elements 9 and 6 are cached as before. But now, the cache capacity is full, and to put the next element, we have to evict the least recently used element in the cache.

Before the implementation of LRU cache, it's good to know some aspects of the cache:

- All operations should run in order of $O(1)$
- The cache has a limited size
- All cache operations must support concurrency
- If the cache is full, adding a new item must invoke the LRU strategy

For a multithreaded environment, I use *ReentrantReadWriteLock* and *ConcurrentHashMap* for a thread-safe environment.

Also, the structure used in the LRU cache is hashmap and doubly linked list.

For I/O operations, I used only a single class for it; “*DirectoryLoader*” this class is responsible for loading files from disk if it is not in cache. See **Figure 7** for an example of loader class methods.

```
public static List<String> loadDirs(File parent, int level) {
    List<File> dirs = new ArrayList<>();
    List<String> finalResult = new ArrayList<>();
    File[] files = parent.listFiles();
    if (files == null) return finalResult; // empty dir

    for (File f : files) {
        if (f.isDirectory()) {
            if (level == 0) dirs.add(f);
            else if (level > 0) finalResult.addAll(loadDirs(f, level - 1));
        }
    }

    dirs.forEach(d -> finalResult.add(d.getName()));
    return finalResult;
}
```

Figure 7: loadDirs method from DirectoryLoader class

2.4 NamingServer

The architecture of a microservices application to manage the service registry must have a naming server to handle these things.

For that spring cloud, provide Netflix's Eureka server or Zookeeper.

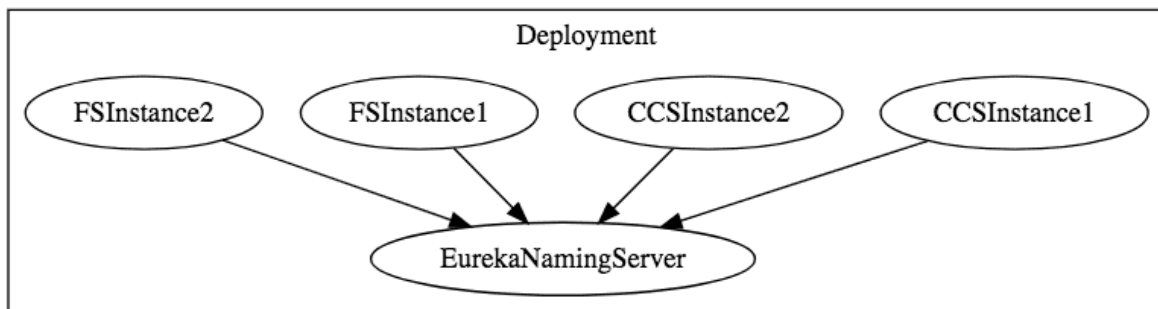


Figure 8: Eureka

The following example is the GUI interface of the eureka naming server, and the next one is how the node registers from the naming server.

Application	AMIs	Availability Zones	Status
API-GATEWAY	n/a (1)	(1)	UP (1) - localhost:api-gateway:8080
MASTER	n/a (1)	(1)	UP (1) - localhost:master:8000
READ	n/a (2)	(2)	UP (2) - localhost:read:9001 , localhost:read:9000

Figure 9: Eureka GUI

```

: Getting all instance registry info from the eureka server
: HTTP GET http://localhost:8761/eureka/apps/
: Accept=[application/json, application/*+json]
  
```

Figure 10: how node registry info from naming server

2.5 API Gateway

API management tool sits between a client and a collection of backend services.

The implementation of the API gateway was quite simple.

The only thing I want for an API gateway is routing between services.

To do that, I have to add configuration to this gateway see **Figure 11**.

```
@Configuration
public class ApiGatewayConfiguration {

    @Bean
    public RouteLocator gatewayRoute (RouteLocatorBuilder builder){
        return builder.routes()
            .route(routeFun -> routeFun.path( ...patterns: "/master/**")
                .uri("lb://master"))
            .route(routeFun -> routeFun.path( ...patterns: "/read/**")
                .uri("lb://read"))
            .build();
    }
}
```

Figure 11: API Gateway configuration

And to see what request you can do, I implement **Swagger UI** to do that. See **Figure-12**.

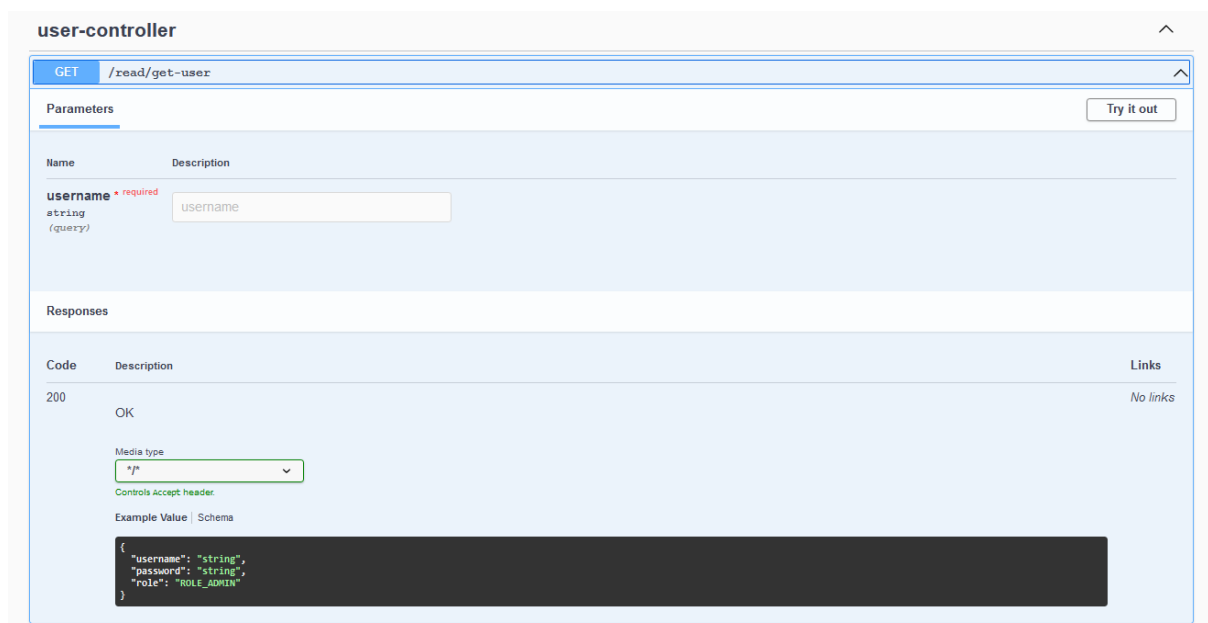


Figure 12: Swagger UI

2.6 Clustering

This is one of the most challenging parts I have faced in this project.

In the beginning, the thing that I need to be done is each node must have both read and write transactions, and the master shall be elected by voting from the other nodes.

The challenge I've faced in this part is the conspiracy of clustering. I saw that clustering is another new world of the IT universe, so first, I tried to use Kafka and Zookeeper with docker. After spending one week setting it up the way that I think it's correct using AWS as a cloud provider, I failed to connect it to DocumentDB microservices program; the reason behind that is the complexity of Kafka configuration, which require deep knowledge in I/O, network, RAM, CPU architect, OS and more.

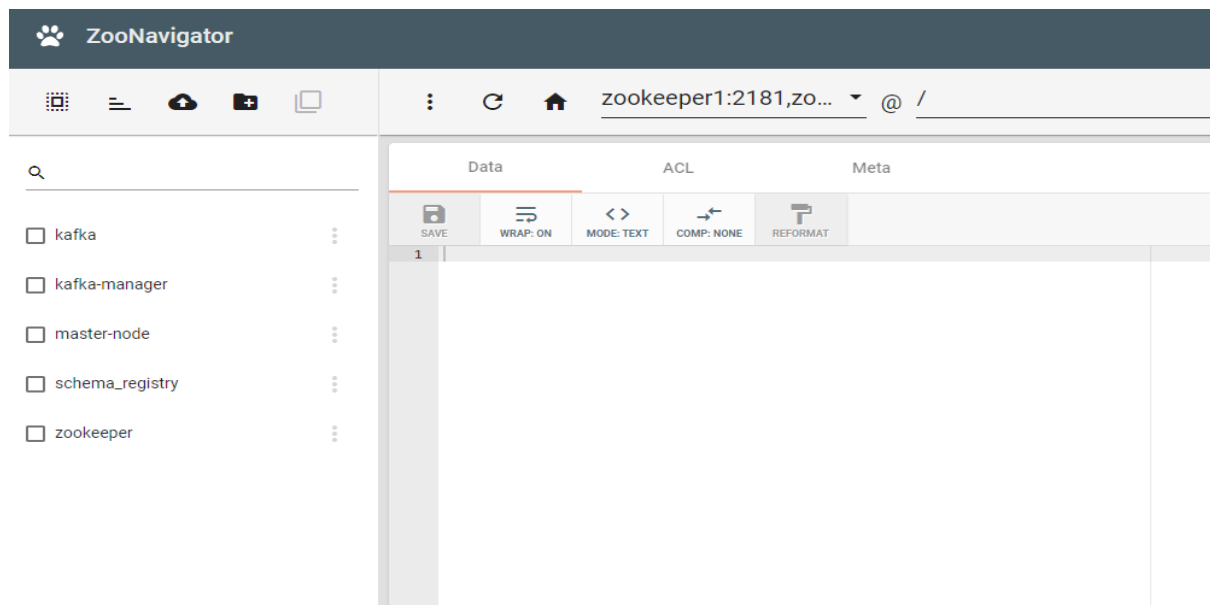


Figure 13: Zookeeper UI

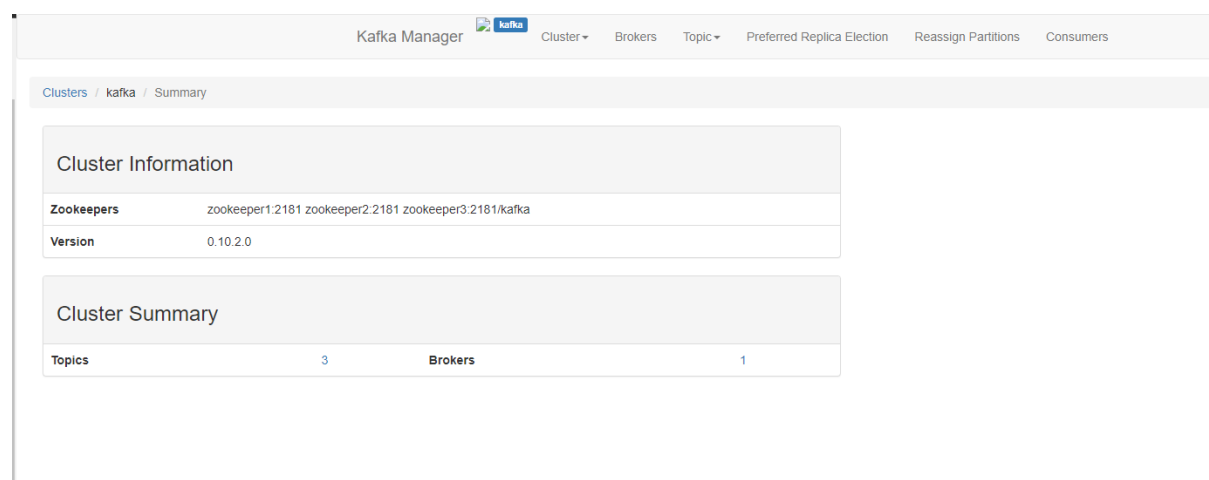


Figure 14: Kafka Manager UI

Also, here are the **AWS EC2** instances.

<input type="checkbox"/>	Name ▾	Instance ID	Instance state ▾	Instance type ▾	Status check	Alarm status
<input type="checkbox"/>	Server 3	i-0aac66f16f0d764f9	✔ Running 🔍	t2.medium	✔ 2/2 checks passed	No alarms
<input type="checkbox"/>	Server 1	i-08f5304b26882259c	✔ Running 🔍	t2.medium	✔ 2/2 checks passed	No alarms
<input type="checkbox"/>	Web Tools	i-0f3eea84b4700c386	✔ Running 🔍	t2.small	✔ 2/2 checks passed	No alarms
<input type="checkbox"/>	Server 2	i-04b846912ef184176	✔ Running 🔍	t2.medium	✔ 2/2 checks passed	No alarms

Figure 15: AWS instances view

After all of this disappointment for me and to know how much I need to work on myself, I try to use docker for that, and if also I cannot figure it out, then I'll go to the last option which is just open port to scaling and do clustering.

Docker was easier to use than Kafka, but my project's failure was that the communication with each other was always failed or corrupted, and I didn't have much time to deal with that, so I went for the last chance, different port in the same machine.

See **Figure 16** for the docker-compose YAMAL sample.

```
version: '3.7'

services:
  master-node:
    image: 0xmohammad/documentdb-naming-server:0.0.1-SNAPSHOT
    mem_limit: 700m
    ports:
      - "8000:8000"
    networks:
      - document-db-network
    depends_on:
      - naming-server

  replica-node:
    image: 0xmohammad/documentdb-replica-node:0.0.1-SNAPSHOT
    mem_limit: 700m
    ports:
      - "9000:9000"
    networks:
      - document-db-network
    depends_on:
      - naming-server
```

Figure 16: docker-compose YAMAL

For different ports running on the same machine, I make java run time execute a shell script to do this job but before that, I'll write the port number to the execution file then run it. And to scale the cluster horizontally, Suppose an update has been done to the database. The ancient replicas will be killed and generate another replica to serve them. In that case, the users will continue reading from the old replica until the data completely changes.

The creation of the other replica will be through the controller node by executing a shell script.

```
#!/bin/sh
cd 'C:/Users/mdss4/Documents/Atypen/DOCUMENT-DB PROJECT/ReplicaNode/target'
java -jar replica-node-0.0.1-SNAPSHOT.jar --server.port=9005
```

Figure 17: shell script to create a replica

And to kill the using replica the java runtime will execute a command for it
The command by using **npm** command to kill the port

```
PS C:\Users\mdss4> npx kill-port 9000
Process on port 9000 killed
```

Figure 18: command to kill port (kill replica)

3.0 Multithreading the locks

Locks are a synchronization technique used to limit access to a resource in an environment where there are many threads of execution.

This concept has been implemented in the LRU cache, write operation, and scaling.

3.1 LRU cache

To make the read as fast as possible, I've implemented LRU to minimize the disk reads but the challenge here to make the LRU cache thread-safe

Thus, to make the LRU thread-safe, I've used `ConcurrentHashMap` and `ReentrantReadWriteLock`. Also, the LRU cache has been implemented in a generic way to store any type of data. See **Figure 19**:

```
public class LRUCache<K, V> {
    private final int MAX_CAPACITY;
    private AtomicInteger curSize = new AtomicInteger();
    private ConcurrentHashMap<K, Node> map;
    private Node head;
    private Node tail;
    private ReentrantReadWriteLock readWriteLock;

    public static class Node <K,V>{...}

    public LRUCache(int maxCapacity) {...}

    public V get(K key) {...}

    public void put(K key, V value) {...}

    private void moveToHead(Node node) {...}

    private void removeNode(Node node) {...}

    private void addToHead(Node node) {...}

    public void evict(K key) {...}

    public void clearAll(){...}
}
```

Figure 19: LRU Cache implementation

So whatever the data has been retrieved, it will be on the top of the doubly linked list, making the read operation faster.

But the question is why I used thread locks in LRU?

The answer is simple: What if two write simultaneously to cache? What will happen?

There is a high chance for threads to step on each other, so each thread must acquire a lock before writing. Only one thread can hold the lock at a time, so only one thread can write to the file at a time.

From the previous paragraph, the thread locks must be implemented if we want to write, delete or update to file.

3.2 ThreadLocks With I/O

If DocuemntDB needs to make an action to database, collection, or document, the movement must be thread-safe.

For example, if two admins need to take action to a specific database

The thread must be locked for the faster one. (the processing of action will discuss in the following chapter **REST API**).

Figure 20 represents an example of thread locks for file management:

```
public class DirectoryRemover {
    private static DirectoryRemover remover;
    private static final AppLogger LOGGER = new AppLogger( logName: "DirectoryRemover LOGGER");
    private DirectoryRemover(){}

    public static DirectoryRemover getInstance() {
        synchronized (DirectoryRemover.class) {
            if (remover == null)
                remover = new DirectoryRemover();
        }
        return remover;
    }

    // the removing will be done recursively
    public synchronized void deleteDir(String filename) {
        try {
            Files.walk(Path.of( first: getMasterDir() + "/" + filename))
                .sorted(Comparator.reverseOrder())
                .map(Path::toFile)
                .forEach(File::delete);
        } catch (IOException e){
            LOGGER.logError(e);
        }
    }
}
```

Figure 20: ThreadLocks with DirectoryRemover class

As you can see in Figure 14, I have as much as I can to keep checking for threading locks to make the race condition the best way to do it.

Also, when any API requests for action, I have to ensure that there is no corruption.

```
@PostMapping("/master/add-user")
@PreAuthorize("hasAuthority('ROLE_ADMIN')")
public synchronized void addUser(@RequestBody String username,
                                  @RequestBody String password,
                                  @RequestBody Role role) {
    service.addUser(username, password, role);
}
```

Figure 21: ThreadLocks example in REST Request

4.0 Desing patterns

The most important thing to developing a system is design; this section will discuss the design patterns that have been implemented in this project.

Effective software design requires considering issues that may not become visible until later implementation. Design patterns can speed up the development process by providing tested, proven development paradigms. Reusing design patterns helps prevent subtle issues that can cause significant problems and improves code readability for coders and architects familiar with the practices.

4.1 Builder Pattern

The builder pattern is designed to provide a flexible solution to various object creation problems in object-oriented programming. The Builder design pattern intends to separate the construction of a complex object from its representation.

“Wikipedia definition”

Thus, to solve the problems of Encapsulate creating, and assembling the parts of a complex object in a separate Builder object, a class delegates object creation to a Builder object instead of making the objects directly. The main structure of the builder pattern is as shown in **Figure 22** below:

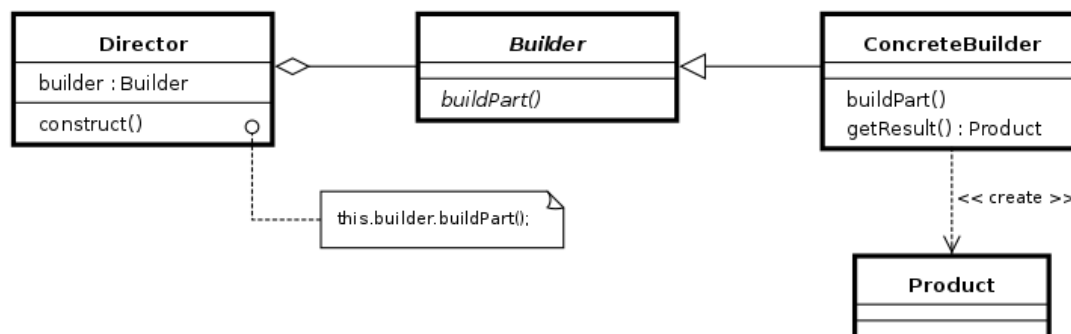


Figure 22: Builder pattern structure

This pattern is implemented in the user class to avoid the earlier problem.

Figure 23 represents an example of a builder pattern that has been implemented.

```
private User(UserBuilder builder) {  
    this.username = builder.username;  
    this.password = builder.password;  
    this.role = builder.role;  
    this.userID = builder.USER_ID;  
}
```

Figure 23: builder pattern (User Class)

4.2 Singleton Pattern

The main benefit of the singleton pattern is to ensure a class has only one instance that is publically accessible; this pattern will be applied in directory management classes.

Figure 24 below represents the structure of the singleton pattern:

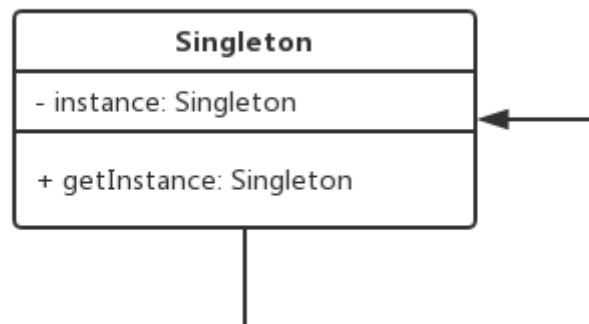


Figure 24: singleton structure

But for thread lock (well be discussed later on), we must double-check to keep the operation atomic; see **Figure 25**.

```
public static DirectoryCreator getInstance() {
    synchronized (DirectoryCreator.class) {
        if (creator == null)
            creator = new DirectoryCreator();
    }
    return creator;
}
```

Figure 25: Singleton pattern (DirectoryCreator class)

4.3 Observer Pattern

Observer pattern has an excellent methodology. Instead of checking for any update, the publisher or the master will notify all subscribers of any update.

For example, on YouTube, when you subscribe to some channel, there is no need to keep checking if there are any new videos.

The producer will notify you when the video is uploaded! See **Figure 26**:

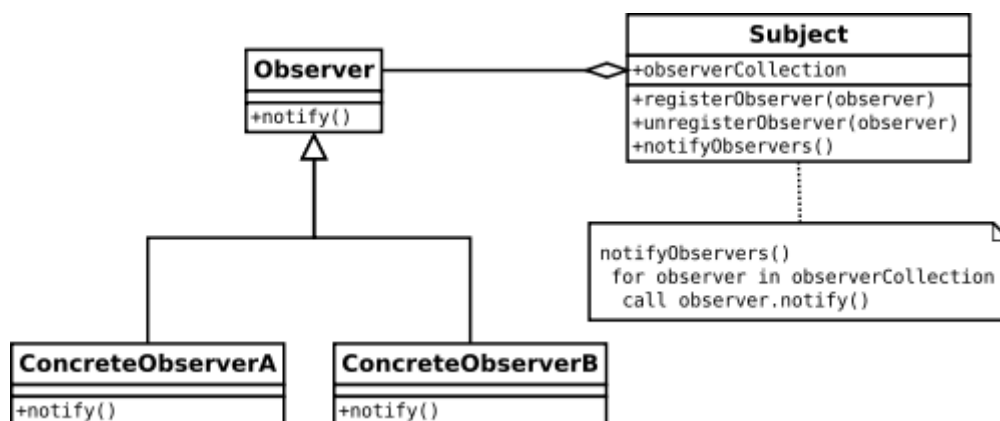


Figure 26: Observer pattern structure

The uses of observer patterns will be on master and replicas to solve replica checking for any updates on DB. The master will notify all other replicas.

Figure 27 represents the uses of the observer pattern.

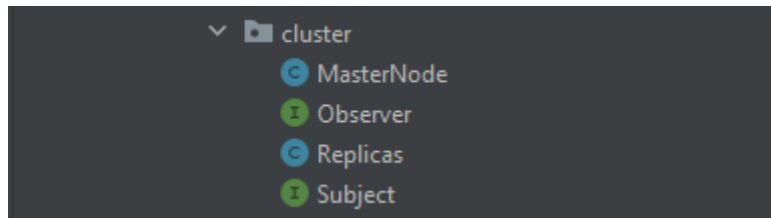


Figure 27: observer pattern (master/replica)

4.4 API Gateway Pattern

Because we build a microservice system for DocumentDB, the project must implement an API gateway pattern.

How do the clients of Microservices-based applications access the individual services? API gateway pattern solves it by handling requests in one of two ways. Some requests are proxied/routed to the appropriate service, while others are fanned out to multiple services.

The use of this pattern will be on the connection between the client and the read/write services. See **Figure 28**:

```

@Bean
public RouteLocator gatewayRoute (RouteLocatorBuilder builder){
    return builder.routes()
        .route(routeFun -> routeFun.path( ...patterns: "/master/**")
            .uri("lb://master"))
        .route(routeFun -> routeFun.path( ...patterns: "/read/**")
            .uri("lb://read"))
        .build();
}
  
```

Figure 28: API Gateway (ApiGatewayConfiguration class)

5.0 Data structures

A data structure is a data organization, management, and storage format that enables efficient access and modification. “*Wikipedia definition*”

This chapter will discuss the data structure used in the DocumentDB project.

5.1 Primitive Data Structures

Like any program, it should contain a data structure that stores only one type, like an *integer* data type.

5.2 Non-primitive Data Structure

This part is one of the challenges I’ve faced, the deficiency of non-primitive data structure is how the data will be stored efficiently; for the DocumentDB project, I have implemented several types of data structure:

5.2.1 ArrayList

ArrayLists have been implemented for nodes, databases, users, and authentication.

5.2.2 Map

Map is used for many things like using the hash map to store the collection in the database because, in java, HashMap uses the RedBlack tree if two entities have the same key.

Also, a map has been implemented for inserting JSON objects on our DocumentDB. And more like using map in caching (**LRU Cache**).

5.2.3 BTree

BTree combines a red-black tree and a binary search tree associated with a key stored in the same node as the key.

BTrees are guaranteed that each operation has $O(\log n)$ see **Figure 29**

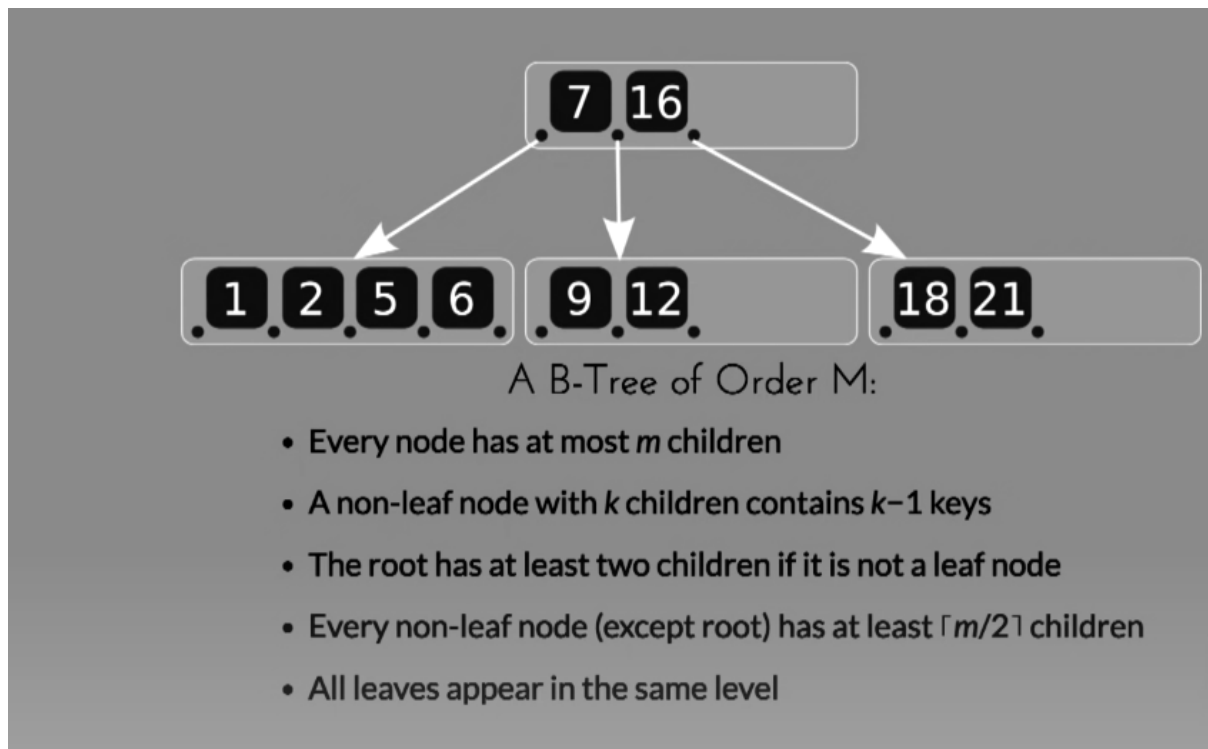


Figure 29: Terms of BTree

In the DocumentDB project, I've implemented BTree as a generic type to fit all data types of key or value stored, and the number of M children is 5.

See Figure 30.

```
public class BTree<Key extends Comparable<Key>, Value> {
    private static final int MAX_CHILDREN_NUM = 5;
    private Node root;
    private int height;
    private int groupNum;

    public BTree() { root = new Node( childrenNum: 0); }
```

Figure 30: BTree implementation

But how BTree work?

To answer this question, it may take many pages to explain it, so I'll try my best to explain it briefly in 6 roles.

- **Rule 1:** The root can have as few as one element (or even no elements if it has no children); every node has at least **MINIMUM** elements.

- **Rule 2:** The maximum number of elements in a node is twice the value of MINIMUM.
- **Rule 3:** The elements of each B-tree node are stored in a partially filled array, sorted from the smallest element (at index 0) to the largest element (at the final used position of the array).
- **Rule 4:** The number of subtrees below a non-leaf node is always one more than the number of elements in the node.

Subtree 0, subtree 1, ... etc.

- **Rule 5:** For any non-leaf node:
 1. An element at index i is greater than all the elements in subtree number i of the node, and
 2. At index, i is less than all the elements in subtree number $i + 1$ of the node.
- **Rule 6:** Every leaf in a B-tree has the same depth. Thus it ensures that a B-tree avoids the problem of an unbalanced tree.

See this [link](#) to make you very clear about BTree's data structure.

I chose BTree for the reason of time complexity. If it merged with a hashmap, the indexing would be fantastic.

Each collection has documents containing JSON objects; the user can identify an index, for example, the phone number, and if he doesn't create an index, then the index will be generated by the system as “*_objecID*”.

Figure 31 represents the indexing methods.

```
public void add(String jsonObject, String indexProperty){...}

@Override
public void add(String jsonObject){...}
```

Figure 31: JSON indexing

6.0 Clean Code

6.1 Introduction

“Even bad code can function. But if code isn't clean, it can bring a development organization to its knees. Every year, countless hours and significant resources are lost because of poorly written code” - Robert Cecil Martin (Uncle Bob).

In this chapter and the next two, I'll defend my code according to *Uncle Bob Clean Code*.

6.2 Meaningful Names

Intention-Revealing Names

“The name of a variable, function, or class should answer all the big questions. It should tell you why it exists, what it does, and how it is used. If a name requires a comment, then the name does not reveal its intent.”

Each name implemented in the DocumentDB project answer the above question.

Avoid Disinformation

“Programmers must avoid leaving false clues that obscure the meaning of code. We should avoid words whose entrenched meanings vary from our intended meaning.”

The names used in the project entirely avoid the false clues that obscure the meaning of the code.

Make Meaningful Distinctions

“Programmers create problems for themselves when they write code solely to satisfy a compiler or interpreter.”

The clean code does not make the compiler satisfied with the code. It's to fulfill the human while reading code and take the information when they look at it.

Thus, the DocumentDB project has avoided all noise words; there are no same names with different things on one scope.

Use Searchable Names

“Single-letter names and numeric constants have a particular problem in that they are not easy to locate across a body of text.”

All names that were used were searchable.

One Word per Concept

“Pick one word for one abstract concept and stick with it.”

For adding, I use add, and for removing, I use delete.

Don’t Pun

“Avoid using the same word for two purposes. Using the same term for two different ideas is essentially a pun.”

This concept and the **one word per concept** were confusing, but I know the meaning. **Don’t Pun** on this project, so I try my best not to be a pun.

```
@Override
public void add(String jsonObject) {
    if (isValidJson(jsonObject))
        documentSchema.put(SchemaBuilder.idCounter(), SchemaBuilder.build(jsonObject));
    throw new IllegalArgumentException("the json entered invalid !!");
}
```

Figure 32: Don’t Pun example

Class Names

“Classes and objects should have noun or noun phrase names.”

Each class has noun or phrase noun names.

Method Names

“Methods should have verb or verb phrase names”

Also, methods name implemented to has verb or verb phrase name, i.e., add, delete, get.

6.3 Functions

Small, Do One Thing

“The first rule of functions is that they should be small. The second rule of functions is that they should be smaller than that.”

All function is more miniature than they can be, and this rule is very effective for maintainability because it doesn't take time to know its purpose. **Figure 33** represents an example of a **small** rule.

```
public synchronized void addUser(String username,
                                String password, Role role) {
    if (isUserExist(username))
        throw new UserConflictException("the user exist");

    USER_GROUP.add(new User.UserBuilder()
        .username(username)
        .password(password)
        .role(role)
        .build());

    DirectoryCreator.getInstance().overrideWriteFile(USER_FILE.getPath(), USER_GROUP.toString());
}
```

Figure 33: small rule example

Most of the methods do one thing, and if not (there is an if-else statement), I apply the **One Level of Abstraction per Function** role.

Thus, all methods can read code top-down.

Use Descriptive Names

“Choosing descriptive names will clarify the design of the module in your mind and help you to improve it. It is not at all uncommon that hunting for a good name results in a favorable restructuring of the code.”

Methods name are disruptive, i.e. *isUserExist()*.

Function Arguments

“The ideal number of arguments for a function is zero (niladic). Next comes one (monadic), followed closely by two (dyadic). Three arguments (triadic) should be avoided where possible. More than three (polyadic) requires very special justification—and then shouldn’t be used anyway.”

I try my best to make methods have the lowest number of arguments. Unfortunately, some cases must add more than three args.

For example, if you want to add a JSON object, you must identify the *database name*, *collection name*, *document name*, *JSON object*, and *index*. See **Figure 34**

```
@PostMapping("/{databaseName}/{collectionName}/{documentName}/{index}")
@PreAuthorize("hasAuthority('ROLE_ADMIN')")
public void putJsonObject(@PathVariable String databaseName,
                        @PathVariable String collectionName,
                        @PathVariable String documentName,
                        @PathVariable String index,
                        @RequestBody Map<String, Object> json) {
    service.addJSON(databaseName, collectionName, documentName, json, index);
    masterNode.notifyAllReplicas();
}
```

Figure 34: Function arguments example.

Flag Arguments

“Flag arguments are ugly. Passing a boolean into a function is a truly terrible practice. It immediately complicates the signature of the method, loudly proclaiming that this function does more than one thing. It does one thing if the flag is true and another if the flag is false!”

There is one situation in which the method can have a flag argument, and to avoid that, I’ve refactored the method and split it into two methods to make each method do one thing. See **Figure 35**.

```
public synchronized void overrideWriteFile(String dirPath, String content) {...}

public synchronized void writeFile(String dirPath, String content) {...}
```

Figure 35: Avoid Flag arguments solution

Don't Repeat Yourself (DRY)

“Duplication may be the root of all evil in software. Many principles and practices have been created for the purpose of controlling or eliminating it.”

To avoid the DRY principle, I implemented generic classes, such as *BTree* and the *LRUCache* classes. See **Figure 36**

```
public class BTree<Key extends Comparable<Key>, Value> {
public class LRUCache<Key, Value> {
```

Figure 36: DRY implementation.

Try/Catch Blocks

“Try/catch blocks are ugly in their own right. They confuse the structure of the code and mix error processing with normal processing. So it is better to extract the bodies of the try and catch blocks out into functions of their own.”

Most try/catch are extracted to their function.

```
public static String readFile(String filename) {
    String content = "";
    try (BufferedReader fileReader = new BufferedReader(new FileReader(filename))) {
        content = readFileHelper(fileReader);
    } catch (IOException e) {
        LOGGER.logError(e);
    }
    return content;
}
```

Figure 37: DRY implementation.

6.4 Comments

“The proper use of comments is to compensate for our failure to express ourself in code. Note that I used the word failure. I meant it. Comments are always failures. We must have them because we cannot always figure out how to express ourselves without them, but their use is not a cause for celebration.”

Comments are rarely used in the project, and the used ones are legal to clarify what the part of this is doing.

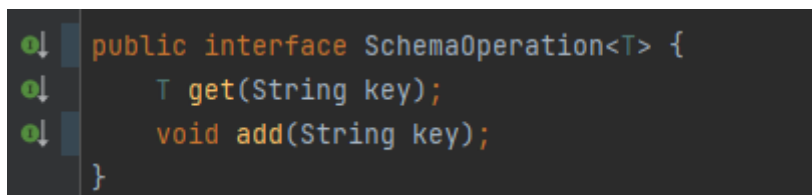
6.5 Objects and Data Structures

“There is a reason that we keep our variables private. We don’t want anyone else to depend on them. We want to keep the freedom to change their type or implementation on a whim or an impulse. Why, then, do so many programmers automatically add getters and setters to their objects, exposing their private variables as if they were public?”

To achieve this goal of clean code, let's aim the objective needs:

1. Variables are private; the access is done through getters and setters.
2. Avoid train wrecks
3. Law of Demeter and low coupling
4. Interfaces

All of these methodologies have been applied to the project.



```
public interface SchemaOperation<T> {  
    T get(String key);  
    void add(String key);  
}
```

Figure 38: Object and data structure example

6.6 Error Handling

Use Exceptions Rather Than Return Codes and Define Exception Classes in Terms of a Caller's Needs

“It is better to throw an exception when you encounter an error. The calling code is cleaner. Its logic is not obscured by error handling.”

Most method has exception thrown if an error occurs, and to do that, I implement an exception factory for handling the request exception from the user.

```
public synchronized void changeUserRole(String username, Role role) {
    List<User> tempUsersGroup = getAllUser();
    if (isUserExist(username)) {
        getUser(username).setRole(role);
        DirectoryCreator.getInstance().overrideWriteFile(USER_FILE.getAbsolutePath(), JSON.toJson(tempUsersGroup));
    } else {
        throw new NotFoundException("USER NOT FOUND !");
    }
}
```

Figure 39: Exception example

Don't Return Null

“I think that any discussion about error handling should include mention of the things we do that invite errors. The first on the list is returning `null`. I can't begin to count the number of applications I've seen in which nearly every other line was a check for `null`”

There is no null value return in DocumentDB project.

Don't Pass Null.

“Returning `null` from methods is bad, but passing `null` into methods is worse. Unless you are working with an API which expects you to pass `null`, you should avoid passing `null` in your code whenever possible.”

Because of the reason above, passing null value is never exists in the DocumentDb project.

6.7 Classes

Classes Should Be Small, Maintaining Cohesion Results in Many Small Classes

“The first rule of classes is that they should be small. The second rule of classes is that they should be smaller than that.”

All classes are small as they can be.

And if there are more complex in one class, I split it into two different classes and keep the single responsibility principle.

```
1 package com.mohammad.masternode.index.btree;
2
3 class Entry {
4     protected Comparable key;
5     protected Object value;
6     protected Node next; // helper field to iterate over array entries
7     public Entry(Comparable key, Object value, Node next) {
8         this.key = key;
9         this.value = value;
10        this.next = next;
11    }
12 }
13
```

Figure 40: small class example

7.0 SOLID Principles

7.1 Introduction

“The SOLID principles tell us how to arrange our functions and data structures into classes, and how those classes should be interconnected. The use of the word “class” does not imply that these principles are applicable only to object-oriented software. A class is simply a coupled grouping of functions and data. Every software system has such groupings, whether they are called classes or not. The SOLID principles apply to those groupings.” - Robert C Martin

SOLID is an acronym for five other class-design principles:

1. S - Single Responsibility Principle.
2. O - Open/closed Principle.
3. L - Liskov Substitution Principle.
4. I - Interface Segregation Principle
5. D -Dependency Inversion Principle

This chapter will discuss how I use these principles in the DocumentDB project according to *Clean Architecture* by *Robert C Martin*.

7.2 Single Responsibility Principle (SRP)

Each class should be responsible for a single part or functionality of the system.

“Make no mistake, there is a principle like that”

All classes have one and only one single job to do. Thus, there is one reason to change the class.

For example, *DatabaseService* in *ReplicaNode* class is only responsible for the operations that are relevant to database *read all databases or read particular one*.

Also, *RestFilter* class is only responsible for filtering the API request and sending it to *RestSecurityConfiguration* class.

I noticed how much the SRP is powerful when we consider that it will provide more fast development, maintainability, and reliability.

7.3 Open/Closed Principle (OCP)

“A software artifact should be open for extension but closed for modification.”

-Bertrand Meyer.

For example, if we want to add another hashing mechanism, you want to implement the parent class (password encoder) and dot the *encode* and *match* method.

```
@Override
public String encode(CharSequence password) {...}

@Override
public boolean matches(CharSequence rawPassword, String encodedPassword) {...}
```

Figure 41: OCP example

7.4 Liskov Substitution Principle (LSP)

*“What is wanted here is something like the following substitution property: If for each object *o1* of type *S* there is an object *o2* of type *T* such that for all programs *P* defined in terms of *T*, the behavior of *P* is unchanged when *o1* is substituted for *o2* then *S* is a subtype of *T*.” - Barbara Liskov*

This were applied in many classes, such as the structure of database document and collection I've implemented *SchemaOperation* interface contains only the major operation **add**, **get**.

7.5 Interface Segregation Principle (ISP)

Clients should not be forced to depend upon interface members they do not use. In other words, do not force any client to implement an interface that is irrelevant to them; it is pretty similar to LSP.

To clarify the idea, if we have Zoo class and Employee interface, the employee interface contain pet the cat, now the chaser class cannot use the pet method, which is why ISP comes.

In my project *SchemaOperation* interface contains only the major operation **add**, **get**.

7.6 Dependency Inversion Principle (DIP)

A class should not depend on low-level concrete classes, instead it should depend on abstractions

It was hard for me to apply this principle, so I couldn't use it in all of my projects

But my example may we can consider the LRU cache as an application for DIP.

```
private final int MAX_CAPACITY;  
private AtomicInteger curSize = new AtomicInteger();  
private ConcurrentHashMap<Key, Node> map;  
private Node head;  
private Node tail;  
private ReentrantReadWriteLock readWriteLock;
```

Figure 42: DIP example

8.0 Effective Java

This chapter will discuss the item that I used in my project after reading *Effective Java* by *Joshua Bloch*

8.1 Creating and Destroying Objects

Item 1: Consider static factory methods instead of constructors

“There is another technique that should be a part of every programmer’s toolkit. A class can provide a public static factory method, which is simply a static method that returns an instance of the class.”

In my project, this role was applied; for example, In *RestUser* class, you can just *create* instead of creating an object to create a new user. See **Figure 43**

```
private RestUser(User user) { this.user = user; }

public static RestUser create(User user){
    return new RestUser(user);
}
```

Figure 43: item 1 example

Item 2: Consider a builder when faced with many constructor parameters

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

The Builder design pattern was used in the project to build users, so as the book mentions builder pattern, the code will be easy to read.

“This client code is easy to write and, more importantly, easy to read”

```
USER_GROUP.add(new User.UserBuilder()
    .username(username)
    .password(password)
    .role(role)
    .build());
```

Figure 44: item 2 example

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

“A singleton is simply a class that is instantiated exactly once.”

The singleton pattern was also used in my project, so this item will be included, for example, in all directories classes (*DirectoryCreator*, *DirectoryRemover*) that have the singleton property.

```
public static DirectoryRemover getInstance() {
    synchronized (DirectoryRemover.class) {
        if (remover == null)
            remover = new DirectoryRemover();
    }
    return remover;
}
```

Figure 45: item 3 example

Item 4: Enforce noninstantiability with a private constructor

“Such utility classes were not designed to be instantiated: an instance would be nonsensical.”

And as the book mentioned, this item was used in the utility classes. See **Figure 46**

```
private JSON(){
    throw new AssertionError();
}
```

Figure 46: item 4 example

Item 5: Prefer dependency injection to hardwiring resources

“Although dependency injection greatly improves flexibility and testability, it can clutter up large projects, which typically contain thousands of dependencies.”

This item was used in services classes and maven as well. See **Figure 47**.

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter</artifactId>
</dependency>
```

```
@Autowired
private CollectionService service;
```

Figure 47: item 5 example

Item 6: Avoid creating unnecessary objects

In my project, there are no unneeded objects used.

```
public class LRUCache<Key, Value> {  
    private final int MAX_CAPACITY;  
    private AtomicInteger curSize = new AtomicInteger();  
    private ConcurrentHashMap<Key, Node> map;  
    private Node head;  
    private Node tail;  
    private ReentrantReadWriteLock readWriteLock;  
}
```

Figure 48: item 6 example

Item 8: Avoid finalizers and cleaners

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

Finalizers and cleaners were avoided in my project.

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

I try my best to use try-with-resources to try-finally.

```
try (BufferedWriter writer = new BufferedWriter(new FileWriter(file))) {  
    writer.write(content);  
} catch (IOException e) {  
    LOGGER.logError(e);  
}
```

Figure 49: item 9 example

8.2 Methods Common to All Objects

Item 12: Always override toString

“providing a good toString implementation makes your class much more pleasant to use and makes systems using the class easier to debug.”

All toString methods were overridden in all DocumentDB projects.

```
@Override
public String toString() {
    return "{" +
        "userID:" + userID +
        ", username:" + username + '\'' +
        ", password:" + password + '\'' +
        ", role:" + role +
        '\'';
}
```

Figure 50: item 12 example

Item 14: Consider implementing Comparable

“By implementing Comparable, you allow your class to interoperate with all of the many generic algorithms and collection implementations that depend on this interface. You gain a tremendous amount of power for a small amount of effort.”

Also, because of comparable's advantage by implementing it with generic algorithms (in my case *BTree*). I implement *comparable*.

“By implementing Comparable, you allow your class to interoperate with all of the many generic algorithms and collection implementations that depend on this interface. You gain a tremendous amount of power for a small amount of effort.”

```
public class BTree<Key extends Comparable<Key>, Value> {
```

Figure 51: item 14 example

8.3 Classes and Interfaces

Item 15: Minimize the accessibility of classes and members

“Information hiding is important for many reasons, most of which stem from the fact that it decouples the components that comprise a system, allowing them to be developed, tested, optimized, used, understood, and modified in isolation.”

I applied this item in all my projects. There are no members that could be accessed with no *getter* or *setter*, and for classes, there are some classes I made it *package-private*; there is no use for them outside the package (*Node* class and *Entry* class for *BTree imp*).

```
class Node {
    protected int childrenNum;
    protected Entry[] children = new Entry[getMaxChildrenNum()];

    public Node(int childrenNum) { this.childrenNum = childrenNum; }
}

class Entry {
    protected Comparable key;
    protected Object value;
    protected Node next;    // helper field to iterate over array entries
    public Entry(Comparable key, Object value, Node next) {
        this.key = key;
        this.value = value;
        this.next = next;
    }
}

public class DirectoryRemover {
    private static final AppLogger LOGGER = AppLogger.create("DirectoryRemover LOGGER");
    private static DirectoryRemover remover;
```

Figure 52: item 15 example

Item 16: In public classes, use accessor methods, not public fields.

Also, to achieve *encapsulation*, all parameters can only be accessed by the *getter* or *setter*.

```
public String getUsername() { return username; }
```

Figure 53: item 16 example

Item 17: 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.”

For example, in *User* class and because I use *Builder* pattern, I remove all *setter*. Thus, most class members are private or final, so there is no *setter*.

Item 18: 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. Even then, inheritance may lead to fragility if the subclass is in a different package from the superclass and the superclass is not designed for inheritance.”

I apply this item and use forwarding instead of inheritance in most of my projects.

```
public class Document implements SchemaOperation {
    private String documentName;
    private BTree<String, String> documentSchema = new BTree<String, String>();
}
```

Figure 54: item 18 example

Item 20: Prefer interfaces to abstract classes

“When a class implements an interface, the interface serves as a type that can be used to refer to instances of the class. That a class implements an interface should therefore say something about what a client can do with instances of the class. It is inappropriate to define an interface for any other purpose.”

Also, *“interfaces should be used only to define types. They should not be used merely to export constants.”*

I apply this item to my code and use interface rather than abstract class. See **Figure-55**.

```
public interface SchemaOperation<T> {
    T get(String key);
    void add(String key);
}
```

Figure 55: item 20 example

Item 24: Favor static member classes over nonstatic

“there are four different kinds of nested classes, and each has its place. If a nested class needs to be visible outside of a single method or is too long to fit comfortably inside a method, use a member class. If each instance of a member class needs a reference to its enclosing instance, make it nonstatic; otherwise, make it static. Assuming the class belongs inside a method, if you need to create instances from only one location and there is a preexisting type that characterizes the class, make it an anonymous class; otherwise, make it a local class.”

This item was applied all over my projects, for example, the directories classes.

Item 25: Limit source files to a single Top-level class

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

This item was applied in all code; there are no two classes or interfaces in a single source file except *User* class, because of the *Builder* pattern.

8.4 Generics

Item 26: Don't use raw types

“using raw types can lead to exceptions at runtime, so don't use them.”

The raw type wasn't used in my code, unless, for example, in *BTree*, I used raw type of comparable to help compare and refactor the *tree*.

```
class Entry {
    protected Comparable key;
    protected Object value;
    protected Node next;    // helper field to iterate over array entries
    public Entry(Comparable key, Object value, Node next) {
        this.key = key;
        this.value = value;
        this.next = next;
    }
}
```

Figure 56: item 26 example

Item 28: Prefer lists to arrays

“As a rule, arrays and generics don't mix well. If you find yourself mixing them and getting compile-time errors or warnings, your first impulse should be to replace the arrays with lists.”

I used *lists* rather than *arrays* in all of my projects.

Item 29: Favor generic types

“generic types are safer and easier to use than types that require casts in client code.”

And because of that reason, I used generic types in most of my projects.

For example, to add a JSON object, the structure will be of type *BTree*.

```
private BTree<String, String> documentSchema = new BTree<String, String>();
```

Figure 57: item 29 example

Item 30: Favor generic methods

“generic methods, like generic types, are safer and easier to use than methods requiring their clients to put explicit casts on input parameters and return values. Like types, you should make sure that your methods can be used without casts, which often means making them generic. And like types, you should generify existing methods whose use requires casts. This makes life easier for new users without breaking existing clients.”

Same in here, I used generic methods. For example, in *LRUCache*, I used a generic method to add or evict value.

```
public void evict(K key) {  
    Node node = map.get(key);  
    if (node != null) {  
        removeNode(node);  
        map.remove(key);  
    }  
}
```

Figure 58: item 30 example

8.5 Enums and Annotations

Item 34: Use enums instead of int constants

“the advantages of enum types over `int` constants are compelling. Enums are more readable, safer, and more powerful.”

I used enums instead of constant, for example, to define the *role* of users.

```
public enum Role {
    ROLE_ADMIN, ROLE_USER, ROLE_INVALID_USER
}
```

Figure 59: item 34 example

Item 40: Consistently use the Override annotation

“the compiler can protect you from a great many errors if you use the `Override` annotation on every method declaration that you believe to override a supertype declaration, with one exception.”

I apply this item to all of my projects whenever I need it.

```
@Override
protected void configure(AuthenticationManagerBuilder auth) { auth.authenticationProvider(authProvider()); }

@Override
protected void configure(HttpSecurity http) throws Exception {
    http.csrf().disable()
        .authorizeRequests(authorize ->{
            authorize.antMatchers("/master/**").hasAuthority("ROLE_ADMIN");
        })
        .authorizeRequests()
        .anyRequest().authenticated()
        .and().httpBasic().authenticationEntryPoint(entryPoint)
        .and().sessionManagement().sessionCreationPolicy(SessionCreationPolicy.STATELESS);
    http.addFilterAfter(new RestFilter(), BasicAuthenticationFilter.class);
}
```

Figure 60: item 40 example

8.6 Lambdas and Streams

Item 42: Prefer lambdas to anonymous classes

“lambdas make it so easy to represent small function objects that it opens the door to functional programming techniques that were not previously practical in Java.”

I applied this item to benefit from lambdas; for example, in *CacheUtils* class, I used lambda to add a database to cache. If the user calls it and gets it from the directory, the first needs to call it again for the second time.

```
databaseGroup.forEach(data -> GLOBAL_CACHE.put(data.getDatabaseName(),data));
```

Figure 61: item 42 example

Item 43: 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 try to use this item as I can; for example, in *MasterNode* class to update replicas.

```
REPLICA_GROUP.forEach(Observer::update);
```

Figure 62: item 43 example

8.7 Methods

Item 49: Check parameters for validity

“Most methods and constructors have some restrictions on what values may be passed into their parameters.”

I applied this item to my project. For example, if the user needs to get unexist data, the system will throw *NotFoundException*.

Item 51: Design method signatures carefully

“Choose method names carefully, Don’t go overboard in providing convenience methods, and Avoid long parameter lists”

When I read this item, its *clean code* principles, so, because of that, this Item was applied already by applying *clean code* principle.

Item 52: Use overloading judiciously

“just because you can overload methods doesn’t mean you should. It is generally best to refrain from overloading methods with multiple signatures that have the same number of parameters. In some cases, especially where constructors are involved, it may be impossible to follow this advice.”

In my case, I overload some methods because I need to.

For example, in *Document* class, there are two ways to add JSON objects; the first is when the client has an index to order the objects; otherwise, the *_objectID* will automatically be generated.

```
public void add(String jsonObject, String indexProperty) {...}  
  
@Override  
public void add(String jsonObject) {...}
```

Figure 63: item 52 example

8.7 General Programming

Item 57: 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, keep methods small and focused ”

All variables are in the smallest scope it can be.

Item 58: 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. ”

I used *for-each* loop wherever it is needed. For example, in any list that I’ve declared, I used *for-each* method to send or find the object of this list.

```
databaseGroup.forEach(data -> GLOBAL_CACHE.put(data.getDatabaseName(),data));
```

Figure 64: item 58 example.

Item 59: Know and use the libraries

“ don’t reinvent the wheel. If you need to do something that seems like it should be reasonably common, there may already be a facility in the libraries that does what you want. If there is, use it”

I applied this item to my project; for example, I used *Gson Jackson* libraries to check the validation of any JSON object and the conversion to a JSON object.

```
public static String toJson(Object json){
    return gson.toJson(json);
}
public static boolean isValidJson(String jsonObject) {
    try {
        gson.fromJson(jsonObject, Object.class);
        return true;
    } catch (com.google.gson.JsonSyntaxException ex) {
        return false;
    }
}
```

Figure 65: item 59 example

Item 60: Avoid float and double if exact answers are required

“Don’t use `float` or `double` for any calculations that require an exact answer.”

I applied this item to my project; there is no use of *double* or *float*, and the reason is all of the calculation needs an exact answer.

```
public class BTree<Key extends Comparable<Key>, Value> {  
    private static final int MAX_CHILDREN_NUM = 5;  
    public class LRUCache<K, V> {  
        private final int MAX_CAPACITY;
```

Figure 66: item 60 example

Item 64: Refer to objects by their interfaces

“If there is no appropriate interface, just use the least specific class in the class hierarchy that provides the required functionality.”

This item was applied to my project; for example, when I declare any *ArrayList* reference, it by the *List<>* interface then assign it to *ArrayList* object.

```
private static final List<User> userGroup = new ArrayList<>();
```

Figure 67: item 64 example

8.8 Exceptions

Item 69: Use exceptions only for exceptional conditions

“throw checked exceptions for recoverable conditions and unchecked exceptions for programming errors. When in doubt, throw unchecked exceptions.”

I applied this item in my project; for example, in *UserService* class, if the user is not found (and this unchecked exception), the system will throw it if it happens.

```
} else {  
    throw new NotFoundException("USER NOT FOUND !");  
}
```

Figure 68: item 69 example

Item 70: Use checked exceptions for recoverable conditions and runtime exceptions for programming errors

“ throw checked exceptions for recoverable conditions and unchecked exceptions for programming errors. When in doubt, throw unchecked exceptions.”

This item was also applied in all of the REST API calls, and the reason behind that is most API calls exception is *runtime exception* rather than unchecked exception.

```
public class UserConflictException extends RuntimeException{  
    public UserConflictException(String message) { super(message); }  
}
```

Figure 69: item 70 example

Item 71: Avoid unnecessary use of checked exceptions

“when used sparingly, checked exceptions can increase the reliability of programs; when overused, they make APIs painful to use. If callers won’t be able to recover from failures, throw unchecked exceptions. If recovery may be possible and you want to force callers to handle exceptional conditions, first consider returning an optional. Only if this would provide insufficient information in the case of failure should you throw a checked exception.”

This item applied to my project, so no unnecessary checked exception was used.

Item 72: 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.”

There are a lot of standard exceptions used in my projects, such as *IOException* and *InterruptedException*.

Item 77: Don’t ignore exceptions.

“The advice in this item applies equally to checked and unchecked exceptions. Whether an exception represents a predictable exceptional condition or a programming error, ignoring it with an empty `catch` block will result in a program that continues silently in the face of error.”

In my projects, there is no ignored exception.

8.9 Concurrency

Item 78: Synchronize access to shared mutable data

*“The **synchronized** keyword ensures that only a single thread can execute a method or block at one time.”*

Because my DocumentDB is a microservices project, for sure, there are many data shared. Thus, this item must be applied to my project; many synchronized keywords make it *thread-safe*.

```
public synchronized void overrideWriteFile(String dirPath, String content) {
| public synchronized void writeFile(String dirPath, String content) {
```

Figure 70: item 78 example

Item 79: Avoid excessive synchronization

“to avoid deadlock and data corruption, never call an alien method from within a synchronized region.”

I try my best to apply this item as much as possible; here is an example of using this item in my code.

```
public synchronized void changeUserPassword(String username, String newPassword) {
    List<User> tempUsersGroup = getAllUser();
    if (isUserExist(username)) {
        getUser(username).setPassword(newPassword);
        DirectoryCreator.getInstance().overrideWriteFile(USER_FILE.getAbsolutePath(), JSON.toJson(tempUsersGroup));
    } else {
        throw new NotFoundException("USER NOT FOUND !");
    }
}
|
}
```

Figure 71: item 79 example

Item 84: Don't depend on the thread scheduler

“do not depend on the thread scheduler for the correctness of your program. The resulting program will be neither robust nor portable.”

This item was applied to my project; see an example in *LRUCache* class.

```
private void removeNode(Node node) {
    try {
        readWriteLock.writeLock().lock();
        Node prev = node.prev;
        Node next = node.next;
        prev.next = next;
        next.prev = prev;
    } finally {
        readWriteLock.writeLock().unlock();
    }
}
|
}
```

Figure 72: item 80 example

9.0 DocumentDB Issues

9.1 Introduction

This chapter will discuss the security issues and Scalability/Consistency Issues in my DocumentDB microservice project.

9.2 Security Issues

The only security authentication in my DocumentDB is *BasicAuth*, which can be attacked in Brute Force. Also, there is no HTTPS(SSL/TLS) there, for any penetration tester with zero level of security knowledge can get some great information in this system.

For example, if the hacker used *Nmap/ZenmapGUI* (the first step to *penTest* any system) to check the open ports, he will get this valuable information.

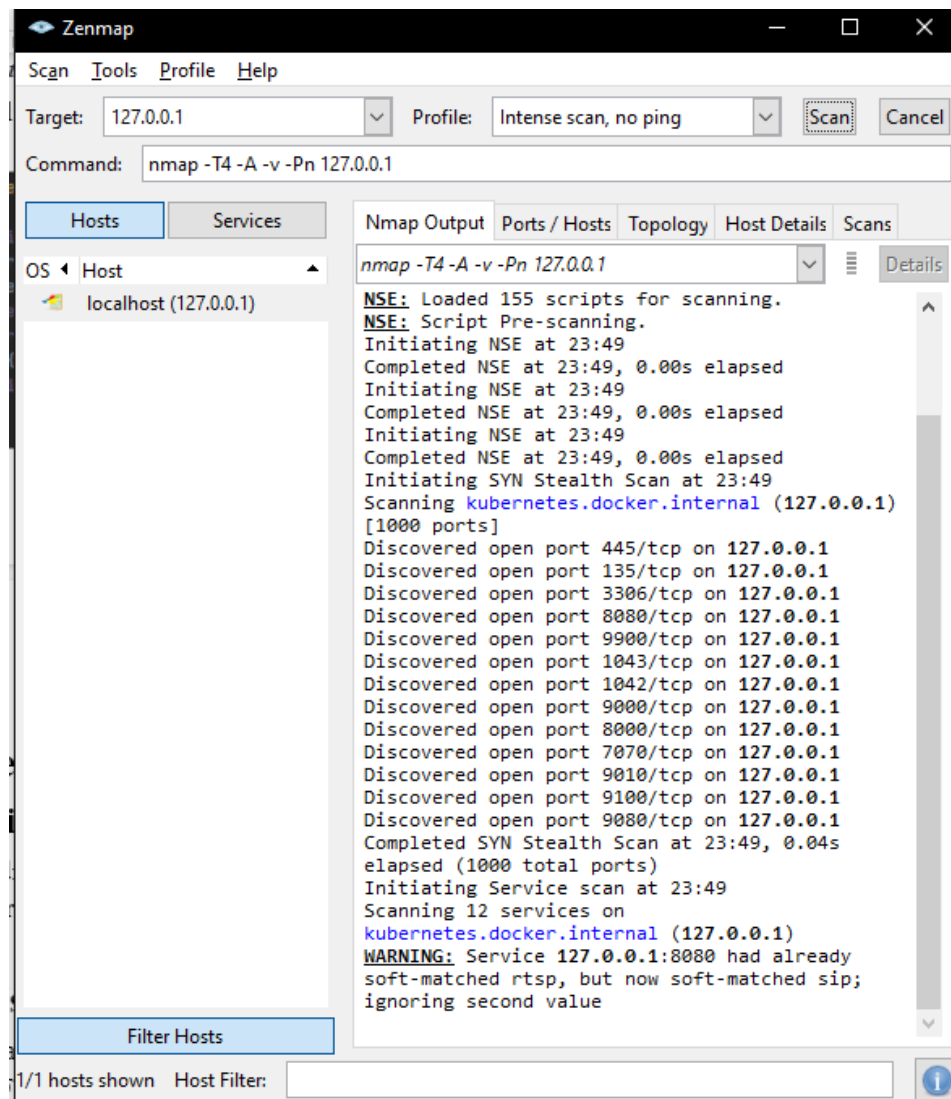


Figure 72: first *Nmap* scan

As you can see in **Figure 72**, he can see the open ports, and that is very dangerous, and the reason is attackers can exploit shortcomings like weak credentials, no two-factor authentication, or even vulnerabilities in the application itself.

Thus, if we wait for *Nmap* to finish this scan, we will get these results.

```
8000/tcp open  http-alt
|_http-title: Site doesn't have a title.
|_http-auth:
|_ HTTP/1.1 401 \x0D
|_   Basic
|_   Realm
|_   -
|_   DocumentDB
|_http-methods:
|_ Supported Methods: GET HEAD POST OPTIONS
|_fingerprint-strings:
|_ FourOhFourRequest:
|_   HTTP/1.1 401
|_   WWW-Authenticate: Basic Realm - DocumentDB
|_   Content-Length: 75
|_   Date: Wed, 09 Feb 2022 21:49:20 GMT
|_   Connection: close
|_   HTTP Status 401 - Full authentication is required to access this resource
|_GetRequest:
|_   HTTP/1.1 401
|_   WWW-Authenticate: Basic Realm - DocumentDB
|_   X-Content-Type-Options: nosniff
|_   X-XSS-Protection: 1; mode=block
|_   Cache-Control: no-cache, no-store, max-age=0, must-revalidate
|_   Pragma: no-cache
|_   Expires: 0
|_   X-Frame-Options: DENY
|_   Content-Length: 75
|_   Date: Wed, 09 Feb 2022 21:49:20 GMT
|_   Connection: close
|_   HTTP Status 401 - Full authentication is required to access this resource
|_X11Probe:
|_   HTTP/1.1 400
|_   Content-Type: text/html; charset=utf-8
|_   Content-Language: en
|_   Content-Length: 2026
|_   Date: Wed, 09 Feb 2022 21:49:20 GMT
|_   Connection: close
|_   <!doctype html><html lang="en"><head><title>HTTP Status 400
|_   Request</title><style type="text/css">body {font-family:Tahoma,Arial,sans-serif;} h1, h2, h3,
|_   b {color:white;background-color:#525D76;} h1 {font-size:22px;} h2 {font-size:16px;} h3 {font-
|_   size:14px;} p {font-size:12px;} a {color:black;} .line {height:1px;background-
|_   color:#525D76;border:none;}</style></head><body><h1>HTTP Status 400
|_   Request</h1><hr class="line" /><p><b>Type</b> Exception Report</p><p><b>Message</b> Invalid
|_   character found in method name [10x000x0b0x000x000x000x000x000x000x000x000x000x00...]. HTTP method
|_   names must be tokens</p><p><b>Description</b> The server cannot or will not process the request due
```

Figure 73: the MasterNode open port

And as you can see, the header contains the *BasicAuth* type of security, which can lead to DDOS attack, Bruteforce attack, and more.

The same thing repeats with port **9000** and the *API Gateway*, but the hacker will know that this is a **SpringBoot** Application and that he will find some way to hack the system. See **Figures 73**.

```

8080/tcp open  rtsp
|_http-title: Site doesn't have a title (application/json).
|_rtsp-methods: ERROR: Script execution failed (use -d to debug)
| fingerprint-strings:
|   GetRequest:
|     HTTP/1.0 404 Not Found
|     Content-Type: application/json
|     Content-Length: 7452
|     {"timestamp":"2022-02-09T21:49:15.050+00:00","path":"/","status":404,"error":"Not
Found","message":null,"requestId":"d9eb6cd5-11","trace":"org.springframework.web.server.ResponseStatusException:
404 NOT_FOUND
|     org.springframework.web.reactive.resource.ResourceWebHandler.lambda$handle$1(ResourceWebHandler.java:408)
|     Suppressed: The stacktrace has been enhanced by Reactor, refer to additional information below:
|     Error has been observed at the following site(s):
|     *__checkpoint
|     org.springframework.cloud.gateway.filter.WeightCalculatorWebFilter [DefaultWebFilterChain]
|     *__checkpoint
|     org.springframework.boot.actuate.metrics.web.reactive.server.MetricsWebFilter [DefaultWebFilterChain]
|     *__checkpoint
|     HTTP GET "/" [ExceptionHandlerWebHandler]
|     Original Stack Trace:
|   HTTPOptions:
|     HTTP/1.0 404 Not Found
|     Content-Type: application/json
|     Content-Length: 7456
|     {"timestamp":"2022-02-09T21:49:15.061+00:00","path":"/","status":404,"error":"Not
Found","message":null,"requestId":"df94bf6d-12","trace":"org.springframework.web.server.ResponseStatusException:
404 NOT_FOUND
|     org.springframework.web.reactive.resource.ResourceWebHandler.lambda$handle$1(ResourceWebHandler.java:408)
|     Suppressed: The stacktrace has been enhanced by Reactor, refer to additional information below:
|     Error has been observed at the following site(s):
|     *__checkpoint
|     org.springframework.cloud.gateway.filter.WeightCalculatorWebFilter [DefaultWebFilterChain]
|     *__checkpoint
|     org.springframework.boot.actuate.metrics.web.reactive.server.MetricsWebFilter [DefaultWebFilterChain]
|     *__checkpoint
|     HTTP OPTIONS "/" [ExceptionHandlerWebHandler]
|     Original Stack Trace:
|_
9000/tcp open  cslistener?
| fingerprint-strings:
|   GetRequest, HTTPOptions:
|     HTTP/1.1 401
|     WWW-Authenticate: Basic Realm - DocumentDB
|     X-Content-Type-Options: nosniff
|     X-XSS-Protection: 1; mode=block
|     Cache-Control: no-cache, no-store, max-age=0, must-revalidate
|     Pragma: no-cache
|     Expires: 0
|     X-Frame-Options: DENY
|     Content-Length: 75
|     Date: Wed, 09 Feb 2022 21:49:20 GMT
|     Connection: close
|     HTTP Status 401 - Full authentication is required to access this resource
|   RPCCheck, RTSPRequest:
|     HTTP/1.1 400
|     Content-Type: text/html; charset=utf-8
|     Content-Language: en
|     Content-Length: 435
|     Date: Wed, 09 Feb 2022 21:49:20 GMT
|     Connection: close
|     <!doctype html><html lang="en"><head><title>HTTP Status 400
|     Request</title><style type="text/css">body {font-family:Tahoma,Arial,sans-serif;} h1, h2, h3, b
|     {color:white;background-color:#525D76;} h1 {font-size:22px;} h2 {font-size:16px;} h3 {font-size:14px;} p {font-
|     size:12px;} a {color:black;} .line {height:1px;background-color:#525D76;border:none;}</style></head><body><h1>HTTP
|     Status 400
|     Request</h1></body></html>
|_

```

Figure 73: more info from *Nmap*

Now. You can see how much information the attacker can get from a single scan that costs nothing.

The attacker now knows the gateway, replica, and controller node.

Also, he can track each request traffic using the *WireShark* application

9.3 Scalability/Consistency Issues

9.3.1 Scalability Issues

For scalability issues, I see that the way of repeating the replica is not the efficient way, but the right way to do that is to make every node reader and writer; also the way of scaling which in my case is *horizontal scaling* is not correct the correct way is to make sharding with simply hashing the key of every document id and then send it to the new server, so, when the request comes to that document simply redirect it to that server.

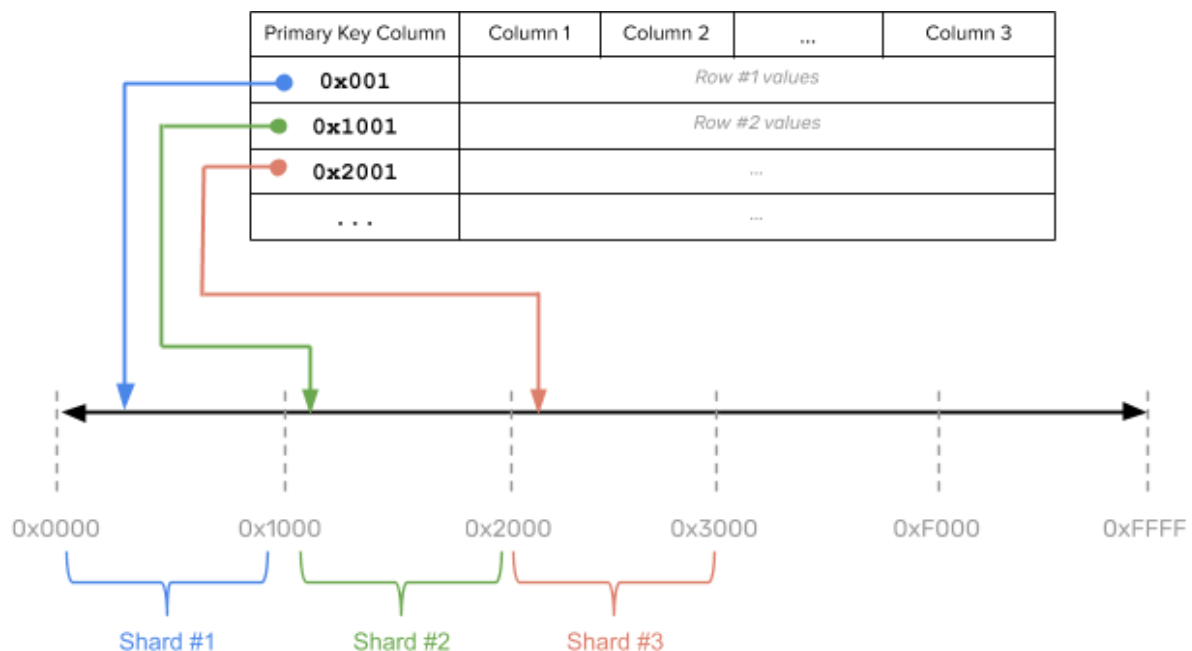


Figure 74: sharding example

9.3.2 Consistency Issues

For consistency issues, the first thing is cache, every cache are done in the server (node), so every node got its cache; therefore I cannot handle it right, so the way to make the replicas done correctly is to delete the running replica after the update done, then create a new replica and that cost a lot.

The correct way is to generate a distributed global cache.

So when the node is killed or stopped for any reason, there is no need to worry because the global cache will already store all data.

REFERENCES

<https://learning.oreilly.com/library/view/spring-security-in/9781617297731/>
<https://learning.oreilly.com/library/view/head-first-design/0596007124/>
<https://learning.oreilly.com/library/view/nosql-for-mere/9780134029894/>
<https://learning.oreilly.com/library/view/microservices-with-spring/9781801072977/>
<https://learning.oreilly.com/library/view/effective-java/9780134686097/>
<https://learning.oreilly.com/library/view/clean-architecture-a/9780134494272/>
<https://learning.oreilly.com/library/view/clean-code-a/9780136083238/>
<https://learning.oreilly.com/videos/apache-kafka-series/9781789346534/>
<https://learning.oreilly.com/library/view/nosql-distilled-a/9780133036138/>
<https://learning.oreilly.com/library/view/professional-nosql/9781118167809/>
<https://learning.oreilly.com/library/view/pro-java-clustering/9781484229859/>
<https://learning.oreilly.com/library/view/database-design-for/9780136788133/>
<https://algs4.cs.princeton.edu/home/>
<https://docs.mongodb.com/manual/introduction/>
<https://docs.aws.amazon.com/documentdb/latest/developerguide/how-it-works.html>
https://en.wikipedia.org/wiki/Basic_access_authentication
[https://en.wikipedia.org/wiki/Cache_replacement_policies#Least_recently_used \(LRU\)](https://en.wikipedia.org/wiki/Cache_replacement_policies#Least_recently_used_(LRU))
<https://www.geeksforgeeks.org/introduction-of-b-tree-2/>
<https://en.wikipedia.org/wiki/B-tree>
<https://microservices.io/patterns/apigateway.html>
<https://docs.oracle.com/javase/tutorial/essential/io/file.html>
<https://learning.oreilly.com/library/view/head-first-java/9781492091646/>
<https://nmap.org/docs.html>
<https://www.guru99.com/indexing-in-database.html>