# PREFACE

Document databases is one of the most powerful and useful NoSQL databases, in this report I’m going to talk about my journey to implement a Document database using java.

Through this report we will be talking about several things from the document databases, reading and writing to it, clustering, indexing, multithreading, java sockets, the data structure used in the implementation, scalability, consistency, security issues in the implementation, and many more.

Also, this report will discuss the clean code principles from the book “**Clean code by Uncle Bob**”,

Effective java principles and items according to **Joshua Bloch**, **SOLID** principles, Design patterns, and **DevOps** practices in this project.

Before we start let’s see the requirements for this project.

We need to bulid a document database which has:

1. Reads are the majority of transactions
2. Writes are done only rarely only through a connection to the database controller
3. Must have fast reads (in memory caching)
4. It must be JSON structure, not XML
5. Must have indexing and it must be in efficient manner
6. Can scale horizontally
7. Export and import the databases

# PRERQUISITE

## Introduction

This chapter will discuss the theory behind document databases.

## What is Document database?

According to AWS website: “**A document database is a type of nonrelational database that is designed to store and query data as JSON-like documents. Document databases make it easier for developers to store and query data in a database by using the same document-model format they use in their application code. The flexible, semi structured, and hierarchical nature of documents and document databases allows them to evolve with applications’ needs. The document model works well with use cases such as catalogs, user profiles, and content management systems where each document is unique and evolves over time. Document databases enable flexible indexing, powerful ad hoc queries, and analytics over collections of documents**.”

## What is a Document?

A document is a record in a document database.

Documents store data in field-value pairs. The value can be a variety of types and structures including strings, numbers, dates, objects, or array. Documents can be stored in formats like JSON, BSON, and XLM.

Example of a JSON document:

|  |
| --- |
|  |

## What is a Collection?

Collection is a group of documents. Collections typically store documents that have similar contents i.e., products, cares … etc.

One of a main advantages of document databases that not all documents in a collection are required to have the same fields.

This Figure shows the relationship between the collection and the document

|  |
| --- |
|  |

## Database

Database is a group of collections. You can imagine these three as tree where the database is the root node, and the collections is a child of the root node, and every collection node has many children’s which is the documents.

This figure shows the relationship between all of them:

|  |
| --- |
|  |

The summarization of the above sections if you want to develop a document database you should have a database which contains a group of collections, and the collection should contain a group of related documents.

## Indexing

Indexing is a way to optimize the performance of a database by minimizing the number of disk accesses required when a query is processed, it uses a data structure technique.

|  |
| --- |
|  |

Clustering  
a cluster is a collection of linked computers and works together as if they are one machine.

## Scaling:

There are two types of scaling the first one is vertical scaling and the second is the horizontal scaling:

1. Vertical scaling: is adding more power to your current machines i.e., upgrade the CPU, SSDs … etc.
2. Horizontal scaling: this refers to adding additional nodes to the clusters.

## What have I implemented?

After a lot of thinking reading searching failing repeatedly, the final Idea to implement our Document data base will be as follow:

1. The communication between server and client will be using java sockets
2. Indexing will be done using a B+Tree
3. Caching will be an LRU cache done using a LinkedHashMap
4. Security: uses an encrypted password to access the administration tasks
5. The JSON processing library will be Jackson
6. Clustering will be done by running multiple jar files and saving the ports on the main controller

# Tools and Recourses

## Introduction

In this chapter I will share some tools that’s helped me with this project

## IntelliJ IDEA Premium

|  |
| --- |
| IntelliJ IDEA logo |

A very helpful and powerful IDE for java programmers it has a lot of useful features such as code completion, refactoring … etc.

## Github

A well-known website almost every programmer now-a day use it, for me it was very useful, the ability to access the past commits was amazing.

## O'Reilly Media

O’Reilly media -known before as safari books online- is tremendous source of information especially in IT fields, it gives you the books that you need with easy noticeable surfing -unlike pdf way-.

## GitHub Copilot

GitHub Copilot is an AI pair programmer that helps you write code faster and with less work. GitHub Copilot draws context from comments and code and suggests individual lines and whole functions instantly. GitHub Copilot is powered by OpenAI Codex, a new AI system created by OpenAI. The GitHub Copilot technical preview is available as an extension for Visual Studio Code, Neovim, and the JetBrains suite of IDEs.

|  |
| --- |
| how it works |

This amazing tool was very helpful in this project!

An example of this working is:

|  |
| --- |
|  |

# Data structures

## Introduction

In this chapter I will talk about the data structures I used in this project

## B+ Tree

A B+ Tree is simply a balanced binary search tree, in which all data is stored in the leaf nodes, while the internal nodes store just the indices. Each leaf is at the same height and all leaf nodes have links to the other leaf nodes. The root node always has a minimum of two children.

### Properties of B+ Trees

1. All data is stored in the leaf nodes, while the internal nodes store just the indices.
2. Each leaf is at the same height.
3. All leaf nodes have links to the other leaf nodes.
4. The root node has a minimum of two children.
5. Each node except root can have a maximum of m children and a minimum of m/2 children.
6. Each node can contain a maximum of m-1 keys and a minimum of ⌈m/2⌉ - 1 keys.

### Time complexity of the B+Tree

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

### Inserting

1. Since every element is inserted into the leaf node, go to the appropriate leaf node
2. Insert the key into the leaf node.

Case 1:

If the leaf is not full, insert the key into the leaf node in increasing order.

Case 2:

1. If the leaf is full, insert the key into the leaf node in increasing order and balance the tree in the following way
2. Break the node at m/2th position
3. Add m/2th key to the parent node as well
4. If the parent node is already full, follow steps 2 to 3

### Searching

1. Start from the root node. Compare k with the keys at the root node [k1, k2, k3,......km - 1].
2. If k < k1, go to the left child of the root node.
3. Else if k == k1, compare k2. If k < k2, k lies between k1 and k2. So, search in the left child of k2.
4. If k > k2, go for k3, k4,...km-1 as in steps 2 and 3.
5. Repeat the above steps until a leaf node is reached.
6. If k exists in the leaf node, return true else return false.

### Deleting

We have two cases for deleting

Case 1:

The key to be deleted is present only at the leaf node not in the indexes (or internal nodes). There are two cases for it:

1- There is more than the minimum number of keys in the node. Simply delete the key

2- There is an exact minimum number of keys in the node. Delete the key and borrow a key from the immediate sibling. Add the median key of the sibling node to the parent.

Case 2:

The key to be deleted is present in the internal nodes as well. Then we have to remove them from the internal nodes as well. There are the following cases for this situation.

1- If there is more than the minimum number of keys in the node, simply delete the key from the leaf node and delete the key from the internal node as well.  
Fill the empty space in the internal node with the in order successor.

2- If there is an exact minimum number of keys in the node, then delete the key and borrow a key from its immediate sibling (through the parent).  
Fill the empty space created in the index (internal node) with the borrowed key.

3- This case is similar to Case II(1) but here, empty space is generated above the immediate parent node.  
After deleting the key, merge the empty space with its sibling.  
Fill the empty space in the grandparent node with the in order successor.

## Hash-Table

Very popular data structure, almost it is Used nearly in every program

Hash Table is a data structure which stores data in an associative manner. In a hash table, data is stored in an array format, where each data value has its own unique index value. Access of data becomes very fast if we know the index of the desired data.

Thus, it becomes a data structure in which insertion and search operations are very fast irrespective of the size of the data. Hash Table uses an array as a storage medium and uses hash technique to generate an index where an element is to be inserted or is to be located from.

### ConcurrentLinkedQueue

An unbounded thread-safe [queue](https://docs.oracle.com/javase/7/docs/api/java/util/Queue.html) based on linked nodes. This queue orders elements FIFO (first-in-first-out). The *head* of the queue is that element that has been on the queue the longest time. The *tail* of the queue is that element that has been on the queue the shortest time. New elements are inserted at the tail of the queue, and the queue retrieval operations obtain elements at the head of the queue. A ConcurrentLinkedQueue is an appropriate choice when many threads will share access to a common collection. Like most other concurrent collection implementations, this class does not permit the use of null elements.

## LinkedHashMap

Hash table and linked list implementation of the Map interface, with predictable iteration order. This implementation differs from HashMap in that it maintains a doubly-linked list running through all of its entries. This linked list defines the iteration ordering, which is normally the order in which keys were inserted into the map (*insertion-order*). Note that insertion order is not affected if a key is *re-inserted* into the map. (A key k is reinserted into a map m if m.put(k, v) is invoked when m.containsKey(k) would return true immediately prior to the invocation.)

### ArrayList

Resizable-array implementation of the List interface. Implements all optional list operations, and permits all elements, including null. In addition to implementing the List interface, this class provides methods to manipulate the size of the array that is used internally to store the list. (This class is roughly equivalent to Vector, except that it is unsynchronized.)

# Indexing

Indexing is a way to optimize the performance of a database by minimizing the number of disk accesses required when a query is processed. It is a data structure technique which is used to quickly locate and access the data in a database.

For the indexing implementation of my database, I used B+Tree to do it.

The main functionality of my indexer is from the book “Algorthims by Robert Sedgwick Kevin Wayne”

|  |
| --- |
|  |

his example is very similar to what I done in my project code, but instead of symbol table I used B+Tree and instead of queue I used Array List.

## Indexer

This is an interface to define the behavior of every indexer that can be implemented in my design

|  |
| --- |
| void makeIndexOn(String property, ArrayNode jsonNode);  ArrayList<JsonNode> get(String property, String exactValue);  void addToIndexed(String property, JsonNode node);  String[] getAllPropertyIndexed();  void addToAllIndexed(JsonNode node);  void deleteFromIndexed(String property, JsonNode node);  void deleteFromAllIndexed(JsonNode node);  void deleteSpecificKey(String property, String key);  boolean has(String key);  BTree<String, BTree<String, ArrayList<JsonNode>>> getIndexed();  void setIndexed(BTree<String, BTree<String, ArrayList<JsonNode>>> indexed); |

## BtreeIndexer

This class is the implementation of the indexer it uses nested B+Tree to store the indexes

|  |
| --- |
| private BTree<String, BTree<String, ArrayList<JsonNode>>> index; |
| |  | | --- | | public void makeIndexOn(String property, ArrayNode jsonNode) {  if (index.search(property) == null)  index.insert(property, new BTree<>());  if (jsonNode.size() == 0) {  return;  }  for (JsonNode node : jsonNode) {  this.addToIndexed(property, node);  } } | |

This function will make an index on any property you want, it takes two parameters, the first one is the property the second is the data

|  |
| --- |
| public void addToIndexed(String property, JsonNode node) {  String nodeValue;  if (node.has(property)) {  nodeValue = node.get(property).asText();  BTree<String, ArrayList<JsonNode>> temp = index.search(property);  if (temp.search(nodeValue) == null) {  temp.insert(nodeValue, new ArrayList<>());  }  temp.search(nodeValue).add(node);  } } |

This function does as the it name suggests, it add to the already indexed property a new value

|  |
| --- |
| public void addToAllIndexed(JsonNode node) {  for (String property : getAllPropertyIndexed()) {  this.addToIndexed(property, node);  }  } |

This function add a value to all indexed properties

|  |
| --- |
| public ArrayList<JsonNode> get(String property, String exactValue) {  if (index.search(property) == null)  return null;  return new ArrayList<>(index.search(property).search(exactValue)); } |

This function finds a value from the indexing and return it

|  |
| --- |
| public void deleteFromIndexed(String key, JsonNode node) {  String value = node.get(key).asText();  BTree<String, ArrayList<JsonNode>> temp = index.search(key);  if (temp.search(value) != null) {  temp.search(value).remove(node);  } } |

This function deletes a value from the index

|  |
| --- |
| public void deleteSpecificKey(String property, String key) {  if (index.search(property) == null)  return;  index.search(property).delete(key); } |

This function deletes a key from the index.

|  |
| --- |
| public void deleteFromAllIndexed(JsonNode node) {  for (String property : this.getAllPropertyIndexed()) {  this.deleteFromIndexed(property, node);  } } |

This function deletes a value from all properties in the index

# Database class

This class represents a database in our Document database

It has two member variables

|  |
| --- |
| private String name; private final HashMap<String, JsonCollection> collections; |

The name of the database and a HashMap that store all the collections that’s belong to it

|  |
| --- |
| public Database() {  this.name = "default";  collections = new HashMap<>(); }  public Database(String name) {  this.name = name;  collections = new HashMap<>(); }  public void addCollection(String name) {  collections.put(name,  new JsonCollection(name)); }  public JsonCollection getCollection(String name) {  return collections.get(name); }  public void deleteCollection(String name) {  collections.remove(name); }  public String getDatabaseName() {  return name; }  public void setDatabaseName(String name) {  this.name = name; }  public HashMap<String, JsonCollection> getCollectionGroup() {  return collections; } |

The methods of this class is very clear and don’t need to be explained.

# JsonCollection class

This class represents a collection in our Document database

Same as the database it has two member variables

|  |
| --- |
| private String name; private HashMap<String, JsonDocument> documents; |

The name of the collection and the documents it holds.

And just like the Database class it methods is very clear and don’t need to be explained

|  |
| --- |
| public JsonCollection() {  this.name = "default";  documents = new HashMap<>(); }  public JsonCollection(String name) {  this.name = name;  documents = new HashMap<>(); }  public JsonDocument addDocument(JsonDocument document) {  documents.put(document.getDocumentName(), document);  return document; }  public JsonDocument getDocument(String documentName) {  return documents.get(documentName); }  public HashMap<String, JsonDocument> getDocuments() {  return documents; }  public String getName() {  return name; }  public void setName(String name) {  this.name = name; }  public void deleteDocument(String name) {  documents.remove(name); }  public void setDocuments(HashMap<String, JsonDocument> documents) {  this.documents = documents; }  @Override public String toString() {  return "JsonCollection{" +  "name='" + name + '\'' +  '}'; } |

# JsonDocument class

This class is like the core of our database it has 6 member values which are:

|  |
| --- |
| private final String documentName; private final ObjectMapper mapper; private final String id; private ArrayNode nodesArray; private final Indexer indexer; private int i = 0; |

1- The name of the document

2- An ObjectMapper from the Jackson library that used to map String to JSON objects and vice versa

3- The id of the document

4- ArrayNode which is an Array of JsonNodes

5- The indexer for this document

This class has two constructers

|  |
| --- |
| public JsonDocument(String documentName) {  this.documentName = documentName;  this.mapper = new ObjectMapper();  this.id = UUID.*randomUUID*().toString();  nodesArray = mapper.createArrayNode();  indexer = new BTreeIndexer(); }  public JsonDocument(String documentName, String id) {  this.documentName = documentName;  this.mapper = new ObjectMapper();  nodesArray = mapper.createArrayNode();  indexer = new BTreeIndexer();  this.id = id; } |

The first one is used when no ID is passed by the user, so it generates a random UUID

The second one is used when the user pass an ID.

### Insert And InsertMany

These two methods are responsible for inserting into the document

|  |
| --- |
| public void insert(String json) throws IOException {  if(json==null || json.equals("")) {  throw new IllegalArgumentException("Json is null or empty");  }  JsonNode node = mapper.readTree(wrapId(json));  nodesArray.add(node);  this.updateIndex(node); } public void insertMany(ArrayList<String> jsons) throws IOException {  if(jsons==null || jsons.isEmpty()) {  throw new IllegalArgumentException("Json is null or empty");  }  for (String s : jsons) {  insert(s);  } } |

First it checks if the json is null or empty then it read it into JsonNode then it adds to the nodesArray and update the index.

### updateOne and updateMany

these two methods are responsible for updating the document

|  |
| --- |
| public void updateOne(String property, String propertyValue, String newValue) throws IOException {  if(property==null || property.equals("") || propertyValue==null || propertyValue.equals("") || newValue==null || newValue.equals("")) {  throw new IllegalArgumentException("property, propertyValue or newValue is null or empty");  }  int i = 0;  for (JsonNode node : nodesArray) {  if (node.get(property).asText().equals(propertyValue)) {  JsonNode temp = mapper.readTree(wrapId(newValue));  nodesArray.set(i, temp);  this.removeFromIndexed(node);  this.updateIndex(temp);  break;  }  i++;  } } public void updateMany(String property, String propertyValue, String newValue) throws IOException {  if(property==null || property.equals("") || propertyValue==null || propertyValue.equals("") || newValue==null || newValue.equals("")) {  throw new IllegalArgumentException("property, propertyValue or newValue is null or empty");  }  for (JsonNode node : nodesArray) {  if (node.get(property).asText().equals(propertyValue)) {  updateOne(property, propertyValue, newValue);  }  }  } |

First it checks if one of the arguments is empty or null if yes it throws an Exception

Else it loops through the ArrayNodes and get the property then update it and then update the indexing.

The way that we update the json object is by creating a new Json Object with the new value and then setting it in the place of the old node.

### Delete And deleteMany

|  |
| --- |
| public void delete(String property, String propertyValue) throws IOException {  if(property==null || property.equals("") || propertyValue==null || propertyValue.equals("")) {  throw new IllegalArgumentException("property or propertyValue is null or empty");  }  int i = 0;  for (JsonNode nodes : nodesArray) {   if (nodes.get(property).asText().equals(propertyValue)) {  nodesArray.remove(i);  this.removeFromIndexed(nodes);  break;  }  i++;  } } |

For deleting a single document, it very straightforward we loop over all the nodes if a node matches the exact value, we want to delete we remove it and we break.

|  |
| --- |
| public void deleteMany(String property, String propertyValue) throws IOException {  if (property == null || property.equals("") || propertyValue == null || propertyValue.equals("")) {  throw new IllegalArgumentException("property or propertyValue is null or empty");  }   ArrayNode temp = mapper.createArrayNode();  for (JsonNode nodes : nodesArray) {  if (!nodes.get(property).asText().equals(propertyValue)) {  temp.add(nodes);  } else  this.removeFromIndexed(nodes);  }  this.nodesArray = temp; } |

Using the same approach mentioned for the deleteOne method will result in bugs with this method so my approach for it we create a temp ArrayNode and store all the nodes that’s don’t match the if statement and delete the others then assign the nodes array we have to the temp.

### Find

|  |
| --- |
| public ArrayList<JsonNode> find(String property, String propertyValue) {  if (property == null || property.equals("") || propertyValue == null || propertyValue.equals("")) {  throw new IllegalArgumentException("property or propertyValue is null or empty");  }  if (indexer.has(property)) {  return indexer.get(property, propertyValue);  } else {  return this.findForNonIndex(property, propertyValue);  } } |

We first check if we have an index on the property if yes we return it else we search in non-Indexed documents

|  |
| --- |
| public ArrayList<JsonNode> findForNonIndex(String property, String propertyValue) {  ArrayList<JsonNode> temp = new ArrayList<JsonNode>();   for (int i = 0; i < nodesArray.size(); i++) {  if (nodesArray.get(i).get(property).asText().equals(propertyValue)) {  temp.add(nodesArray.get(i));  }   }  return temp; } |

Which is done by looping over the nodes and reading it from the disk and comparing it to the propertyValue we want to find

# Services

Services or managers is used to manage the Database, Collection, Document, CRUD operations

## DatabaseService

This class manage the database it has on member variable which is a List of databases

|  |
| --- |
| public static Database addDatabase(Database database) {  try {  LocksManager.*getInstance*().getLock("DatabaseLock").writeLock().lock();  *DATABASE\_LIST*.add(database);  DirectoryCreator.*getInstance*().createDirectory(database.getDatabaseName());  return database;  }finally {  LocksManager.*getInstance*().getLock("DatabaseLock").writeLock().unlock();   }  } |

To add a database to our databases first we lock the method then we add it to the database list and create A Directory for it.

|  |
| --- |
| public static void deleteDatabase(String databaseName) {  try {  LocksManager.*getInstance*().getLock("DatabaseLock").writeLock().lock();  *DATABASE\_LIST*.removeIf(database -> database.getDatabaseName().equals(databaseName));  DirectoryRemover.*getInstance*().deleteDirectory(databaseName);  } finally {  LocksManager.*getInstance*().getLock("DatabaseLock").writeLock().unlock();   }   } |

Delete is the same as before but instead of adding we remove if the name of the database is equals to the name provided by the user, then we Delete its Directory

|  |
| --- |
| public static Optional<Database> getDatabase(String databaseName) {  try {  LocksManager.*getInstance*().getLock("DatabaseLock").writeLock().lock();  Optional<List<Database>> temp = *getDatabaseList*();  if (temp.isPresent()) {  return *getDatabaseList*().get().stream().filter(database -> database.getDatabaseName().equals(databaseName)).findFirst();  }  }  finally{  LocksManager.*getInstance*().getLock("DatabaseLock").writeLock().unlock();   }  return Optional.*empty*(); } |

To get a database we first load all databases then check if its present if its true we return the database that has a name equals to the name provided by the user

|  |
| --- |
| public static Optional<List<Database>> getDatabaseList() {  return Optional.*of*(*loadDatabases*()); } |

Load the whole database and return it as an Optional.

|  |
| --- |
| private static List<Database> loadDatabases() {  File mainFile = DirectoryCreator.*getInstance*().getMasterDir();  List<Database> databases = new ArrayList<>();  FileLoader.*loadDirectories*(mainFile).forEach(directory -> {  Database database = new Database(directory.getName());  databases.add(database);  });   for (int i = 0; i < databases.size(); i++) {  int j = i;  File temp = new File(mainFile.getPath() + "/" + databases.get(i).getDatabaseName());  FileLoader.*loadDirectories*(temp).forEach(  collection -> {  databases.get(j)  .getCollectionGroup()  .put(collection.getName(),  new JsonCollection(collection.getName())  );  }  );   for (JsonCollection collection : databases.get(i).  getCollectionGroup().  values()) {  AtomicReference<JsonDocument> tempDocument = new AtomicReference<>();  FileLoader.*loadFiles*(  new File(mainFile.getPath()  + "/"  + databases.get(i).getDatabaseName()  + "/"  + collection.getName())  ).forEach(  file1 -> {  ArrayNode nodes = new ObjectMapper().createArrayNode();  try {  JsonNode node = new ObjectMapper().readTree(file1);  if (!node.isEmpty()) {  nodes.add(node);  }  if (!nodes.isEmpty()) {  String json;  json = nodes.toString().replace("[", "").replace("]", "");  String json3 = "[" + json + "]";  nodes = new ObjectMapper().readValue(json3, ArrayNode.class);  }  } catch (IOException e) {  throw new RuntimeException(e);  }  tempDocument.set(databases.get(j)  .getCollection(collection.getName())  .addDocument(new JsonDocument(file1.getName())));   if (!nodes.isEmpty()) {  tempDocument.get().setNodesArray(nodes);  try {  File file =new File(databases.get(j).getDatabaseName() +"/" + collection.getName() + "/" + "indexer.out");  if(file.exists()) {  tempDocument.get().getIndexer().setIndexed(JsonWriter.*getInstance*().read("indexed.out"));  }  } catch (IOException | ClassNotFoundException e) {  throw new RuntimeException(e);  }   }  tempDocument.get().getIndexer().makeIndexOn("\_id", tempDocument.get().getNodesArray());  }   );    }  }  return databases; } |

This code is responsible for loading the data base //TODO EXPALIN IT

## CollectionService

It has two methods the addCollection and deleteCollection

|  |
| --- |
| public static Optional<JsonCollection> addCollection(String databaseName, JsonCollection collection) {  try {  LocksManager.*getInstance*().getLock("CollectionLock").writeLock().lock();   DatabaseService.*getDatabase*(databaseName).ifPresent(database -> database.addCollection(collection.getName()));  DirectoryCreator.*getInstance*()  .createDirectory(databaseName  + "/"  + collection.getName()  );  return Optional.*of*(collection);  }finally {  LocksManager.*getInstance*().getLock("CollectionLock").writeLock().unlock();  }   }   public static void deleteCollection(String databaseName, String collectionName) {  try {  LocksManager.*getInstance*().getLock("CollectionLock").writeLock().lock();   DatabaseService.*getDatabase*(databaseName).ifPresent(database -> database.deleteCollection(collectionName));  DirectoryRemover.*getInstance*()  .deleteDirectory(databaseName  + "/"  + collectionName  );  }  finally {  LocksManager.*getInstance*().getLock("CollectionLock").writeLock().unlock();  }  }  } |

First, we get the database then if it present, we either add or delete it.

## DocumentService

Just like before it has two methods adding and deleting

|  |
| --- |
| public static Optional<JsonDocument> addDocument(String databaseName, String collectionName, JsonDocument document) {  try {  LocksManager.*getInstance*().getLock("DocumentLock").writeLock().lock();  DatabaseService.*getDatabase*(databaseName).ifPresent(db -> db.getCollection(collectionName).addDocument(document));  DirectoryCreator.*getInstance*().createFile(  databaseName  + "/"  + collectionName,  document.getDocumentName()  );   return Optional.*of*(document);  } finally {  LocksManager.*getInstance*().getLock("DocumentLock").writeLock().unlock();   }  }   public static void deleteDocument(String databaseName, String collectionName, String documentName) {  try {  LocksManager.*getInstance*().getLock("DocumentLock").writeLock().lock();   DatabaseService.*getDatabase*(databaseName).ifPresent(db -> db.getCollection(collectionName).deleteDocument(documentName));  DirectoryCreator.*getInstance*().createDirectory(  databaseName  + "/"  + collectionName  + "/"  + documentName);  } finally {  LocksManager.*getInstance*().getLock("DocumentLock").writeLock().unlock();  }  }   } |

We get the database then the collection then we either add or delete the document based on the called function.

## CRUDService

1- Insert

|  |
| --- |
| public void insert(String json) {  try {  LocksManager.*getInstance*().getLock("CRUDLock").writeLock().lock();  currentDocument.insert(json);  JsonWriter.*getInstance*().write(currentDatabase.getDatabaseName() + "/" + currentCollection.getName() + "/" + currentDocument.getDocumentName(), currentDocument.getNodesArray());   } catch (IOException e) {  throw new RuntimeException(e);  } finally {  LocksManager.*getInstance*().getLock("CRUDLock").writeLock().unlock();  } } |

We lock the method then we call the insert method from the Document and commit changes to the file

2- insertMany

|  |
| --- |
| public void insertMany(ArrayList<String> jsons) {  try {  LocksManager.*getInstance*().getLock("CRUDLock").writeLock().lock();  currentDocument.insertMany(jsons);  JsonWriter.*getInstance*().write(currentDatabase.getDatabaseName() + "/" + currentCollection.getName() + "/" + currentDocument.getDocumentName() + ".json", currentDocument.getNodesArray());   } catch (IOException e) {  throw new RuntimeException(e);  } finally {  LocksManager.*getInstance*().getLock("CRUDLock").writeLock().unlock();  } } |

Nothing new same as before

3- updateOne

|  |
| --- |
| public void updateOne(String property, String propertyValue, String newValue) {  try {  LocksManager.*getInstance*().getLock("CRUDLock").writeLock().lock();  currentDocument.updateOne(property, propertyValue, newValue);  JsonWriter.*getInstance*().write(currentDatabase.getDatabaseName() + "/" + currentCollection.getName() + "/" + currentDocument.getDocumentName(), currentDocument.getNodesArray());   } catch (IOException e) {  throw new RuntimeException(e);  } finally {  LocksManager.*getInstance*().getLock("CRUDLock").writeLock().unlock();  } } |

Also, nothing changed same as before

4- updateMany

|  |
| --- |
| public void updateMany(String property, String propertyValue, String newValue) {  try {  LocksManager.*getInstance*().getLock("CRUDLock").writeLock().lock();  currentDocument.updateMany(property, propertyValue, newValue);  JsonWriter.*getInstance*().write(currentDatabase.getDatabaseName() + "/" + currentCollection.getName() + "/" + currentDocument.getDocumentName() + ".json", currentDocument.getNodesArray());   } catch (IOException e) {  throw new RuntimeException(e);  } finally {  LocksManager.*getInstance*().getLock("CRUDLock").writeLock().unlock();  } } |

Nothing changed.

5- delete and delteMany

|  |
| --- |
| public void deleteOne(String property, String propertyValue) {  try {  LocksManager.*getInstance*().getLock("CRUDLock").writeLock().lock();  currentDocument.delete(property, propertyValue);  DirectoryRemover.*getInstance*().deleteDirectory(currentDatabase.getDatabaseName() + "/" + currentCollection.getName() + "/" + currentDocument.getDocumentName() + ".json");  } catch (IOException e) {  throw new RuntimeException(e);  } finally {  LocksManager.*getInstance*().getLock("CRUDLock").writeLock().unlock();  } }  public void deleteMany(String property, String propertyValue) {  try {  LocksManager.*getInstance*().getLock("CRUDLock").writeLock().lock();  currentDocument.deleteMany(property, propertyValue);  DirectoryRemover.*getInstance*().deleteDirectory(currentDatabase.getDatabaseName() + "/" + currentCollection.getName() + "/" + currentDocument.getDocumentName() + ".json");  } catch (IOException e) {  throw new RuntimeException(e);  } finally {  LocksManager.*getInstance*().getLock("CRUDLock").writeLock().unlock();  }  } |

The only difference here that we delete the file instead of writing to it.

6-find

|  |
| --- |
| public ArrayList<JsonNode> find(String property, String propertyValue) {  return currentDocument.find(property, propertyValue); } |

We call the find method from the document

7- makeIndexOn

|  |
| --- |
| public void makeIndexOn(String property) {  try {  LocksManager.*getInstance*().getLock("CRUDLock").writeLock().lock();  currentDocument.makeIndexOn(property);  } finally {  LocksManager.*getInstance*().getLock("CRUDLock").writeLock().unlock();   } } |

# IO

These classes are responsible for IO communications

## DirectoryCreator

A singleton class that responsible for creating Directories or files

It holds one member variable which is

|  |
| --- |
| private static final File *MASTER\_DIR* = new File("databases"); |

|  |
| --- |
| public static DirectoryCreator getInstance() {  return DirectoryCreatorHolder.*INSTANCE*; } |

It returns an instance of the class

|  |
| --- |
| public void createDirectory(String path) {  File dir = new File(*MASTER\_DIR* + "/" + path);  if (!dir.exists()) {  try {  FileUtils.*forceMkdir*(dir);  } catch (IOException e) {  throw new RuntimeException(e);  }  }  } |

Create a directory after checking if the directory dose not exists before. It used FileUtils class from apache commons.

|  |
| --- |
| public File createFile(String path, String fileName) {  File dir = new File(*MASTER\_DIR* + "/" + path);   this.createDirectory(dir.getPath());   File file = new File(dir, fileName);  if (file.exists()) {  return file;  }  try {  file.createNewFile();  } catch (IOException e) {  throw new RuntimeException(e);  }  return file; } |

Same is before but now it creates a file

|  |
| --- |
| private static class DirectoryCreatorHolder {  private static final DirectoryCreator *INSTANCE* = new DirectoryCreator(); } |

This is a Holder for the Singleton which ensure Thread safety will talk about this in the multithreading section

## DirectoryRemover

|  |
| --- |
| private static final File *MASTER\_DIR* = new File("databases");  private DirectoryRemover() { }  public static DirectoryRemover getInstance() {  return DirectoryRemoverHolder.*INSTANCE*;  }  public void deleteDirectory(String directoryPath) {  try {  FileUtils.*forceDelete*(new File(*MASTER\_DIR* + "/" + directoryPath));  } catch (IOException e) {  throw new RuntimeException(e);  }  }  private static class DirectoryRemoverHolder {   private static final DirectoryRemover *INSTANCE* = new DirectoryRemover(); } |

Same as before but here we delete the Directory using forceDelete method

## FileLoader

This class is responsible for loading directories and files from the disk to File objects in java

|  |
| --- |
| public class FileLoader {  private FileLoader() { }   public static FileLoader getInstance() {  return FileLoaderHolder.*INSTANCE*;  }   public List<File> loadDirectories(File parent) {  File[] files = Objects.*requireNonNull*(parent.listFiles());  List<File> directories = new ArrayList<>();   for (File file : files) {  if (file.isDirectory()) {  directories.add(file);  }  }  return directories;   }   public List<File> loadFiles(File parent) {  File[] files = Objects.*requireNonNull*(parent.listFiles());  List<File> files1 = new ArrayList<>();   for (File file : files) {  if (file.isFile() && file.getName().endsWith(".json")) {  files1.add(file);  }  }  return files1;  }  private static class FileLoaderHolder {  private static final FileLoader *INSTANCE* = new FileLoader();  }  } |

In the loadDirectories methods we get all the files from the parent directory and and loop over them and if the file is a directory we add it the list after the loop we return

The loadFiles method is the same

## FileWriterReader

This class used to write file to the disk and read it from it.

|  |
| --- |
| public class FileWriterReader {  private final ObjectMapper mapper = new ObjectMapper();  private static final File *MASTER\_DIR* = new File("databases");    private FileWriterReader() {  }   public static FileWriterReader getInstance() {  return JsonWriterHolder.*INSTANCE*;  }   public void writeJson(String filePath, ArrayNode content) throws IOException {  File file = new File(*MASTER\_DIR* + "/" + filePath);  mapper.writerWithDefaultPrettyPrinter().writeValue(Files.*newOutputStream*(file.toPath()), content);  }    public <T extends Serializable> void write(T object, String pathTo) throws IOException {  ObjectOutputStream objectOutputStream = new ObjectOutputStream(Files.*newOutputStream*(Paths.*get*(pathTo)));  objectOutputStream.writeObject(object);  objectOutputStream.close();  }   public <T extends Serializable> T read(String pathTo) throws IOException, ClassNotFoundException {  ObjectInputStream objectInputStream = new ObjectInputStream(Files.*newInputStream*(Paths.*get*(pathTo)));  T object = (T) objectInputStream.readObject();  objectInputStream.close();  return object;  }   private static class JsonWriterHolder {  private static final FileWriterReader *INSTANCE* = new FileWriterReader();  } } |

The writeJson method writes to the document using the mapper Object

The write method is used for serialization an object

And the rea metho is used for de-serialization an object.

## ZipManager

Sending nested Directories and files using sockets is difficult, so I decided to zip them before sending, and unzip it after receiving, this way I will just send a one file and receive one file

|  |
| --- |
| public class ZipManager {  private ZipManager() {  }   public static ZipManager getInstance() {  return ZipManagerHolder.*INSTANCE*;  }  public void unZip(Path source, Path destination) throws IOException{  new ZipFile(source.toFile()).extractAll(destination.toString());  }  public void zip(String zipName, String filePath) throws ZipException {  new ZipFile(zipName).addFolder(new File(filePath));  }  private static class ZipManagerHolder {  private static final ZipManager *INSTANCE* = new ZipManager();  }    } |

This class used zip4j as a library for zipping its very useful library and easy to use and understand.

## SocketFileManager

This class is responsible for sending and receiving files through the sockets

|  |
| --- |
| public class SocketFileManager {   private final DataOutputStream dataOutputStream;  private final DataInputStream dis;   private SocketFileManager(Socket socket) throws IOException {  this.dataOutputStream = new DataOutputStream(socket.getOutputStream());  dis = new DataInputStream(socket.getInputStream());  }   public static SocketFileManager create(Socket socket) {  try {  return new SocketFileManager(socket);  } catch (IOException e) {  throw new RuntimeException("can't create a new File manager with this socket");  }  }    public void sendFile(String filePath) throws IOException {  try {  LocksManager.*getInstance*().getLock("sendFile").writeLock().lock();   ZipManager.*getInstance*().zip("databasesToBeSend.zip", filePath);   int bytes = 0;  File fileToBeSent = new File("databasesToBeSend.zip");   FileInputStream fileInputStream = new FileInputStream(fileToBeSent);   dataOutputStream.writeLong(fileToBeSent.length());   byte[] buffer = new byte[4 \* 1024];   while ((bytes = fileInputStream.read(buffer)) != -1) {  dataOutputStream.write(buffer, 0, bytes);  dataOutputStream.flush();  }  fileInputStream.close();  FileUtils.*delete*(fileToBeSent);  }finally {  LocksManager.*getInstance*().getLock("sendFile").writeLock().unlock();  }  }   public void receiveFile(String fileName,String savePath) throws IOException {  try {  LocksManager.*getInstance*().getLock("sendFile").writeLock().lock();  FileOutputStream fos = new FileOutputStream(fileName + ".zip");  int bytesRead;  long size = dis.readLong();  byte[] buffer = new byte[4 \* 1024];   while (size > 0 && (bytesRead = dis.read(buffer, 0, (int) Math.*min*(buffer.length, size))) != -1) {  fos.write(buffer, 0, bytesRead);  size -= bytesRead;  }  fos.close();  ZipManager.*getInstance*().unZip(new File(fileName + ".zip").toPath(), new File(savePath).toPath());  FileUtils.*delete*(new File(fileName + ".zip"));   }finally {  LocksManager.*getInstance*().getLock("sendFile").writeLock().unlock();  }  }  } |

It uses the famous technique to send a file by splitting it up to a small chunk and sending it

Same as receive it receive small chunks of the file then group it.

# Nodes

## NodeServer

This class is used to send the databases to all connect nodes it uses a subscriber-publisher pattern to do this.

First at every connection it saves the client socket to a ConcurrentLockedQueue

Then it calls function broadcast which is used to send the databases to the reading node

After that it will keep waiting for new connections or database updated when a database update happens it will call the function sendDatabase and send the updated database to the reading node.

|  |
| --- |
| private final Queue<Socket> connections; private PrintWriter writer;  public NodeServer() {  connections = new ConcurrentLinkedQueue<>(); }  public void connect(ServerSocket serverSocket) throws IOException, InterruptedException {  Socket socket = serverSocket.accept();  this.subscribe(socket);  this.broadcast(socket); }    public void subscribe(Socket node) {  connections.add(node); }  public void unSubscribe(Socket node) {  connections.remove(node); }  public synchronized void broadcast(Socket node) {  SocketFileManager socketFileManager =SocketFileManager.*create*(node);  try {  writer = new PrintWriter(node.getOutputStream());  writer.println("sending databases");  writer.flush();  socketFileManager.sendFile("databases");  } catch (IOException e) {  throw new RuntimeException(e);  } } public synchronized void sendDatabase(String databaseName) {  for(Socket node : connections) {  SocketFileManager fileManager = SocketFileManager.*create*(node);  try {  writer = new PrintWriter(node.getOutputStream());  writer.println(databaseName);  writer.flush();  fileManager.sendFile("databases/"+databaseName);  } catch (IOException e) {  throw new RuntimeException(e);  }  } }  public void runServer() throws IOException {  ServerSocket masterNode = new ServerSocket(9097);   Thread t1 = new Thread(() -> {  try {  while (true) {  connect(masterNode);  }  } catch (IOException | InterruptedException e) {  throw new RuntimeException(e);  }  });   t1.start(); } |

## PortGenerator

This class is very simple it generate a new Port (int) every time when we call it method.

|  |
| --- |
| public class PortGenerator {  private int port;  private PortGenerator() {  this.port = 8099;  }    public static PortGenerator getInstance() {  return PortGeneratorHolder.*INSTANCE*;  }  public int generateNewPort() {  return ++port;  }   public int getPort() {  return port;  }  private static class PortGeneratorHolder {  private static final PortGenerator *INSTANCE* = new PortGenerator();  } } |

## ReadingNode

This class represents a node from the cluster it’s not the actual implementation it only has the jar file path and a way to run it

|  |
| --- |
| public class ReadingNode implements Runnable {   private static String *nodeFilePath* = "ReadingNode-1.0-SNAPSHOT.jar";  private final int port;  private final String NODE\_RUN = "java -jar "  + *nodeFilePath* + " "  + PortGenerator.*getInstance*().getPort();   public ReadingNode() {  this.port = PortGenerator.*getInstance*().generateNewPort();  }   public void killNode() {  String killCommand ="wmic Path win32\_process Where \"CommandLine Like '% "ReadingNode-1.0-SNAPSHOT  .jar%'\" Call Terminate";  ShellManager shellManager = ShellManager.*create*(killCommand);  shellManager.start();  }   @Override  public void run() {  ShellManager shellManager = ShellManager.*create*(NODE\_RUN);  shellManager.start();  }   public int getPort() {  return port;  } } |

So, how we run it?

We Use a ShellManager class to execute the command! Very simple and straightforward

## ShellManager

This class uses the ProcessBuilder class provided by java and execute a shell command using it

|  |
| --- |
| public class ShellManager{  private final String command;    private ShellManager(String command) {  this.command = command;  }   public static ShellManager create(String command) {  return new ShellManager(command);  }   private static void runShellCommand(String command) {  try {  ProcessBuilder builder = new ProcessBuilder("cmd.exe","/c",command);  builder.start();  builder.redirectOutput(new File("log.txt"));   } catch (IOException e) {  throw new IllegalArgumentException("the command is not correct");  }  }  public void run() {  *runShellCommand*(command);  } } |

# Clean Code Discipline

Chapter 2 (Meaningful Names):

Here the book talk about an important part of clean coding which is chose the best name as you

can, in programming everything you create you had to name it like the variables, classes,

methods even packages.

For example, of **Using Intention-Revealing Names**, the method naming in the

CollectionService class e.g., addCollection or deleteCollection it reveals my intent of

making the method and what the method is doing.

**Avoid Disinformation**, sometimes naming variable of object with the name of class like

listOfIndexes for map of indexes, but that I avoided it and name it indexes straight forward.

My project code applies nearly everything on this chapter, all of them are **Pronounceable Names**

**and Searchable, no cute stuff.**

It uses a **Solution Domain Names** like CollectionService or LoadBalancer classes.

Chapter 3 (Functions):

Here the rules become harder to achieve and my code applied as much as possible.

Let’s start with the **Small** the author says “The first rule of functions is that they should be small. The

second rule of functions is that they should be smaller than that.”.

Most of my function are around 4-5 lines long and that very good, but there some violation in my code like the MainServer class or the client class or DatabaseServic.

**Do One Thing**, most of my function do only one thing like deleteDatabase this function just delete the database! I have some violation in my code for this but its very little.

**Function arguments**, the author says “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.”

Most of my function takes one to two arguments when I needed more, I used the builder pattern

**Blocks and Indenting**, most of indenting is done by the ide.

**Don't Repeat Yourself**, the best application of this rule is the utilities classes, these classes are

made to have a global method that used all over the project.

Chapter 4 (Comments):

* Didn’t used comments on my project because I used **Explain Yourself in code**

Chapter 5 (Formatting)

IDE do these things for us nowadays

Chapter 6 (Objects and Data Structures)

This chapter talk about the importance of design patterns and SOILD principles

**Data Abstraction**, you can say I violate this rule, my code depends on getters and setters a lot

**The Law of Demeter**, this law is applied by making a Managers and Services, it

means that the client doesn’t need to see all manipulations to get the purpose of certain method.

**Data Transfer Objects**, there is no DTO objects in my code

Chapter 7 (Error Handling):

**Use Exceptions Rather Than Return Codes**, I never used return codes in my code

**Provide Context with Exceptions**, I printed a message to the user explaining why the exceptions happened.

**Don't Return Null**, this rule is partially applied, using the java Optional I managed to write

some methods to handle this rule.

**Don't Pass Null**, never passed null

Chapter 8 is not about clean coding so I will skip it in this section,

Chapter 9 is big subject, and I will talk about it later in a single section.

Chapter 10 (Classes):

**Encapsulation**, this is a solid rule that we applied it first time we learn about OOP, it makes the

code more robust by preventing the change of the class’s fields on wrong ways.

**Classes Should be Small**, my classes is very small I think the average length is about 100-110

**The Single Responsibility Principle**, this principle will be discussed later in the SOLID section.

**Isolating from Change**, I tried to make the access modifiers to the only used cases, same thing

for immutability and defining the class fields as final

# SOLID Principles

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

## SRP

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

In my code most classes have only job to do for example the CollectionService class has on job to do which is creating and deleting Collections

## Ocp

Open closed principle is “A software artifact should be open for extension but closed for modification”

In my code you can see this like for example if we want to add a new Indexer, we simply implement the interface “Indexer” we don’t need to change the BTreeIndexer.

## 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.”

I don’t think I have a use case for this principle in my code.

## DIP

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

My high-level classes depend on A Services not the low level details so I think I did well for this principle

# Effective Java

**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.”

I applied this item in the SocketFileManager class

|  |
| --- |
| public static SocketFileManager create(Socket socket) {  try {  return new SocketFileManager(socket);  } catch (IOException e) {  throw new RuntimeException("can't create a new File manager with this socket");  } } |

**Item 2: Consider a builder when faced with many constructors’ parameters**

“Static factories and constructors share a limitation: they do not scale well to large

numbers of optional parameters.”

I used this item in my CRUD service class

|  |
| --- |
| public static class CRUDServiceBuilder {  private Database currentDatabase;  private JsonCollection currentCollection;  private JsonDocument currentDocument; |

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

this item is used a lot in my code almost every IO class is a singleton, and I used a private constructor method to enforce it

**Item 4: Enforce noninstantiability with a private constructor**

I applied this principle in the PortGenerator Class which has private constructor and public static methods

**Item 5: Prefer dependency injection to hardwiring resources**

Can’t apply this in my code since it needs a framework like spring to do this for me

**Item 6: Avoid creating unnecessary objects**

in my code every objects that is created is being used

**Item 8: Avoid finalizers and cleaners**

Brian Goetz in Java Concurrency in Practice strongly recommends the following technique for ensuring that a lock is released. It uses a try..finally style; this ensures that no matter what happens during the time the lock is held, it will eventually be unlocked. Even if an unexpected RuntimeException is thrown, the lock will still be released.

So, I used finally for uncloaking locks

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

I used this a lot in my code also I volatile it a lot

**Item 12: Always override toString**

Almost every class in my code override toString

**Item 14: Consider implementing Comparable**

The only class I needed this for is the BTree class

**Item 15: Minimize the accessibility of classes and members**

Almost every member in my classes is private

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

All my classes use this.

**Item 17: Minimize mutability**

Every filed in my classes that can be final is final

**Item 18: Favor composition over inheritance**

I used this item in the Data classes like the document and Collection

**Item 20: Prefer interfaces to abstract classes**

I don’t have any abstract classes in my code

**Item 24: Favor static member classes over nonstatic**

Every member class in my code is static

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

Every class has it own file

**Item 26: Don’t use raw types**

I used the generic types always never used a raw type

**Item 28: Prefer lists to arrays**

Never used arrays in my code, I used ArrayList

**Item 30: Favor generic methods**

I applied this principle in the serialization method in the FileWriterReader

|  |
| --- |
| public <T extends Serializable> void write(T object, String pathTo) throws IOException {  ObjectOutputStream objectOutputStream = new ObjectOutputStream(Files.*newOutputStream*(Paths.*get*(pathTo)));  objectOutputStream.writeObject(object);  objectOutputStream.close(); }  public <T extends Serializable> T read(String pathTo) throws IOException, ClassNotFoundException {  ObjectInputStream objectInputStream = new ObjectInputStream(Files.*newInputStream*(Paths.*get*(pathTo)));  T object = (T) objectInputStream.readObject();  objectInputStream.close();  return object; } |

**Item 34: Use enums instead of int constants**

Did not use this item as I don’t need any int constants

**Item 40: Consistently use the Override annotation**

Every time I override a metho I annotate it with the Override annotation

**Item 42: Prefer lambdas to anonymous classes**

I used lambdas instead of anonymous classes, IntelIJ suggested you to change the anonymous classes to lambdas if possible

Item 43: Prefer method references to lambdas

Did not use this item very well

**Item 49: Check parameters for validity**

I applied this item in my project for example if the user wants to insert a null json to the data base it will throw an IlleagelArgumentExecption

**Item 51: Design method signatures carefully**

This item is like the naming chapter in the clean code principles book

**Item 52: Use overloading judiciously**

Never used overloading in my project

**Item 57: Minimize the scope of local variables**

Every variable in my project is in the smallest scope it can be in.

**Item 58: Prefer for-each loops to traditional for loops**

I used for-each loops a lot in my project an I think I used the traditional loops just three time or four

**Item 59: Know and use the libraries**

I used many libraries like zip4j Jackson

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

Did not used floats and double in my code

**Item 64: Refer to objects by their interfaces**

I applied this item nearly to every class that has an interface in my code for example

|  |
| --- |
|  |

**Item 69: Use exceptions only for exceptional conditions**

I applied this item in my project; for example, in JsonDocument class, if the user is trying to insert null json or empty string (and this unchecked exception), the system will throw it if it happens.

**Item 72: Favor the use of standard exceptions**

All the exceptions used in my project are standard exceptions.

**Item 77: Don’t ignore exceptions.**

Never ignored any exception in my project.

**Item 78: Synchronize access to shared mutable data**

Did not use synchronized but I used ReadWriteLocks.

**Item 79: Avoid excessive synchronization**

Same as the last point

**Item 84: Don’t depend on the thread scheduler**

As mentioned earlier I used try-finally to unlock the methods.

# MultiThreading

## Thread Safe Singletons

To ensure that a Singleton class is thread safe there is a lot of methods like locks synchronized but the best method I found when I was searching is William -Bill- Pugh singleton which is like this

|  |
| --- |
| class JsonWriter {  private JsonWriter() {}  /\* ... other methods ... \*/  public static JsonWriter getInstance() {  return SingletonHelper.INSTANCE;  }  private static class SingletonHelper {  private static final JsonWriter INSTANCE = new JsonWriter();  }  } |

This implementation ensures multi-thread safety and ensure best performances as it avoids eager creation. So how?

the static member INSTANCE, initialized at class level, will be created only when the nested class is loaded by the class loader, i.e., when the nested class will be used. In the following implementation, this can only happen when the getInstance() method is invoked. In fact, conversely from the EagerSingleton, this model allows using the enclosing class without causing an eager instantiation. This means that any other method offered by the enclosing class can be safely used without initializing INSTANCE; only the getInstance() method will cause it.

// TODOS

FINSHIN THE CODING REPORT

TESTING

DESGIN PATTERNS

AND MULTI-THREADING

Reference

1. <https://www.geeksforgeeks.org/mongodb-database-collection-and-document/>
2. <https://aws.amazon.com/nosql/document/>
3. <https://www.mongodb.com/document-databases>