Database Standards and Guidelines

Version: 1.0.1

Status: In Review

Date: 14 April 2024

Prepared by:			
Name	**Metric number**	**Department**	**Date**
Tengku Puteri Emilia binti Tengku Nazaruddin Shah	U2000865	QA (Standard & Compliance)	14 April 2024
Approved by:			
Name	**Metric number**	**Department**	**Date**
Released by:			
Name	**Metric number**	**Department**	**Date**

Version History

Version	Release Date	Section	Amendments
1.0.0	14 April 2024	All	Original Document
1.0.1	22 April 2024	All	Original Document

Change Record

Version	Date	Affected Section	Change Request #	Reason/Initiation/Remarks

Table of Contents

1.0 Description

1.1 Purpose

This documentation is tailored for the project's database requirements, specifically focusing on Firebase-based setups utilizing NoSQL. It provides standards, guidelines, and design principles crucial for developing the Optical Character Recognition (OCR) System for Handwritten Multiple-Choice Question (MCQ) Answer Sheet Evaluation project, emphasizing the adaptation of SQL code rules for NoSQL within Firebase.

This document emphasizes the database standards and guidelines, design principles, and naming conventions specifically for Firebase database projects. It serves as a reference for the database management teams and software development teams involved in this project, aiming to streamline the creation process while ensuring the resulting code is efficient, maintainable, and adheres to consistent practices within the Firebase NoSQL environment

1.2 Scope

All database code checked into OCR database project' permanent repository shall adhere to the standards and guidelines specified in this document and its references

2.0 Related Documents

2.1 References

Best practices for Cloud Firestore | Firebase. (n.d.). Firebase. https://firebase.google.com/docs/firestore/best-practices

Saddique, A. (2024, February 3). *Firebase Firestore Database - Abubakar Saddique - Medium*. https://medium.com/@abubakarsaddqiuekhan/firebase-firestore-database-78dae380d5cf

Get started with Cloud Firestore Security Rules | Firebase. (n.d.). Firebase. https://firebase.google.com/docs/firestore/security/get-started

2.2 Abbreviations and Acronyms

2.3 Glossary

Collection: A group or the set of documents and it can be stored independently like in the case of authentication, a collection of the documents name should be created respectively.

Documents: The unit of storage is the document in Cloud Firestore. A document is a lightweight record that contains fields, which maps to values. Each document is identified by a name. A document is nothing more than the key value pair and it should be stored inside the collection. A collection can be created inside the document. The value that you are able to store in is the primitive type like string, number, boolean, etc

3.0 Code Standards

3.1 Firebase Project Name

A descriptive and unique name for the Firebase project must be chosen and it needs to reflect the purpose or theme of the software development project. Since the project is regarding Optical Character Recognition (OCR) System for Handwritten Multiple-Choice Question (MCQ) Answer Sheet Evaluation, it might be best to name it `OCRSystemforMCQAnswerSheet` as a suggestion.

3.2 Database Naming

- Database Instance Name: Firebase allows creating multiple database instances. Name them according to their purpose.
  For example: `main_database`
- Collection Names: Use plural nouns for collection names and be descriptive.
  For example: `students`, `answer_sheets`, `scores`
- Document IDs: Keep document IDs concise and meaningful. They should ideally reflect the data they represent. The id can be meaningless which is auto generated by Firebase or meaningful. Meaningless id will never change, and it is unique, but it is hard to fetch from the database, while meaningful id is easier to fetch but it might not be unique, and it might change dynamically. It is suggested to use meaningless ids when writing client code and only use meaningful ones in case when the data is inserted manually in a collection where they will not clash.
  For instance: `answer_sheet_johndoe`, `total_score_johndoe`.
  1. Avoid the document IDs . and . .
  2. Avoid using / forward slashes
  3. Do not use monotonically increasing document IDs such as Customer1, Customer2.. (such sequential IDs can lead to hotspots that impact latency)
- Field Names: avoid the following characters in field names because they require extra escaping such as period (.), left and right brackets ([]), asterisks (*), backtick (`)

3.3 Cloud Functions Naming

- Function names should be descriptive names that reflect the purpose of the function. It must be precise, and it should follow the camelCase naming conventions as a suggestion
  For example: `processAnswerSheet`, `returnEvaluationResults`
- If the function is triggered by an event, it should be reflected in the function name. If the function is triggered when an answer sheet is uploaded to the database, it should be

named:
For example: `onAnswerSheetUploaded`

3.4 Storage Bucket Naming

Bucket names should use lowercase letters and dashes to separate words in bucket names

For example: `ocr-mcq-evaluation-images`

3.5 Authentication and Authorization

This development project has different user roles, and it should be named descriptively in the database system, `admin` is an admin user with access to all features, `evaluator` is a user who is responsible for evaluating answer sheets, and `student` is a regular user who submits answer sheets for evaluation.

3.6 API Keys and Service Account Credentials

Ensure that the API keys and service account credentials have clear, descriptive names, and keep them secure for security and data integrity purposes, so it might not be lost easily. For instance, API key for accessing Firebase services could be `ocr-mcq-evaluation-api-key` and service account credentials for server-to-server interaction could be named like `ocr-mcq-evaluation-service-account`

4.0 Flutter Coding Practices

4.1 Add dependencies

To use the firebase firestore in the software development project you must add the dependencies by running this command with flutter

```
$ flutter pub add cloud_firestore
```

This will add a line like this to the package's pubspec.yaml (and run an implicit flutter pub get):

dependencies:
cloud_firestore: ^4.16.1

Now in the Dart code, you can use:

```
import 'package:cloud_firestore/cloud_firestore.dart';
```

4.2 Create class

Class function for Firebase Firestore database should be created and the class name should be like:

```
Class FirebaseFirestoreDatabase{ }
```

With this class function, the code must be written to interact with Firebase Firestore Database. The CRUD operations method (add, fetch, delete, update) is important in this class as it enables the developers to manage the database accordingly.

4.3 Adding data to the Firebase Firestore

First, create the instance of the Firebase Firestore database.

```
FirebaseFirestore firebaseFirestore = FirebaseFirestore.instance;
```

The instance is the static method of the firebase firestore class that will return the object or the instance of the firebase firestore class.

4.4 Create a collection

To create a collection, you must have at least one document in it. Collection Reference is an abstract class that is used to make a collection inside the Firestore database.

```
abstract class CollectionReference<T extends Object?> implements Query<T>
```

To write the code for adding the first collection to the Firebase Firestore, use the collection method of the Firebase Firestore class to interact with the collection of the Firebase Firestore.

```
CollectionReference<Map<String, dynamic>> collection(String
collectionPath)
```

4.5 Gets a [CollectionReference] for the specified Firestore path

In this method, it will input a string which is the name of the collection reference and this will return the instance of the collection reference class.

```
CollectionReference firstCollection =
firebaseFirestore.collection("my_first_collection");
```

The code for collection reference is written successfully, and it needs to have documents inside it.

4.6 Create Document

Using the document method of the collection reference class to interact with the document of the Firebase Firestore database. Then, input the name of the document that needs to be created.

```
DocumentReference<Object?> doc([String? path])
```

This will return a DocumentReference with the provided path. If no [path] is provided, an auto-generated ID is used for ID documentation.

4.7 Add the Data in the Document

Use the set method, which sets data on the document, overwriting any existing data. If the document does not yet exist, it will be created. If [SetOptions] are provided, the data can be merged into an existing document instead of overwriting.

```
Future<void> set(Object? data, [SetOptions? options])
```

Now, add the data here in the form of the key value pair.

```
await xCollection.doc("x_document).set({
"string": "Student1"
"int": 5
"boolean": true,
"map": {"map_key", "map_value"}
"array": ["first", "second", "third", "fourth"], })
```

In the same way, we can make multiple collection, and create multiple documents in each collection

4.8 Fetching Data from Firebase Firestore

To fetch the data from the Firebase Firestore, we need to make the collection reference in the same that has been created before

```
CollectionReference                    xCollection                        =
firebaseFirestore.collection("my_x_collection");
```

Using the get method of the collection reference, we will get the data of the document

```
Future<QuerySnapshot<Object?>> get([GetOptions? Options])
```

4.9 Fetch the Documents for the Query

This will return the future of the query snapshot class

QuerySnapshot fetchDocumentsData = await users.get();

The query snapshot is an abstract class

```
abstract class QuerySnapshot<T extends Object?>
```

Contains the results of a query. It can contain 0 or more [DocumentSnapshot] objects. Now using the docs method of the query snapshot class we will fetch the list of the document

```
List<QueryDocumentSnapshot<T>> get docs
```

The docs method is the getter of the document snapshot class that is used to get or return the list of the query snapshot documents. Now we have the list of all the documents in the given collection. Using the for-each loop we will iterate the list and uses the documents data.

```
QuerySnapshot fetchDocumentsData = await xCollection.get();
fetchDocumentsData.docs
      .forEach((DocumentSnapshot documentSnapshot) {
      print(documentSnapshot.data()); });
```

It will fetch the data of all the documents that are present inside the given collection. If you want to fetch the data of the specific document, then you must have to give the name of the document

```
DocumentSnapshot fetchData = await xCollection
      .doc("x_document")
      .get();
print(fetchData.data());
```

4.10 Listen to Change in Document

If you want to listen to a document if there is any change in it, you need a method of the name snapshots method.

```
Stream<DocumentSnapshot<Object?>> snapshots({bool includeMetadataChanges =
false})
```

This code will notify the document updates at this location. An initial event is immediately sent, and further events will be sent whenever the document is modified. That method will return the stream of the documents' snapshot class. You must listen to this stream using the listen method.

```
xCollection
      .doc("x_document")
      .snapshots()
      .listen((event) {
      print(event.data());
});
```

4.11 Updating Data

The update method is used to update the data of the Firebase Firestore

```
Future<void> update(Map<Object, Object?> data)
```

This code is used to update data on the document. Data will be merged with any existing document data. Objects key can be a String or a FieldPath. If there is no specific document that exists yet, the update will not perform. In the update method, you have to add the data in the form of key value pair as we have done before in the set method for adding the data to the Firebase Firestore class.

```
void updateDataInFirebaseFirestore() {
 firebaseFirestore
 .collection("my_x_collection")
 .doc("x_document")
 .update({
 "string": "Student1",
 "int": 70,
 });}
```

The document name, collection name and the key of the document's that is paired with that specific value must be similar to update the value of the existing document. If they don't match, then the update will not be performed and will fail instantly.

4.12 Delete Document

To delete the document in the Firebase Firestore database, we will use the delete method of the document reference class.

```
Future<void> delete()
```

This line of code will delete the current document from the collection.

```
void deleteDocumentFromFirebasefireStore() async {
      await firebaseFirestore
            .collection(“my_x_collection”)
            .doc(“x_document”)
            .delete();
}
```

This function will delete the given document from the given collection. If the document or collection's name is not matched, the deletion process will not be performed.

5.0 Cloud Firestore Coding Practices

5.1 Indexes

5.1.1 Reduce Write Latency

The primary factor impacting write latency is index fanout. Effective strategies for minimizing index fanout include:

- Implement collection-level index exemptions. A straightforward approach is to disable Descending and Array indexing by default. Additionally, eliminating unused indexed values can help decrease storage expenses.
- Decrease the quantity of documents within a transaction. When dealing with a significant volume of documents, employing a bulk writer instead of an atomic batch writer can be more efficient.

5.1.2 Index exemptions

Case	Description
Large string fields	If there is a string field that holds long string values that is not used for querying, just simply cut the storage costs by exempting the field from indexing
High write rates to a collection containing documents with sequential values	Indexing a field that changes sequentially between documents in a collection, such as a timestamp, restricts the maximum write rate to 500 writes per second. However, if querying based on the sequential field isn't necessary, exempting it from indexing can bypass this limitation. In scenarios like IoT with frequent writes, a collection with time stamped documents might approach this 500 writes per second threshold.
TTL fields	The TTL field must be a timestamp. Indexing on TTL fields is enabled by default and can affect performance at higher traffic rates. As a best practice, add single-field exemptions for the TTL field
Large array or map fields	Large array or map fields can approach the limit of 40,000 index entries per document. If querying is not based on a large array or map field, you should exempt it from indexing.

5.2 Read and Write Operations

- The maximum rate at which an app can update a single document varies significantly depending on the workload.
- Utilize asynchronous calls, when possible, over synchronous ones to minimize latency impact. For instance, consider an application needing both a document lookup and query results before rendering a response. If there's no data dependency between the lookup and the query, there's no need to synchronously wait for the lookup to finish before initiating the query.
- Avoid using offsets and opt for cursors instead. Offsets merely prevent the skipped documents from being returned to the application, but they are still internally retrieved, affecting query latency and incurring read operation costs for the application.

5.2.1 Transactions retries

The Cloud Firestore SDKs and client libraries automatically retry failed transactions to deal with transient errors. If the application accesses Cloud Firestore through the REST or RPC APIs directly instead of through an SDK, the application should implement transaction retries to increase reliability.

5.3 Designing for Scale

5.3.1 Updates to a single document

When designing the app, assess the speed at which single documents are updated by conducting load testing to gauge performance accurately. The maximum rate of updating a single document varies significantly depending on factors like write rate, request contention, and the number of affected indexes.

Each document write operation not only updates the document itself but also any associated indexes, with Cloud Firestore synchronously applying these operations across a quorum of replicas. However, as write rates increase, the database may experience contention, leading to elevated latency or errors.

5.3.2 High read, write, and delete rates to a narrow document range

Avoid high read or write rates to lexicographically close documents, or the application will experience contention errors. The hotspotting issue will happen if it does any of the following:

- Creates new documents at a very high rate and allocates its own monotonically increasing IDs.
- Cloud Firestore allocates document IDs using a scatter algorithm.
- Creates new documents at a high rate in a collection with few documents.
- Creates new documents with a monotonically increasing field, like a timestamp, at a very high rate.
- Deletes documents in a collection at a high rate.
- Writes to the database at a very high rate without gradually increasing traffic.

5.3.3 Avoid Skipping Over Deleted Data

Avoid queries that skip over recently deleted data. A query may have to skip over many index entries if the early results have recently been deleted.

An example of a workload that might have to skip over a lot of deleted data is one that tries to find the oldest queued work items. The query might look like:

```
docs = db.collection('WorkItems').order_by('created').limit(100)
delete_batch = db.batch()
for doc in docs.stream():
  finish_work(doc)
  delete_batch.delete(doc.reference)
delete_batch.commit()
```

Each time this query runs it scans over the index entries for the **created** field on any recently deleted documents. This slows down queries.

To improve the performance, use the **start_at** method to find the best place to start. For example:

```
completed_items      =      db.collection('CompletionStats').document('all
stats').get()
docs = db.collection('WorkItems').start_at(
    {'created': completed_items.get('last_completed')}).order_by(
        'created').limit(100)
delete_batch = db.batch()
last_completed = None
for doc in docs.stream():
  finish_work(doc)
  delete_batch.delete(doc.reference)
  last_completed = doc.get('created')

if last_completed:
  delete_batch.update(completed_items.reference,
                      {'last_completed': last_completed})
  delete_batch.commit()
```

5.4 Privacy

- Avoid storing sensitive information in a Cloud Project ID. A Cloud Project ID might be retained beyond the life of the project.
- As a data compliance best practice, it is recommended to not store sensitive information in document names and document field names.

5.5 Prevent Unauthorized Access

There should be a necessary requirement/measure to prevent unauthorized operations on the database with Cloud Firestore Security Rules.

Using rules could avoid a scenario where a malicious user repeatedly downloads the entire database.

Every database request from a Cloud Firestore mobile/web client library is evaluated against your security rules before reading or writing any data. If the rules deny access to any of the specified document paths, the entire request fails.

Below are some examples of basic rule sets:

```
service cloud.firestore {
  match /databases/{database}/documents {
    match /{document=**} {
      allow read, write: if request.auth != null;
    }
  }
}
```

This rule allows read/write access on all documents to any user signed in to the application.

```
service cloud.firestore {
  match /databases/{database}/documents {
    match /{document=**} {
      allow read, write: if false;
    }
  }
}
```

This rule denies read/write access to all users under any conditions

```
service cloud.firestore {
  match /databases/{database}/documents {
    match /{document=**} {
      allow read, write: if true;
    }
  }
}
```

This rule allows read/write access to all users under any conditions. This is a serious *warning* to never use this rule set in production since it allows anyone to overwrite the entire database