NoSQL Database

Atypon Training: Final Project

Khalid Nusserat

Introduction

What is the database that the application implements?

The application implements a `NoSQL`, *document-based* database. This means that the application allows users to store entries as *documents*, which in the context of this application will be represented as `JSON` objects.

Unlike traditional `SQL` databases, `NoSQL` databases usually *do not* store or keep track of relationships between documents, and rather leaves this task to the programming language.

Why are NoSQL databases useful?

Since `NoSQL` do not need to keep track of relationships between entries, they can achieve a very high speed on writes, since there is no need to check complex constraints on every write. Moreover, `NoSQL` databases are much easier to *horizontally scale* than traditional `SQL` databases.

ACID properties

ACID properties are a set of properties that a database should have to ensure that the data is valid despite errors. The ACID properties are:

- Atomicity: Transactions are treated as one unit, where either all of the transaction finishes, or none at all.
- Consistency: All transactions must take the database from one valid state to an another valid state.
- Isolation: Transactions should not effect each other.
- Durability: Written data should not be lost.

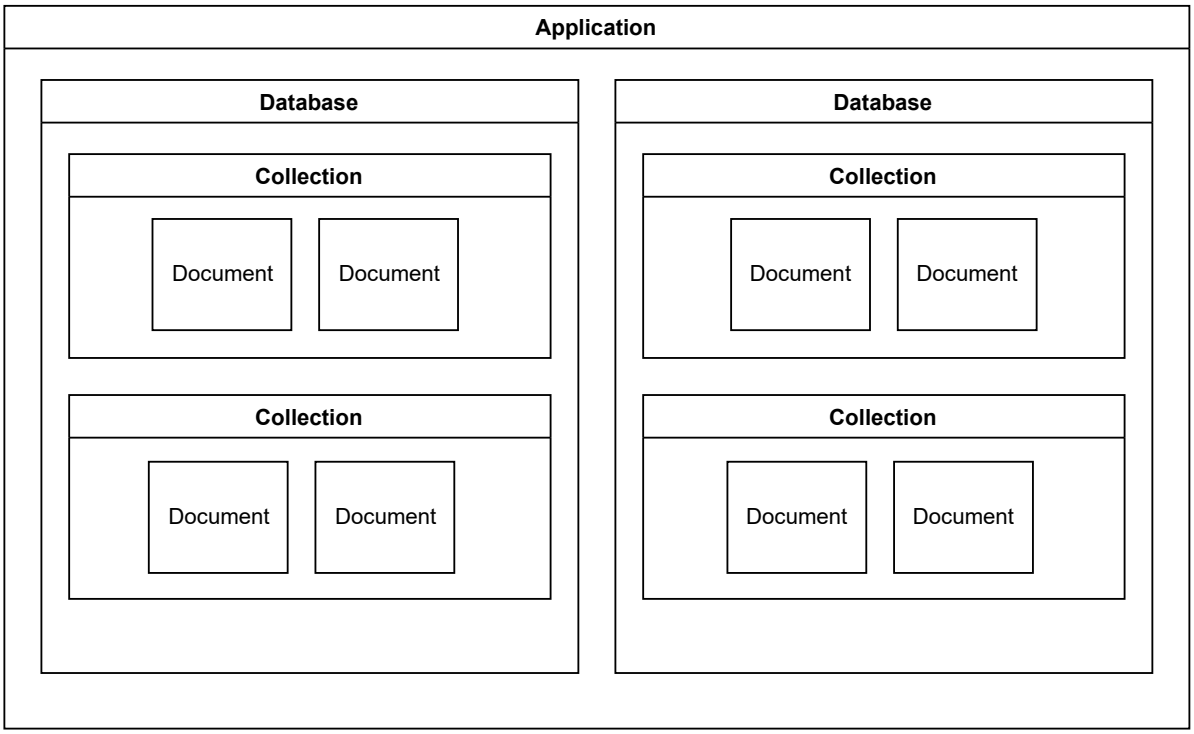
The application ensures the atomicity and consistency of a single document, however this does not hold for multiple documents.

Main features of the application

- Allows users to define databases and collections to group the documents into logical groups.
- Allows querying for documents based on any property.
- Allows users to create indexes on any property or properties to improve querying performance.

- Utilizes multi-threading to improve performance.
- Allows the creation of expressive and complex schemas.
- Allows updating documents by only providing the changes, rather than having to provide the entire updated document every time.
- Utilizes caches to improve I/O speed.
- Protects documents from outside modification by verifying each document when reading it.
- Provides authorization to protect resources using authorities.
- Allows the creation of user-defined roles that encapsulate a number of authorities.
- Allows the creation of replica nodes, to which all read operations will be sent to, in order to lessen the load on the primary node.

Overview of the design



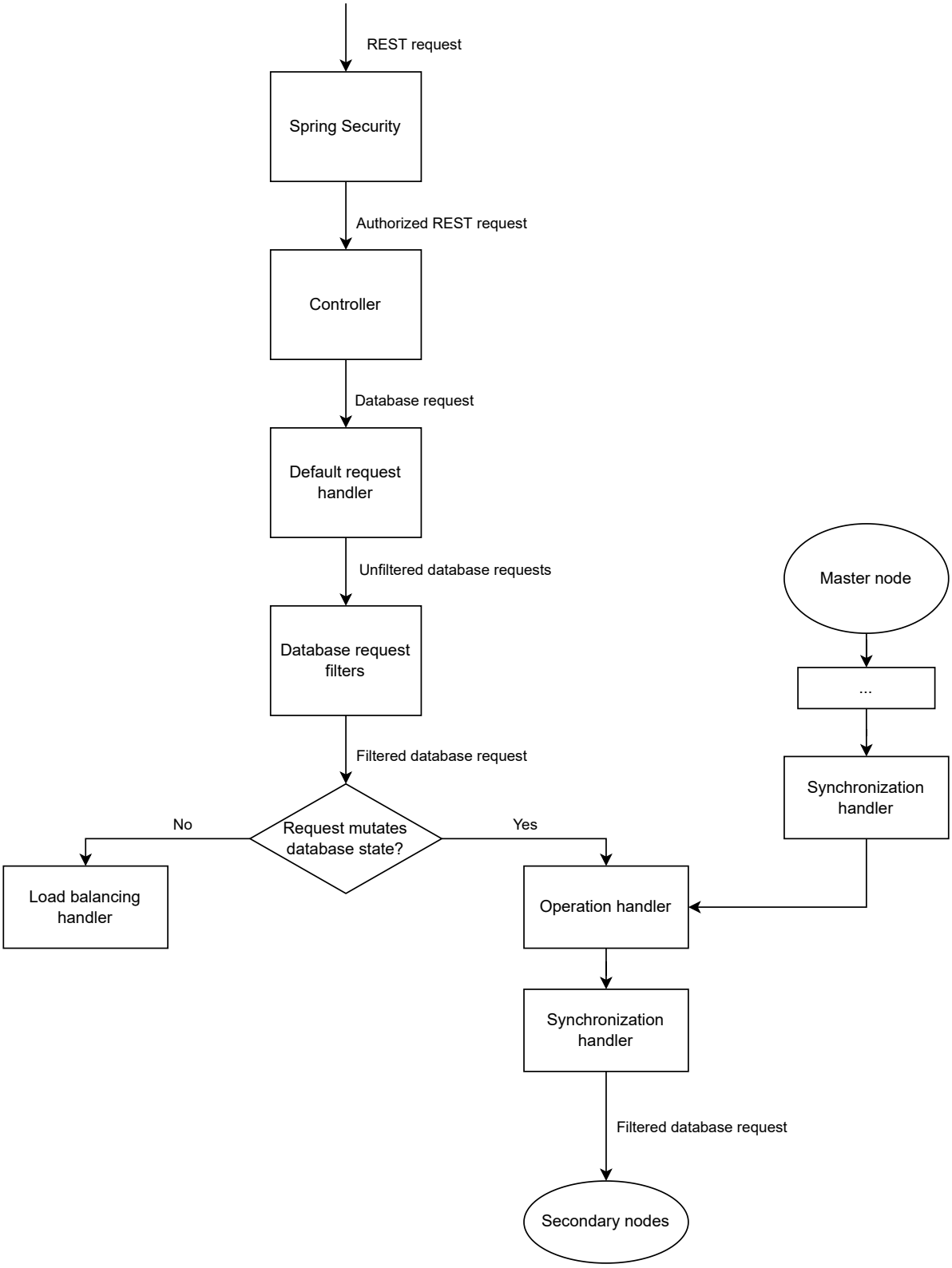
As the above diagram shows, the documents - *which represents the data that the user wants to store* - are stored in *collections*, and collections in turn are stored in a *database*, and the system contains a number of databases.

In addition to storing users' databases, the system also stores a database of its own, called the *metadata database*, which as the name suggests stores information about the database. It contains two collections, namely:

- The users collection: Stores the credentials and the roles of the system users.
- The roles collection: Stores the authorities granted to each role, for example the `ROOT_ADMIN` role has all authorities. Users can of course create their own custom roles.

It is important to note that all objects, including objects like indexes, users data and roles description and whatsoever, are all stored as ***JSON documents***. There is a great benefit to having all objects in the application be stored in the exact same way, which is that the code used to store documents can now also be used to store other objects of interest, such as indexes and users

data.



All of the logic concerning the database and the storage and retrieval of documents is completely decoupled and separated from the logic concerning handling requests, as the diagram above shows.

The requests are sent to the application as `REST` requests. First, Spring authenticate and authorizes requests using Spring Security. Then, using Spring's `RestController`, each endpoint is handled. Each endpoint wraps the entire request in a `DatabaseRequest`, which contains all the required information to process and handle the request, such as the `operation` type, the name

of the `database`. The request is then forwarded to the default `DatabaseRequestHandler`. The handlers accept instances of `DatabaseRequest`, and then returns an instance of `DatabaseResponse`, which contains result of the request that is to be sent back to the user.

The default handler first passes the request through a number of filters. Each filter accepts a `DatabaseRequest` and returns `DatabaseRequest` after modifying it according to its own set of rules. The filtered request is then passed to the `DefaultOperationsHandler`, which then in turn forwards the request to a specific operation handler that is responsible for handling one and only one database operation, for example there can be a handler that handles the operation `ADD_DOCUMENT`. Upon receiving the response from the operation handler, the `DefaultOperationsHandler` returns the response back to the default handler, which then returns the response back to the controller and then to the user.

In depth look at the design

Documents

Documents are the main building block of the application, they represent the data the user wants to store, and they are represented as `JSON` objects.

Documents are represented in the system as instances of the class `Document`, which provides convenient and necessary methods for handling documents. The `Document` class provides a number of static factory methods, such as `Document.fromJson(String json)` and `Document.of(Object... elements)`.

> ***IMPORTANT***: All `Document` objects are expected to be *immutable*, making them thread-safe.

The methods that `Document` objects provide are:

- `subsetOf(Document otherDocument)`: Returns true if all of the values and fields pairs in the `Document` object also exists in `otherDocument`, for example, the first document is a subset of the second document:

```
{
    "field1": "value1",
    "field2": {
        "field3": "value3"
    }
}
```

```
{
    "field1": "value1",
    "field2": {
        "field3": "value3",
        "field4": "value4"
    },
    "field5": "value5"
}
```

- `getValues(Document fields)`: Returns a subset of the `Document` object according to the provided fields argument. Document fields are represented as regular documents, except that the value of every field is *zero*. Take for example the following document:

  ```
  {
      "name": "John Doe",
      "age": 42,
      "hobbies": ["reading"].
      "university": {
          "name": "Univeristy"
      }
  }
  ```

  The fields of such would be represented as:

  ```
  {
      "name": null,
      "age": null,
      "hobbies": null.
      "university": {
          "name": null
      }
  }
  ```

  And if for example `getValues` was called on the above document with the following fields as an argument

  ```
  {
      "name": null,
      "university": {
          "name": null
      }
  }
  ```

  The result of the call would be:

  ```
  {
      "name": "John Doe",
      "university": {
          "name": "University"
      }
  }
  ```

- `getFields()`: Returns the fields document - *as described previously* - of the `Document` object.
- `overrideFields(Document newFieldsValues)`: Returns a copy of the current `Document` object, except that some fields values are replaced with the corresponding values from the `newFieldsValues` argument. For example calling the function on the following document:

```
{
    "name": "John Doe",
    "age": 42,
    "hobbies": ["reading"].
    "university": {
        "name": "Univeristy"
    }
}
```

With the following document as the `newFieldsValue` argument:

```
{
    "age": 43
    "university": {
        "name": "New Univeristy"
    }
}
```

Would return the following `Document`:

```
{
    "name": "John Doe",
    "age": 43,
    "hobbies": ["reading"].
    "university": {
        "name": "New Univeristy"
    }
}
```

- `toMap()`: Returns the document as a `Map<String, Object>`.
- `toObject(Class<T> classOfObject)`: Returns the document as an instance of `T`.
- `withId()`: Returns a copy of the document with an additional `_id` field that is calculated using `IdGenerator`. The default `IdGenerator` in the application is `Sha256IdGenerator`, which returns an id by computing `sha256(object.toString() + currentTime)`, where the current time is used to guarantee that even documents with identical content would have different ids.

> ***IMPORTANT:*** The `Document` class uses `JsonElement` as the building block for its implementation. `JsonElement` can be either a `JsonPrimitive`, in which case it contains a primitive such as `String` or `Number`, or it could be `JsonArray`, in which case it stores an array of `JsonElement` instances, or it could be `JsonNull`, which represents a null value, or it could be a `JsonObject`, which represents a `JSON` object that maps fields to `JsonElement`.

Schema

Each documents collection has one document schema, which describes the structure that all documents in the collection must follow, and any document that violates the schema is to be rejected.

Schemas are stored as documents, with the fields describing the field's name, and the values are *type descriptors*, which can be one of the following:

- A string that describes the type, such as: `"string", "number", "boolean"`, which are currently the supported types in the application. This type descriptor describes a primitive.
- A list that contains a single type descriptor, which describes an array of elements of that type descriptor.
- A schema.

Moreover, fields have the following rules:

- A field that ends with `?` is optional, for example the field `spouse?` is optional.
- A field that ends with `!` cannot be `null`, for example the field `_id!` cannot be null.

An example of schema would be:

```
{
    "name!": "string",
    "age!": "number",
    "hobbies!": ["string"].
    "university?": {
        "name!": "string"
    },
    "spouse": "string"
}
```

The fields `name, age` and `hobbies` are all required and cannot be null, the field `spose` is required but nullable, and finally the field `university` is optional, but if it exists then it describes a `JSON` object that have the required, non-nullable field `name`.

Constraints

The `DocumentSchema` class validates `Document` objects by using *constraints*, which are represented by instances of the `Constraint` interface. The `Constraint` interface describes an object that taken in a `JsonElement` and returns `true` if the element meets the constraint, otherwise it returns `false`. The implementations of `Constraint` are:

- `Constraints`: Contains a number of `Constraint` objects, and returns `true` if the element meets the requirement *all* of them.
- `ArrayElementConstraints`: Contains a `Constraint`, and expects to take in a `JsonArray` as an argument, then it returns `true` if *all* of the array elements meet the constraint.
- `TypeConstraint`: Contains a type, and returns true if the type of the `JsonElement` matches it. For example, a specific type constraint might only return `true` for `JsonElement` that represents `JsonPrimitive` that contains `Number` objects.
- `FieldConstraint`: Contains a field name and information about the field, such as whether it is optional or nullable, and a `Constraint` that applies to the field's value. Then, it expects to receive a `JsonObject` for which it will check the field.

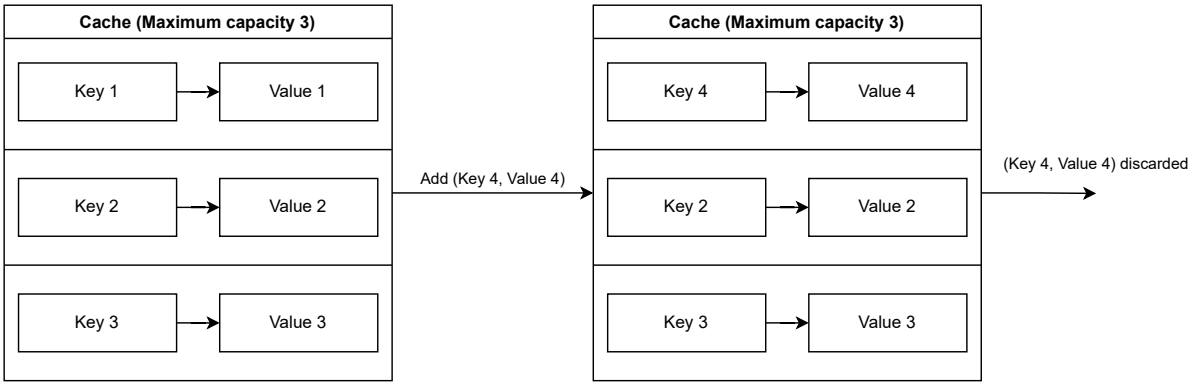
Take for example the following document schema:

```
{
    "name!": "string",
    "hobbies!?": ["string"].
    "university": {
        "name!": "string"
    }
}
```

The constraints would be as follow:

- `Constraints`, contains the following constraints:
  - `FieldConstraint` on the field `name`, it is required an non-nullable. It contains the following constraint:
    - `TypeConstraint` on the type `String`.
  - `FieldConstraint` on the field `hobbies`, it is optional an non-nullable. It contains the following constraint:
    - `ArrayElementsConstraint` that contains the following constraint:
      - `TypeConstraint` on the type `String`.
  - `FieldConstraint` on the field `university`, it is required an nullable. It contains the following constraint:
    - `Constraints`, contains the following constraints:
      - `FieldConstraint` on the field `name`, it is required an non-nullable. It contains the following constraint:
        - `TypeConstraint` on the type `String`.

Cache



The interface `Cache<K, V>` represents objects that are responsible for caching, where they are basically mostly like an ordinary map, *except* that there is a *limit* to the number of objects that can be cached, therefore implementations of the interface have to decide which element to remove in order to accommodate new items.

The only implementation in the application is the `LRUCache` class, which removes the *least recently used* item to accommodate new items upon reaching maximum capacity. It uses a doubly linked list and a map to achieve `O(1)` time complexity on all operations. It also uses `ReentrantReadWriteLock` to achieve thread-safety by locking the read lock on read operations, and locking the write lock on write operations, meaning that reads can be done even if there are other concurrent reads, but not if there are writes, and writes can be done only if there are no other reads or writes.

Storage

Objects in this level are responsible for tasks such as:

- Storing documents.
- Retrieving documents.
- Updating documents.
- Removing documents.

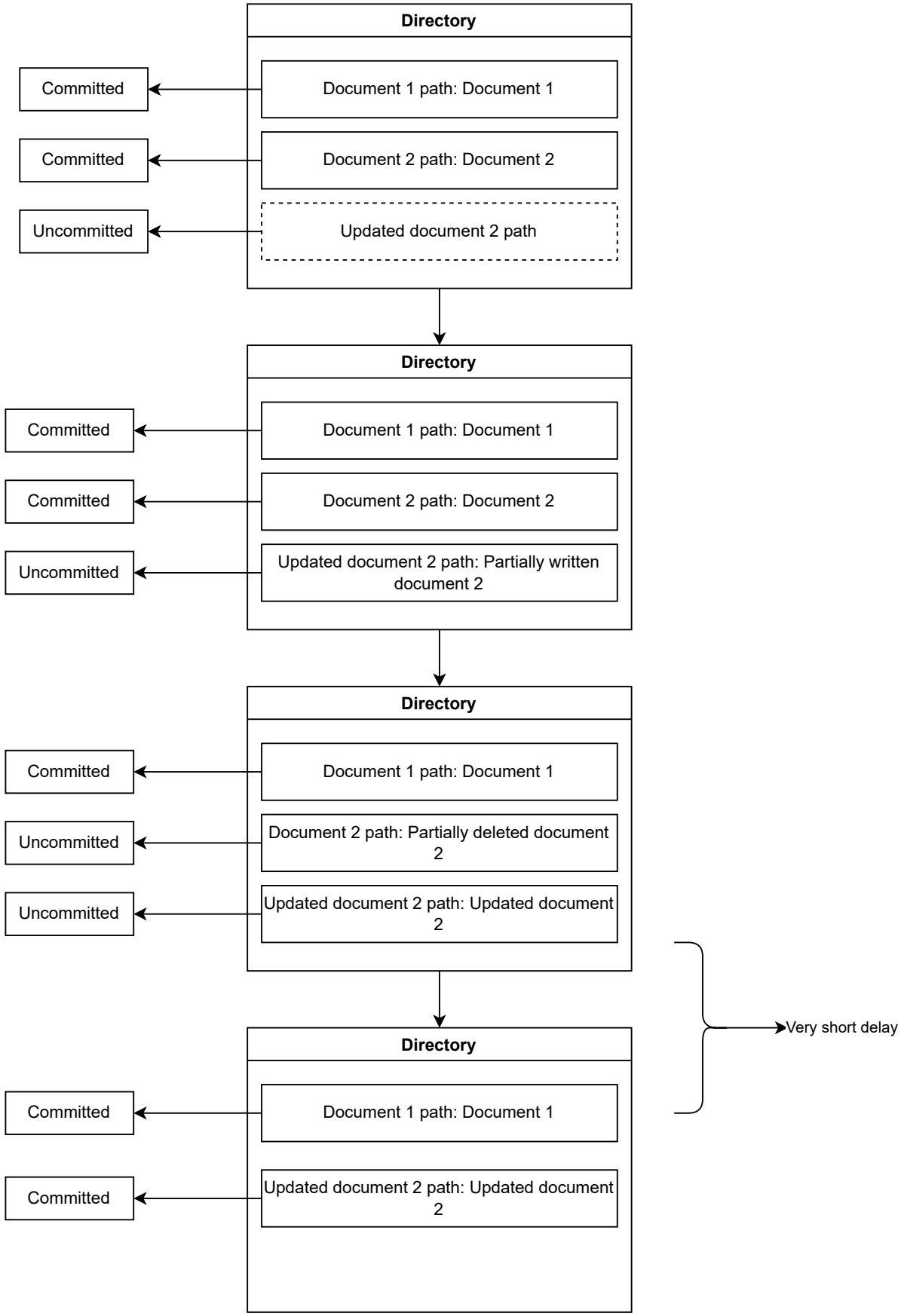
The interface that describes the object of this layer is the `StorgeEngine` interface, which describes an object that does the aforementioned tasks. Storage engines store documents in *directories*, and the exact path that the document must be stored at is not specified to the storage engine, only the *directory* is specified, since the exact path does not matter, the only requirement is that the file is stored, and storage engines are responsible for generating the exact path.

Storage engines respond to reads by returning an *optional* `Document` object, which is empty if there was an error while attempting to read - *such as the file not existing* -, and respond to writes and updates by returning an instance of `Stored<Document>`. `Stored<T>` represents an object that is stored in persisting memory, and contains a reference to that object and the `Path` of the object.

The default storage engine is `NonBlockingStorageEngine`. It stores and retrieves document while not blocking reads during updates. It keeps track of uncommitted files that will be ignored in order to avoid reading partially written or deleted files, therefore ensuring atomicity.

Updates in `NonBlockingStorageEngine` work as follow:

1. First, the updated documents path is generated and marked is uncommitted.
2. The storage engine will start writing the updated document at the previously generated path. Since the path is marked as uncommitted, any reads to the document currently being written will be ignored until it is committed, therefore avoiding duplicates since the original file still not deleted.
3. The storage engine will then delete the original document, after finishing writing the updated document. Note that the deleted file is marked as *uncommitted* as it is being deleted, to avoid reading partially deleted documents.
4. Finally, it commits the updated document as *committed*, so future reads will not ignore it.



Deletes are handled by sending the delete request to an *executor service*, which ensures that calling threads *do not have to wait for deletes to finish*. This is the reason that the delay between between the deleting the old document and committing the new document when updating is very short. It also marks the deleted file as *uncommitted* before the delete starts in order to avoid reading it while it being deleted.

Writes work similarly, with the notable exception that in the case of failures the application will delete any partially written documents, to ensure atomicity.

There are two other implementations of `StorageEngine`:

- `CachedStorageEngine`: This storage engine encapsulates an existing `StorageEngine` and adds caching capabilities on top of it, by utilizing a `Cache<Path, Document>`, and any future request is handled by first checking if the path is stored in the cache, if it is then the cached document is retrieved and returned, otherwise the encapsulated storage engine is used to retrieve the document, and the result is cached to make future reads faster.
- `SecureStorageEngine`: This storage engine encapsulates an existing `StorageEngine` and protects the stored document from change. It does so by storing the document inside the document below. The `verification` field is calculated when writing the document as follow: `verification = sha256(document.hashCode() + secret)`, where `secret` is a predefined, randomly generated string. When reading documents, `SecureStorageEngine` recomputes the verification code of the read document, and if it matches the verification field, then the document is accepted, otherwise it is rejected.

  ```
  {
      "verification": "verification",
      "document": {
          ...
      }
  }
  ```

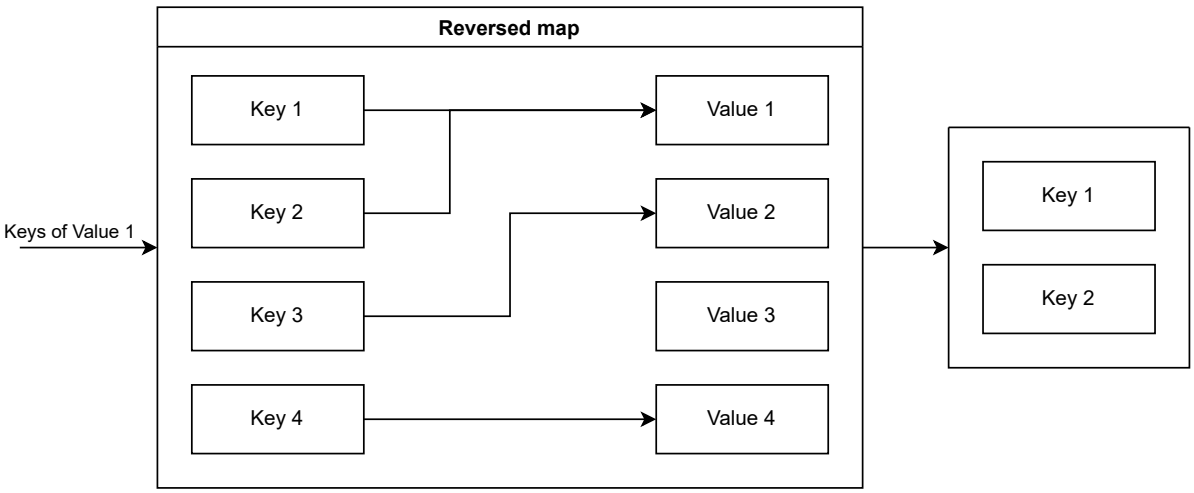
The storage engine used in the application is a cached, secured storage engine, created by wrapping a `NonBlockingStorageEngine` with a `SecureStorageEngine`, and then wrapping the secure engine with a `CachedStorageEngine`.

Indexing

Indexing allows the application to respond to certain queries much faster by not having to search the entire database, instead the indexes store that paths of documents that match the query, and when that query is made, the list of paths is retrieved directly, making queries much faster.

The building block for indexing in the application is the `Index` interface, it represents an object that is responsible for storing an index on a single query, since each `Index` contains a field document - *as described earlier* -. `Index` objects receive documents and their paths, and then whenever a query on the index exact fields is made, the index is consulted and then returns a list of paths containing documents that match the criteria.

The only `Index` implementation in the project is `HashedIndex`, which internally uses a `ReversedHashMap` to respond to queries. The `ReversedHashMap<K, V>` is a special data structure, it acts mostly like an ordinary map, except it allows users to query for a *value* and receive a *collection of all key values that correspond to that value*, it does using two `HashMap` internally, and uses `ReentrantReadWriteLock` for thread-safety. `HashedIndex` uses the reversed hash map with the path to the document as the key, and the document values *(for the index fields only)* as the value. Then, upon receiving queries, it uses the reversed hash map to respond.



While `Index` objects store a single index, `IndexesCollection` store a group of indexes, each represented as an `Index` object, of course. It simplifies dealing with multiple indexes, and is also responsible for persisting and loading the indexes. The `IndexesCollectionFactory` is responsible for instantiating instances of `IndexesCollection`, and is used in dependency injection to avoid making other parts of the application tightly coupled to a specific `IndexesCollection` implementation.

One important thing to note is that `IndexesCollection` only stores a *description* of the index, rather than the entire index, and the index itself is rebuilt every time the application is started, due to difficulties in serializing the index.

> ***IMPORTANT*:** Indexes support the creation of *unique indexes*, which bans duplicate values on that index.

Documents collections

Documents collection are responsible for storing and retrieving documents. They are represented by the interface `DocumentsCollection`, which provides useful functionalities such as `findDocuments(Document crieria)` and `addDocument(List<Document> documents)`. As can be seen from these two methods, the main difference between `DocumentsCollection` and `StorageEngine` is that the latter deals with issues like the path of document, and all of its operations are on a single document at a time, while the other class is not concerned with lower level issues like where to store the documents, but rather *delegates* it to the storage engine, and is further capable of executing operations dealing with *multiple documents*. For example, storage engine can only read one document after providing it with the path, while documents collections use storage engine to find all documents that match a certain criteria.

There are two types of collections, *basic* documents collections, and *indexed* documents collection. Basic documents collections provide the bare minimum requirement for collection, it does *not* do things like indexing or keeping track of the schema, and documents in basic collections are *not* required to have an `_id`. Indexed collections, on the other hand, do keep tracks of indexes and the schema, and require documents to have an `_id` field, which it injects itself into documents, where the `_id` the `Document.withId()` method. It also automatically creates a unique index on the `_id` field.

Each indexed collection stores the following:

- The schema: The schema describes the structure that all of the collection's documents must follow.
- The documents.
- The indexes: The database allows the users to create an index on a property - *or more than one property, as multi-property indexing is also supported* -, which will make all future requests on that index much faster, since the database now no longer needs to search the entire collection to find the results.

While most operations of documents collection are executed in straightforward manner, one thing that deserves to be noted is how documents collections handle updates. The `DocuemntsCollection.updateAll(Document criteria, Document update)` finds all document that matches the `criteria`, and then updates them by overriding their fields with the values from the argument `update`, using `Document.overrideFields` that was discussed earlier.

Database

Similar to how documents collections store multiple documents, databases manage multiple collections. The main interface that describes databases is `Database`. It provides a number of methods that simplify dealing with databases. The default implementation is called `DefaultDatabase`.

There is one database that is created by default, and it is the metadata database, and it is created in a `MetadataDatabase`, which was described earlier in the design overview.

Databases manager

And finally, the interfaces `DatabasesManager` describes an object that is responsible for managing databases. The default implementation is called `DefaultDatabasesManager`.

Security

Security is handled by Spring Security, it is configured to secure all requests using `HttpBasic`, and the user details are retrieved using an implementation of `UserDetailsService`, which is called `DatabaseUsersDetailsService`, which retrieves users' information from the metadata database, and wraps it in a `DatabaseUser` class that contains all information necessary for authorization.

Users and roles are stored in the metadata database, and the users collection have the following schema:

```
{
    "roles!": ["string"],
    "authorities!": ["string"],
    "password!": "string",
    "username!" :"string"
}
```

While the roles collection have the following schema:

```
{
    "role!": "string",
    "authorities!": ["string"]
}
```

Both collections are indexed, and the user collection creates a unique index on the *username*, while the role collection creates a unique index on the *role*.

Authorities are represented using the `DatabaseAuthority` class, which implements Spring Security's `GrantedAuthority`. Authorities allow access to a single database operation, for example `READ_DOCUMENTS` or `CREATE_DATABASE`, therefore each `REST` endpoint requires a specific a specific authority to access, for example the endpoint `POST /databases/{database}/collections/{collection}` requires that the user have the authority `CREATE_COLLECTION`.

Roles contain a number of authorities, and act as a shortcut to avoid having to write dozens of authorities for new users every time. The application provides a number of pre-defined roles, while also allowing users to define their own custom roles, since roles are stored in an application and can be modified just like any other collection. The pre-defined roles are:

- `READER`: Can only read documents.
- `USER`: Can read, write, update and delete documents.
- `OWNER`: Can do anything that the `USER` can, in addition to being able to create indexes.
- `MANAGER`: Can do anything that the `OWNER` can, in addition to being able to create and remove collections,
- `ADMIN`: Can do anything.

Requests and requests handlers

As discussed in the overview, the controllers wrap the incoming requests in a `DatabaseRequest`, which contains all the information necessary to handle the request, which are:

- The operation, which is represented using the Enum `DatabaseOperation`, which also contains information about whether the operation mutates the database state or not.
- The database and collection that are targeted by the operation.
- The payload, which is represented by a `Payload` object, and contains all the information about the payload, such as the added documents if it is an `ADD_OPERATION`.

The `DatabaseRequest` is forwarded from the controller to the default `DatabaseRequestHandler`. The interface `DatabaseRequestHandler` describes objects that contain a single `handle(DatabaseRequest)` method, which returns a `DatabaseResponse` upon calling. Handlers are responsible for resolving requests, and there are a number of them in the application, each of them handles request in a different way.

The default request handler takes in request and applies a number filters to it using `DatanaseRequestFiltersManager`, which keep tracks of all `DatabaseRequestFilter` instances in the application, and applies them to the request. Each filter modifies the request in some way, for example the `AddUserFilter` takes in request that attempts to add documents to the metadata's users collection and encrypts the password.

Then, if the `DatabaseOperation` mutates the database state, then the request is forward to the operations handler and then to the synchronization handler. The operations handler keeps track of all instances of `OperatoinsHandler`, where each `OperatoinsHandler` contains a number of methods, each annotated with `@DatabaseOperationMapping(DatabaseOperation value)`, which specifies the operation that the request must have in order to have the method invoked. Therefore, the operation handler maps the incoming request to the method that should be invoked to handle it. The synchronization handler forwards the request to a number of remote nodes, by making a `POST` request to the node `/exposedNode`, which is handled by taking in the completely wrapped `DatabaseRequest` and forwarding it directly to the operations handler, skipping the default handler entirely.

If the `DatabaseOperation` does not mutate the database state, then it is forward to the `LoadBalancingHandler`, which forwards the request to any of the remote nodes, if there are any, if there are none then it is forwarded to the operations handler.

Design discussion

Managing thread-safety

Thread-safety was achieved in three ways:

- Preferring *immutable* objects whenever possible, since they are readily thread-safe.
- Using thread-safe objects and collections, such as `ConcurrentHashMap`.
- Using locks if necessary, such as in `Cache` and `ReveresedHashMap`.

Dependency injection

Most of the dependencies between objects were injected using Spring, which made the design much more readable and easier to maintain. For example, the class `DefaultOperationHandler` depends on a `List<OperationHandler>`, and Spring automatically collects all instances of `OperationHandler`, wraps them in a list, and injects the dependency in the object.

Used design patterns

Singleton

The singleton design pattern was used for the utility class `RemoteNodeRestClient`.

Builder

The builder design pattern was used in multiple places in the code, whenever the number of arguments was too big. Lombok's `@Builder` annotation was used to automatically generate builder classes in most cases.

Factory

The factory design pattern was used for documents collections and documents databases, to decouple the creation of instances of these objects from the rest code, since the factory will be injected whenever one of these objects is to be created, and then used to create instances of these objects. In the future, if there was a new implementation of these classes, all that is needed to be done is to create a factory for these classes to replace the old factories, and they will be injected wherever they are needed, with no change to other parts of the code at all.

Command

The command design pattern was used by wrapping requests in a `DatabaseRequest` object, which can then be forwarded and passed around the code easily.

Decorator

The decorator design pattern was used in storage engines, for example the cached storge engine takes in an existing storage engine and adds on top of it caching capabilities.

SOLID principles

Single responsibility principle

This principle was followed by making sure that each class and method have one and only responsibility, and additional responsibilities are delegated to other specialized classes. For example, instances of `StorageEngine` can only do `I/O` related things.

Open closed principle

The principle was followed by allowing to easily extend functionalities. For example, if a new filter is to be added, all that needs to be done is to create a new bean that implements the `DatabaseRequestFilter` interface, and the filter will be added to the chain automatically.

Liskov Substitution Principle

This principle was followed by ensuring all subclasses can replace their super-classes without any issue, since they should only extend their functionalities and not remove or change any existing functionality.

Interface Segregation Principle

This principal was followed by making classes in the code depend on *interfaces* rather that on concrete implementations. For example, no section in the code depends on a concrete implementation of `IndexedDocumentsCollection`, but rather on the interface itself, and object creation is done by injecting the appropriate factory.

Dependency Inversion Principle

This principle was followed by ensuring that high level modules depend on *abstractions* rather than on *concrete* implementations. For example, `DatabasesMangaer` does *not* depend on a specific implementation of the lower level Database`, but rather on the interface itself, which will make changes in the future much easier.

Effective Java

Creating and Destroying Objects

Consider static factory methods instead of constructors

Static factory methods were used in order to make the code more readable, for example there are a number of static factory methods for `Document`, such as `Document.of(Object.... elements)`.

Consider a builder when faced with many constructor parameters

Builders were implemented whenever the number of arguments became too high, and they were generated mostly using project Lombok's `@Builder` annotation. For example, the classes `DatabaseRequest` provides a builder.

Enforce the singleton property with a private constructor or an Enum type

While in most cases, most components are created as a singleton by Spring when creating the beans, there were some cases where the singleton property was explicitly enforced, such as in the utility class `RemoteNodeRestClient`.

Enforce non-instantiability with a private constructor

For some classes, the constructor was made private when it made sense, in order to prevent the user from creating new instances of the class when this is not allowed, such as for singletons.

Prefer dependency injection to hardwiring resources

Dependency injection was used very heavily in the application by utilizing *Spring*, which manages the creation of beans and auto-wiring dependencies.

Avoid creating unnecessary objects

No unnecessary objects were created, some fields were made static to ensure that only one object exists and is shared by all instances, and dependency injection further helps avoiding creating new unnecessary objects. For example, whenever a `String` was to be matched against a regular expression fairly regularly, the usage of `String.matches` was avoid, since it internally

creates a new instance of `Pattern` every time, which is not cheap, and instead a single instance of `Pattern` was created and reused.

Avoid finalizers and cleaners

They were not used at all in the application.

Prefer try-with-resources to try-finally

Try with resources was always used whenever possible, for example `NonBlockingStorageEngine` contains the following code segment:

```
private void writeDocumentAtPath(Document document, Path documentPath) {
        try (BufferedWriter writer = Files.newBufferedWriter(documentPath)) {
            add(documentPath);
            writer.write(document.toString());
        } catch (IOException e) {
            FileUtils.deleteFile(documentPath);
            throw new UncheckedIOException(e);
        } finally {
            commit(documentPath);
        }
    }
```

Methods common to all objects

Always override `toString`

The `toString` method was always overridden, mostly using project Lombok's `@ToString` annotation, except for some places where a more custom behavior was desirable.

Classes and interfaces

Minimize the accessibility of classes and members

All fields are `private` by default.

In public classes, use accessor methods, not public fields

Accessors were always used when necessary, either through implementing getters or by making the class a `record`, which automatically generate getters.

Minimize mutability

Many immutable classes were designed since they are thread-safe, for example the class `Document` is thread-safe. Moreover, records were used in many places since they are immutable and reduce boilerplate.

Favor composition over inheritance

Composition was favored over inheritance, for example in `StorageEngine`, where classes decorate each other rather extend them, or in `DocumentsCollection`, where `IndexedDocumentsCollection` does *not* extend the basic documents collection, but rather is composed of it.

Prefer interfaces to abstract classes

Interfaces were always considered first, and abstract classes were used only when necessary.

Favor static member classes over non-static

Fields were made static whenever possible, in order to avoid recreating objects when not necessary.

Generics

Don't use raw types

Raw types were *never* used.

Prefer lists to arrays

Lists were used as default throughout the code, instead of arrays.

Favor generic types

Generic types were used whenever it was possible, for example in `Cache<K, V>` and `ReversedMap<K, V>`.

Favor generic methods

Generic methods were used, for example in the following static factory method from the class `Stored<T>`:

```
public static <T> Stored<T> createStoredObject(T object, Path path) {
    return new Stored<>(object, path);
}
```

Enums and annotations

Use Enums instead of int constants

Enums were used instead of int constants, for example in the Enum `DatabaseOperation`.

Consistently use the `@Override` annotation

The `@Override` annotation was always used.

Lambdas and streams

Prefer lambdas to anonymous classes

Lambdas were always preferred over anonymous classes.

Prefer method references to lambdas

Method references were used whenever possible, and in some cases private methods were created specifically to be used in streams, for example the private method `addDocumentToIndexes(Path documentPath)` was created to be used in the following stream in `HashedIndexesCollection.populateIndexes()`:

```
FileUtils.traverseDirectory(documentsDirectory)
                .filter(FileUtils::isJsonFile)
                .forEach(this::addDocumentToIndexes);
```

Methods

Design method signatures carefully

Method names were chosen carefully, and long parameter lists were avoided.

Use overloading judiciously

Overloading methods was rarely used.

General Programming

Minimize the scope of local variables

Variables are always declared in the exact scope where they are used, limiting their scope.

Prefer for-each loops to traditional for loops

For-each loops were always considered first, and traditional for loops were only used when the for-each loop is impossible, such as when the order of the element in the array is needed in loop's body.

Know and use the libraries

Built-in and third party libraries were used to avoid recreating the wheel.

Refer to objects by their interfaces

Objects were *always* referred to by their interfaces.

Concurrency

Synchronize access to shared mutable data

Access to shared mutable data was synchronized by using thread-safe collections such as `ConcurrentHashMap`, or if the mutable data was an application defined class, it was ensured that class itself is thread-safe.

Avoid excessive synchronization

Excessive synchronization was avoided by carefully applying it only when necessary.

Clean Code

Meaningful names

The names of variables, methods and classes were chosen carefully to be meaningful and descriptive, they clearly display the intent.

Avoid disinformation

When naming, care was taken to avoid misleading the reader with bad names.

Make meaningful distinctions

Concepts with different meaning have different, distinct names.

Use searchable names

Single letters and meaningless names were avoided.

One word per concept

Each concept has a different, unique name. For example, adding a document is called *adding*, while creating a database is called *creating*, since when adding a document, the document already exists but is then added to the collection, but when creating a database, it did not exist before.

Class names

They are always nouns.

Methods names

They are always verbs.

Functions

Small, do one thing

Functions are always small and do only one thing, if a function is too long then methods are extracted from it until it becomes small.

Function Arguments

The number of function arguments was minimized as much as possible,

Flag arguments

Flag arguments were always avoided.

Comments

Comments were not used.

Error Handling

Use exceptions rather than return codes

Exceptions were thrown when errors occur, rather than returning an error code.

Don’t return null

The code never returns null, and in cases where the code needed to convey that sometimes there are no results, the `Optional<T>` class was used.