**UNIVERSITY OF BUEA**

**FACULTY OF ENGINEERING AND TECHNOLOGY (FET)**

**DEPARTMENT OF COMPUTER ENGINEERING**

**COURSE: INTERNET PROGRAMMING AND MOBILE PROGRAMMING (CEF440)**

**REPUBLIQUE DU CAMEROUN**

***PAIX – TRAVAIL – PATRIE***

**MINISTERE DE L’ENSEIGNEMENT SUPERIEUR**

**REPUBLIC OF CAMEROON**

***PEACE – WORK – FATHERLAND***

**MINISTRY OF HIGHER EDUCATION**



**JUNE 2024**

**PRESENTED BY : GROUP 19**

|  |  |  |
| --- | --- | --- |
| **NAME** | **MATRICULE** | **SPECIALTY** |
| NKEMCHOU PIANKE OLIVIER | FE21A275 | SE |
| DJITUE TOGUE BRINDA SPACHELLE | FE21A171 | SE |
| TEGUE NONO MIKEL MODEIRO | FE21A321 | SE |
| REOUTADE ROLAND | FE21A301 | SE |
| AMARACHUKWU GODLOVE AHANONU | FE21A136 | SE |

TASK6: DATABASE DESIGN AND IMPLEMENTATION

**INSTRUCTOR : Dr. NKEMENI VALERY**

Table of content

[1. INTRODUCTION 2](#_Toc169809461)

[1.1 DEFINTION 2](#_Toc169809462)

[2. STEPS TO FOLLOW TO ENSURE GOOD DATABASE DESIGN? 2](#_Toc169809463)

[3. BIOMETRICS STUDENT’S ATTENDANCE DATABASE 3](#_Toc169809464)

[3.1 CONCEPTUAL MODELING 3](#_Toc169809465)

[3.1.1 ENTITY RELATIONAL DIAGRAM 3](#_Toc169809466)

[3.1.2 ENTITY RELATIONAL DIAGRAM DESCRIPTION AND CONSTRAINTS 4](#_Toc169809467)

[3.2 LOGICAL DIAGRAM 6](#_Toc169809468)

[3.2.1 RELATIONAL SCHEMA 6](#_Toc169809469)

[3.2.2 DRAWBACKS FACED WITH SQL DBMS 9](#_Toc169809470)

[4. NO SQL DATABASE 10](#_Toc169809471)

[4.1 WHAT IS NOSQL DATABASE 10](#_Toc169809472)

[4.2 CLASSIFICATION OF NOSQL DATABASES 10](#_Toc169809473)

[4.3 WHY DID WE CHOOSE NOSQL DATABASE 13](#_Toc169809474)

[5. FIREBASE/ FIRESTORE 14](#_Toc169809475)

[5.1 WHAT IS FIRESTORE 14](#_Toc169809476)

[5.2 WHY WE CHOSE FIRESTORE DATABASE FOR A FINGERPRINT STUDENT ATTENDANCE APPLICATION 14](#_Toc169809477)

[5.3 CLOUD FIRESTORE DATA MODEL 15](#_Toc169809478)

[5.3.1 Documents 15](#_Toc169809479)

[5.3.2 Collections 17](#_Toc169809480)

[5.4 STRUCTURE OF BIOTRACK DATABASE DOCUMENTS IN FIRESTORE 17](#_Toc169809481)

[i. Collection: Lecturers 17](#_Toc169809482)

[ii. Collection: Students 18](#_Toc169809483)

[iii. Collection: Course 18](#_Toc169809484)

[iv. Collection: CourseEnrolled 18](#_Toc169809485)

[v. Collection: Admin 19](#_Toc169809486)

[vi. Collection: Attendance Records 19](#_Toc169809487)

[5.5 SCRIPTS USED FOR FIREBASE/FIRESTORE CONFIGURATION AND INITIALIZATION 20](#_Toc169809488)

[5.6 TEST FUNCTION USED TO QUERY THE DATABASE, CREATE AND POPULATE SAMPLE DOCUMENT DATA FOR THE STUDENT’S COLLECTION 21](#_Toc169809489)

[6. CONCLUSION 22](#_Toc169809490)

[7. REFERENCES 23](#_Toc169809491)

# INTRODUCTION

Managing student attendance is a crucial aspect of educational administration, directly influencing academic performance and institutional efficiency. Traditional attendance methods, such as manual roll calls, are not only time-consuming but also susceptible to errors and manipulation. The introduction of biometric systems, specifically fingerprint recognition, offers a precise and efficient solution. This report details the database design and implementation of a Fingerprint Student Attendance App aimed at modernizing attendance tracking.

## 1.1 DEFINTION

Database Design can be defined as a set of procedures or collection of tasks involving various steps taken to implement a database.

The database design for a Biometrics Student Attendance App is a comprehensive series of procedures and tasks that encompass multiple steps essential for implementing a robust and efficient database system. This process involves defining data requirements, creating a detailed data model, and organizing data structures to support the functionality of the attendance app. The design ensures that the database can accurately store and manage student information, fingerprint data, and attendance records, while maintaining data integrity, security, and scalability. Through careful planning and execution, the database design process lays the foundation for an effective and reliable system that automates attendance tracking using fingerprint recognition technology.

Following are some critical points to keep in mind to achieve a good database design:

* Data consistency and integrity must be maintained.
* Low Redundancy
* Faster searching through indices
* Security measures should be taken by enforcing various integrity constraints.
* Data should be stored in fragmented bits of information in the most atomic format possible.

# STEPS TO FOLLOW TO ENSURE GOOD DATABASE DESIGN?

* **Step 1:** Determine the goal of your database, and ensure clear communication with the stakeholders (if any). Understanding the purpose of a database will help in thinking of various use cases & where the problem may arise & how we can prevent it.
* **Step 2:** List down all the entities that will be present in the database & what relationships exist among them.
* **Step 3:** Organize the information into different tables such that no or very little redundancy is there.
* **Step 4:** Ensure uniqueness in every table. The uniqueness of records present in any relation is a very crucial part of database design that helps us avoid redundancy. Identify the key attributes to uniquely identify every row from columns. You can use various key constraints to ensure the uniqueness of your table, also keep in mind the uniquely identifying records must consume as little space as possible & shall not contain any NULL values.
* **Step 5:** After all the tables are structured, and information is organized apply Normalization Forms to identify anomalies that may arise & redundancy that can cause inconsistency in the database.

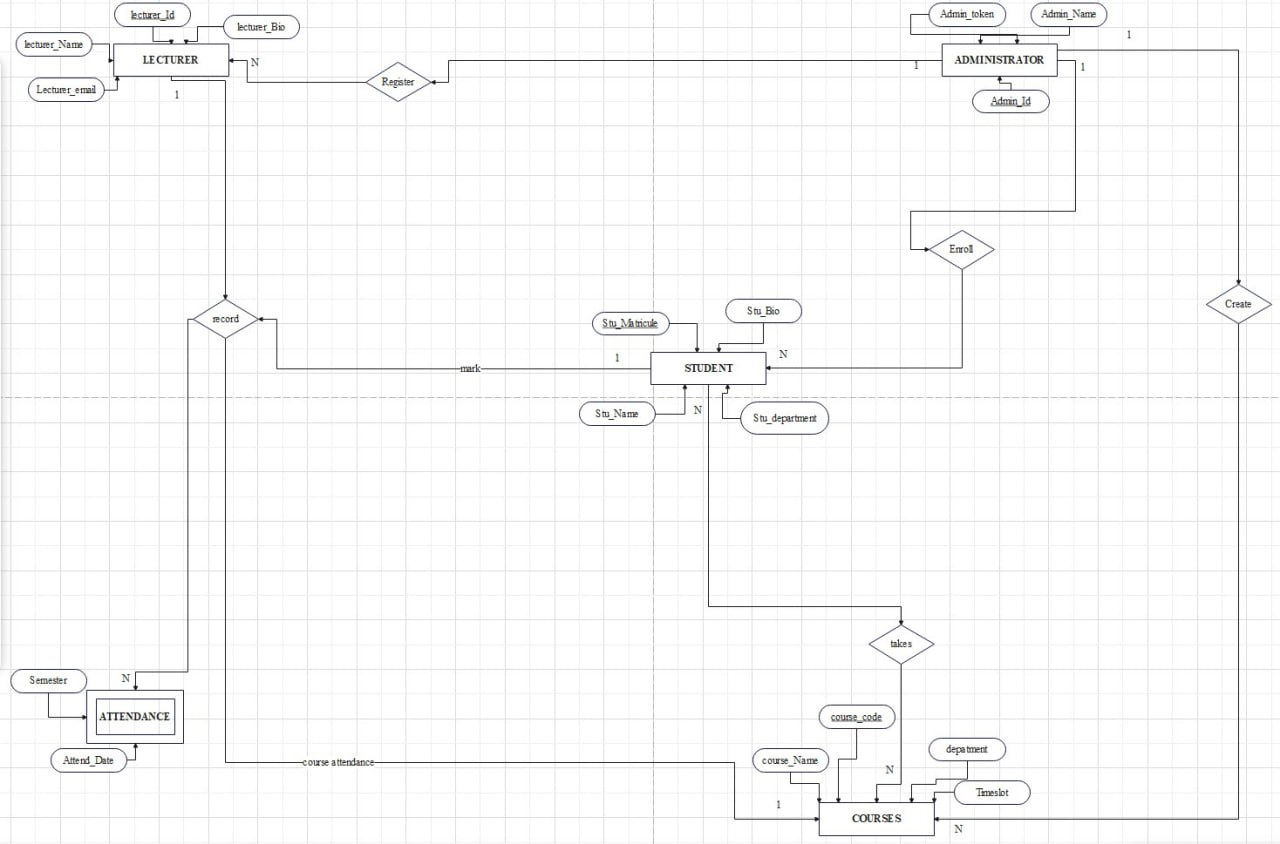
# BIOMETRICS STUDENT’S ATTENDANCE DATABASE

## CONCEPTUAL MODELING

Conceptual Data Model is a representations of data Examine and describe in depth your abstract, high-level business concepts and structures. They are most commonly employed when working through high-level concepts and preliminary needs at the start of a new project. They are typically developed as alternatives or preludes to the logical data models that come later. The main purpose of this data model is to organize, define business problems, rules and concepts. For instance, it helps business people to view any data like market data, customer data, and purchase data.

### ENTITY RELATIONAL DIAGRAM

The ER diagram represents a comprehensive system for managing lectures, students, courses, and attendance records. The relationships and cardinalities are well-defined to ensure accurate representation of the interactions between different entities in the system. Each entity is clearly described with its attributes, and the relationships establish how these entities interact within the system. This structured approach ensures that the system is scalable and maintains data integrity.



### 3.1.2 ENTITY RELATIONAL DIAGRAM DESCRIPTION AND CONSTRAINTS

*Entities and Attributes*

1. **LECTURER**
   * **Attributes**:
     + Lecturer\_ID (Primary Key): Unique identifier for each lecturer.
     + Lecturer\_Name: Name of the lecturer.
     + Lecturer\_Bio: Biometric parameter of the lecturer.
     + Lecturer\_Email: Email address of the lecturer.
   * **Description**: Represents registered lecturers by Administrator who teaches courses and records attendance.
2. **ADMINISTRATOR**
   * **Attributes**:
     + Admin\_ID (Primary Key): Unique identifier for each administrator.
     + Admin\_Token: Security token for administrator access.
     + Admin\_Name: Name of the administrator.
   * **Description**: Represents an administrator who manages course, enrolled student and Instructors
3. **STUDENT**
   * **Attributes**:
     + Stu\_Matricule (Primary Key): Unique identifier for each student.
     + Stu\_Name: Name of the student.
     + Stu\_Department: Department to which the student belongs.
     + Stu\_Bio: Biometric parameter of a student
   * **Description**: Represents a student who registers for courses and is able to mark his attendance status during attendance tracking for an enrolled course.
4. **COURSES**
   * **Attributes**:
     + Course\_Code (Primary Key): Unique identifier for each course.
     + Course\_Name: Name of the course.
     + Department: Department offering the course.
     + Timeslot: Scheduled time for the course.
   * **Description**: Represents courses that are taken by students and taught by lecturers.
5. **ATTENDANCE**
   * **Attributes**:
     + Attend\_Date: Date of the attendance record.
     + Semester: Academic semester for the attendance.
   * **Description**: A weak entity representing attendance records for students in various courses, according to their respective semester

*Relationships and Cardinalities*

1. **LECTURER - Register - ADMINISTRATOR**
   * **Description**: A lecturer is registered by an administrator.
   * **Cardinality**: One-to-Many
     + One administrator registers many lecturers, but each lecturer is registered by one administrator.
2. **ADMINISTRATOR - Enroll - STUDENT**
   * **Description**: An administrator enrolls students into the system.
   * **Cardinality**: One-to-Many
     + One administrator enrolls many students, but each student is enrolled by one administrator.
3. **STUDENT - Takes - COURSES**
   * **Description**: A student takes multiple courses, and each course can be taken by multiple students.
   * **Cardinality**: Many-to-Many
     + This relationship is typically implemented using a join table that includes foreign keys from both entities.
4. **LECTURER - Record - ATTENDANCE**
   * **Description**: A lecturer records attendance for students in courses.
   * **Cardinality**: One-to-Many
     + One lecturer records attendance for many students, but each attendance record is made by one lecturer.
5. **STUDENT - Mark - ATTENDANCE**
   * **Description**: Attendance is marked for each student.
   * **Cardinality**: One-to-Many
     + One student has many attendance records, but each attendance record belongs to one student.
6. **COURSES - Course Attendance - ATTENDANCE**
   * **Description**: Attendance is tracked for each course.
   * **Cardinality**: One-to-Many
     + One course has many attendance records, but each attendance record belongs to one course.
7. **ADMINISTRATOR - Create - COURSES**
   * **Description**: An administrator creates courses.
   * **Cardinality**: One-to-Many
     + One administrator creates many courses, but each course is created by one administrator.

*Constraints*

1. **Primary Key Constraints**: Each entity has a primary key that uniquely identifies each record within the entity (e.g., Lecturer\_ID for LECTURER, Admin\_ID for ADMINISTRATOR, Stu\_Matricule for STUDENT, Course\_Code for COURSES, Attend\_Date for ATTENDANCE).
2. **Foreign Key Constraints**: These are implied in the relationships, ensuring referential integrity between entities (e.g., Stu\_Matricule in ATTENDANCE referencing Stu\_Matricule in STUDENT).
3. **Not Null Constraints**: Essential attributes such as Lecturer\_Name, Stu\_Name, and Course\_Name cannot be null to ensure completeness of data.

## LOGICAL DIAGRAM

In the logical data model, by offering a thorough representation of the data at a logical level, the logical data model expands on the conceptual model. It outlines the tables, columns, connections, and constraints that make up the[data structure.](https://www.geeksforgeeks.org/data-structures/) Although logical data models are not dependent on any particular database management system (DBMS), they are more similar to how data would be implemented in a database. The physical design of databases is based on this idea.

### RELATIONAL SCHEMA

A relational schema is a blueprint for how a database is structured. It defines the tables (relations) that make up the database and specifies the attributes (columns) of each table, the data types for each attribute, and the relationships between tables. A relational schema ensures that the database is organized in a way that supports efficient storage, retrieval, and manipulation of data.

Step-by-Step Guide

1. *Identify Entities*

Identify all entities in the ER diagram. Each entity will become a table in the relational schema.

**Example:** From your ER diagram, entities include:

* Lecturer
* Student
* Courses
* Attendance
* Administrator

#### Identify Attributes

For each entity, identify its attributes. Attributes will become columns in the corresponding table.

**Example:**

* Lecturer: Lecturer\_ID, Lecturer\_Name, Lecturer\_Email, Lecturer\_Bio
* Student: Stu\_Matricule, Stu\_Name, Stu\_Department
* Courses: Course\_Code, Course\_Name, Department, Timeslot
* Attendance: Attend\_Date, Semester
* Administrator: Admin\_ID, Admin\_Name, Admin\_Token

#### Determine Primary Keys

For each table, identify the primary key (PK). This is the attribute or combination of attributes that uniquely identifies each record in the table.

**Example:**

* Lecturer: Lecturer\_ID (PK)
* Student: Stu\_Matricule (PK)
* Courses: Course\_Code (PK)
* Attendance: weak entity
* Administrator: Admin\_ID (PK)

#### Identify Relationships

Determine the relationships between entities and represent them using foreign keys (FK). Relationships include one-to-one (1:1), one-to-many and many-to-many

#### Create Junction Tables for Many-to-Many Relationships

For many-to-many relationships, create a junction table that includes the primary keys from both related tables.

**Example:**

* **Takes** (Junction Table)
  + Stu\_Matricule (FK)
  + Course\_Code (FK)
  + PK: (Stu\_Matricule, Course\_Code)

#### Normalize the Schema

Ensure that the schema is normalized to eliminate redundancy and improve data integrity. Apply normalization rules such as 1NF, 2NF, and 3NF.

**Example:** Check each table for atomicity, remove partial dependencies, and eliminate transitive dependencies.

#### . Define the Final Schema

List all the tables with their attributes, primary keys, and foreign keys.

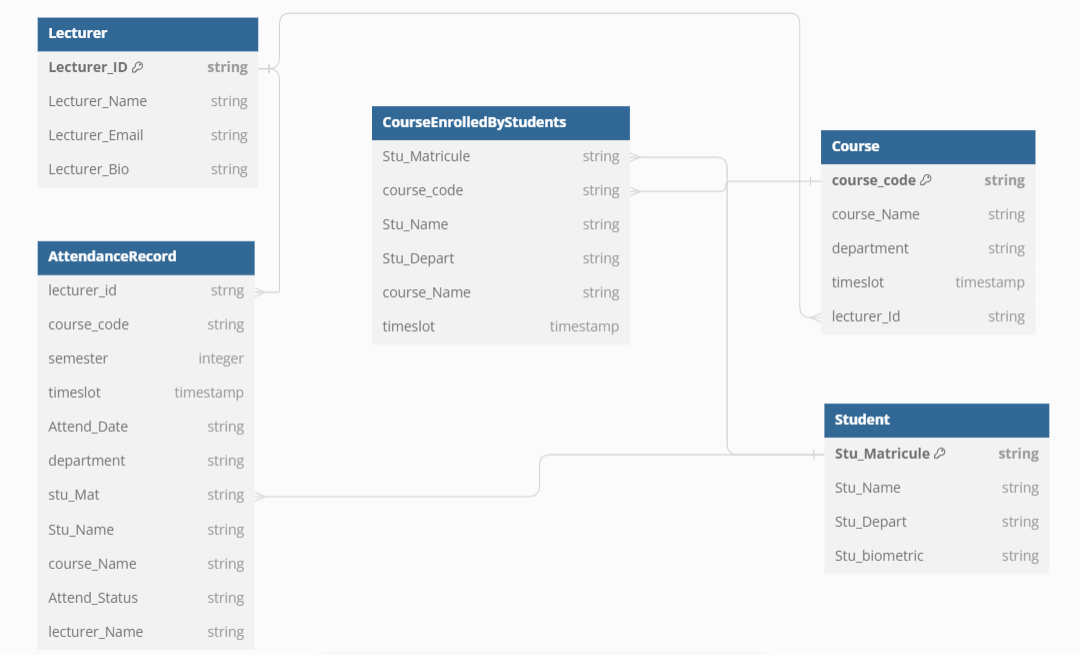
**Lecturer** (lecturer\_id, lecturer\_Name, lecturer\_email, lecturer\_Bio, Admin\_Id)

**Student** (Stu\_mat, Stu\_Name, Stu\_Depart)

**CourseEnrolledByStudents** (Stu\_Mat, course\_code, Stu\_Name, Stu\_depart, Course\_Name, timeslot)

**Courses** (course\_code, Course\_Name, department, timeslot, Admin\_id)

**AttendanceRecord** (lecturer\_id, course\_code, semester, timeslot, Attend\_Date, department, stu\_Mat, Stu\_Name, course\_Name, Attend\_Status, lecturer\_Name)



### 3.2.2 DRAWBACKS FACED WITH SQL DBMS

While SQL (Structured Query Language) Database Management Systems (DBMS) are widely used and offer numerous benefits, they also come with certain drawbacks. Here are some of the main challenges and limitations associated with SQL DBMS:

1. **Scalability Issues:**
   1. **Vertical Scaling:** SQL databases traditionally scale vertically by adding more resources (CPU, RAM) to a single server. This approach has limits and can become expensive and less efficient.
   2. **Horizontal Scaling:** Scaling horizontally (adding more servers) is more challenging with SQL databases compared to NoSQL databases, which are designed to handle such scaling more gracefully.
2. **Complexity in Handling Large Volumes of Data:**
   1. **Performance Degradation:** As the volume of data grows, SQL queries can become slower, and the performance of the database can degrade. Index management and query optimization become critical and complex tasks.
   2. **Big Data:** For very large datasets, SQL databases might not be as efficient as some NoSQL databases designed to handle big data more effectively.
3. **Schema Rigidity:**
   1. **Fixed Schema:** SQL databases require a predefined schema, making them less flexible in handling changing data structures. Altering the schema can be time-consuming and disruptive.
   2. **Evolving Requirements:** In dynamic environments where data models frequently evolve, the rigid schema of SQL databases can be a significant limitation.
4. **Complexity in Distributed Systems:**
   1. **Replication and Sharding:** While possible, implementing replication and sharding in SQL databases can be complex and requires careful planning and management.
   2. **Consistency Issues:** Maintaining consistency across distributed SQL databases can be challenging, especially in high-availability and disaster recovery scenarios.
5. **Cost:**
   1. **Licensing:** Commercial SQL databases like Oracle, SQL Server, and others can be expensive in terms of licensing fees.
   2. **Infrastructure:** The cost of infrastructure required for high-performance SQL databases (e.g., powerful servers, SSDs) can be high.
6. **Handling Unstructured Data:**
   1. **Limited Flexibility:** SQL databases are less efficient at handling unstructured or semi-structured data (e.g., JSON, XML) compared to NoSQL databases, which are designed with these data types in mind.
   2. **Complex Queries:** Complex queries involving unstructured data can be cumbersome and less performant in SQL databases.
7. **Concurrency and Transaction Handling:**
   1. **Locking and Blocking:** High levels of concurrent transactions can lead to locking and blocking issues, impacting performance.
   2. **Deadlocks:** Complex transactions can result in deadlocks, requiring sophisticated deadlock detection and resolution mechanisms.
8. **Maintenance and Tuning:**
   1. **Regular Maintenance:** SQL databases require regular maintenance tasks such as indexing, backups, and performance tuning.
   2. **Skilled Personnel:** Effective management of SQL databases often requires skilled DBAs (Database Administrators), which can be a significant cost for organizations.

# NO SQL DATABASE

For our application, we decided to use a **NoSQL** database.

## 4.1 WHAT IS NOSQL DATABASE

NoSQL is a type of database management system (DBMS) that is designed to handle and store large volumes of unstructured and semi-structured data. Unlike traditional relational databases that use tables with pre-defined schemas to store data, NoSQL databases use flexible data models that can adapt to changes in data structures and are capable of scaling horizontally to handle growing amounts of data.

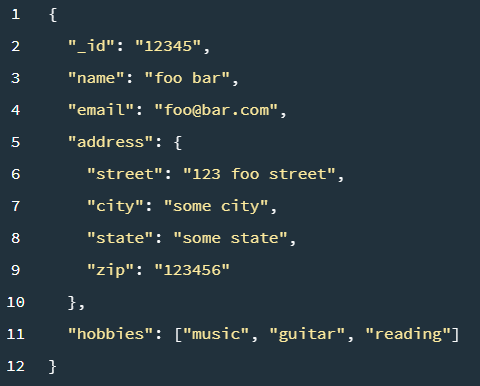
The term NoSQL originally referred to “non-SQL” or “non-relational” databases, but the term has since evolved to mean “not only SQL,” as NoSQL databases have expanded to include a wide range of different database architectures and data models.

## 4.2 CLASSIFICATION OF NOSQL DATABASES

**Document-oriented database**

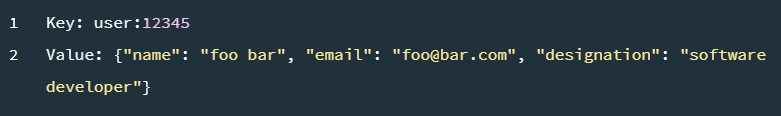
A document-oriented database stores data in documents similar to JSON (JavaScript Object Notation) objects. Each document contains pairs of fields and values. The values can typically be a variety of types, including things like strings, numbers, booleans, arrays, or even other objects. A document database offers a flexible data model, much suited for semi-structured and typically unstructured data sets. They also support nested structures, making it easy to represent complex relationships or hierarchical data.

Examples of document databases are [MongoDB](https://www.mongodb.com/) and Couchbase. A typical document will look like the following:



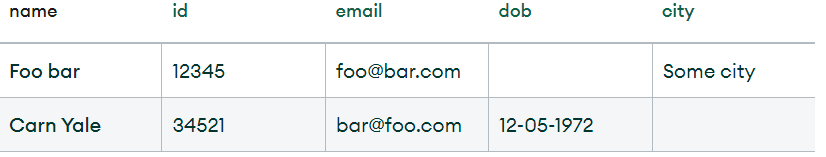
**Key-value databases**

A key-value store is a simpler type of database where each item contains keys and values. Each key is unique and associated with a single value. They are used for caching and session management and provide high performance in reads and writes because they tend to store things in memory. Examples are Amazon DynamoDB and Redis. A simple view of data stored in a key-value database is given below:



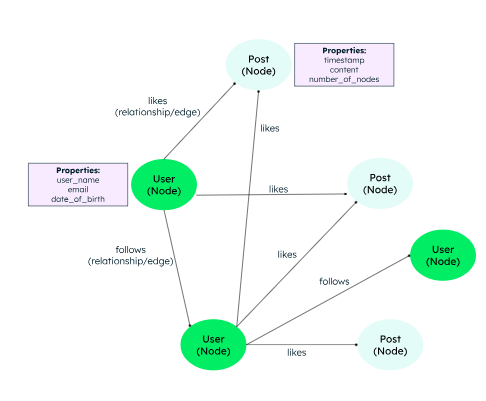
###### **Wide-column stores**

Wide-column stores store data in tables, rows, and dynamic columns. The data is stored in tables. However, unlike traditional SQL databases, wide-column stores are flexible, where different rows can have different sets of columns. These databases can employ column compression techniques to reduce the storage space and enhance performance. The wide rows and columns enable efficient retrieval of sparse and wide data. Some examples of wide-column stores are Apache Cassandra and HBase. A typical example of how data is stored in a wide-column is as follows:



###### **Graph databases**

A graph database stores data in the form of nodes and edges. Nodes typically store information about people, places, and things (like nouns), while edges store information about the relationships between the nodes. They work well for highly connected data, where the relationships or patterns may not be very obvious initially. Examples of graph databases are Neo4J and Amazon Neptune. MongoDB also [provides graph traversal capabilities](https://www.mongodb.com/databases/mongodb-graph-database) using the graphLookup stage of the aggregation pipeline. Below is an example of how data is stored:



###### **Multi-model databases**

Multi-model databases support more than one type of NoSQL data model so that developers can choose based on their application requirements. These databases have a unified database engine that can handle multiple data models within a database instance. Examples are CosmosDB and ArangoDB.

## 4.3 WHY DID WE CHOOSE NOSQL DATABASE

When designing the database for a Biometric Student Attendance App, the choice of a NoSQL database offers several advantages that align with the specific requirements and challenges of the application. Some key reasons for opting for a NoSQL database:

**Scalability**

NoSQL databases are inherently designed to scale horizontally, which is crucial for applications expected to handle a growing number of students and attendance records. This means that as the number of users increases, the database can distribute the load across multiple servers without significant performance degradation.

**Flexibility**

NoSQL databases provide a flexible schema design, allowing for easy modifications as the application evolves. This is particularly important for an attendance app where the data structure might need to adapt to new features or changes in requirements without extensive reconfiguration.

**Performance**

NoSQL databases are optimized for read and write operations, which is essential for an attendance system that needs to quickly process fingerprint data and update attendance records in real-time. This ensures that the system remains responsive and efficient even under heavy load.

**Handling Unstructured Data**

Biometric data, such as fingerprints, can vary significantly in structure and size. NoSQL databases, particularly document-oriented ones like MongoDB, are well-suited for storing and querying such unstructured data efficiently.

**High Availability and Fault Tolerance**

NoSQL databases often include built-in support for replication and distribution, ensuring high availability and fault tolerance. This is critical for an attendance app where data loss or downtime could have significant operational impacts.

**Real-Time Data Processing**

For an attendance app that needs to process and update data in real-time, NoSQL databases provide the necessary performance and flexibility. They support real-time data synchronization, ensuring that attendance records are immediately updated and available for reporting.

**Cost-Effective Scaling**

Scaling a NoSQL database can be more cost-effective compared to traditional relational databases. NoSQL solutions often run on clusters of commodity hardware, reducing the cost of scaling infrastructure to meet growing demands.

# FIREBASE/ FIRESTORE

For our NoSQL database, we use **Firestore**, a NoSQL database offered by Firebase

## 5.1 WHAT IS FIRESTORE

Cloud Firestore is a flexible, scalable database for mobile, web, and server development from Firebase and Google Cloud. Like Firebase Real-time Database, it keeps your data in sync across client apps through real-time listeners and offers offline support for mobile and web so you can build responsive apps that work regardless of network latency or Internet connectivity. Cloud Firestore also offers seamless integration with other Firebase and Google Cloud products, including Cloud Functions.

### 5.2 WHY WE CHOSE FIRESTORE DATABASE FOR A FINGERPRINT STUDENT ATTENDANCE APPLICATION

The decision to use Firestore, a NoSQL database offered by Firebase, for a Biometric Student Attendance App is driven by several key advantages that align perfectly with the specific requirements of the application. The reasons for choosing Firestore include:

**Real-Time Data Synchronization**

Firestore excels in providing real-time data synchronization across multiple clients. This is essential for a student attendance app where updates to attendance records need to be instantly reflected across different devices, ensuring that teachers, administrators, and students have up-to-date information at all times.

**Scalability and Performance**

Firestore is designed to handle large volumes of data with ease. It offers automatic scaling to manage increasing numbers of students and attendance records without compromising performance. This is crucial for educational institutions that experience fluctuating attendance loads and need a database that can seamlessly adjust to these changes.

**Flexible Data Structure**

Firestore uses a flexible, document-oriented data model. This allows for easy storage of varied data types, such as student profiles, fingerprint templates, and attendance logs, without the need for a rigid schema. The ability to quickly adapt the database structure to new requirements makes it ideal for an evolving attendance system.

**Offline Capabilities**

Firestore supports offline data access and synchronization. This feature ensures that the app remains functional even when there is no internet connectivity. Students and teachers can continue to record attendance, and the data will sync automatically once the connection is restored.

**Security**

Firestore offers robust security features through Firebase Authentication and Firestore Security Rules. This ensures that sensitive biometric data and personal information are securely stored and accessed only by authorized users. The security rules can be fine-tuned to meet the specific needs of the attendance app, providing granular control over data access.

**Ease of Integration**

Firestore integrates seamlessly with other Firebase services, such as Firebase Authentication and Firebase Cloud Messaging. This integration simplifies the implementation of user authentication, push notifications, and other essential features, enhancing the overall functionality of the attendance app.

**Cloud Firestore’s Global Replication**

Firestore’s data is automatically replicated across multiple data centers, providing high availability and disaster recovery capabilities. This ensures that the attendance data is always accessible and protected against data loss, which is critical for maintaining continuous operation of the attendance app.

**Cost-Effective**

Firestore offers a pay-as-you-go pricing model, which can be cost-effective for educational institutions. The ability to start small and scale as needed helps manage costs efficiently while providing the necessary infrastructure to support the app's growth.

## 5.3 CLOUD FIRESTORE DATA MODEL

Cloud Firestore is a NoSQL, document-oriented database. Unlike a SQL database, there are no tables or rows. Instead, you store data in *documents*, which are organized into *collections*.

Each *document* contains a set of key-value pairs. Cloud Firestore is optimized for storing large collections of small documents.

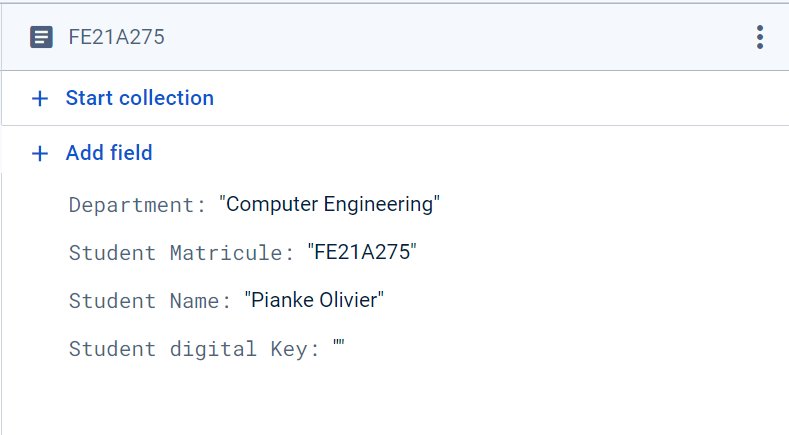
All documents must be stored in collections. Documents can contain *subcollections* and nested objects, both of which can include primitive fields like strings or complex objects like lists.

Collections and documents are created implicitly in Cloud Firestore. Simply assign data to a document within a collection. If either the collection or document does not exist, Cloud Firestore creates it.

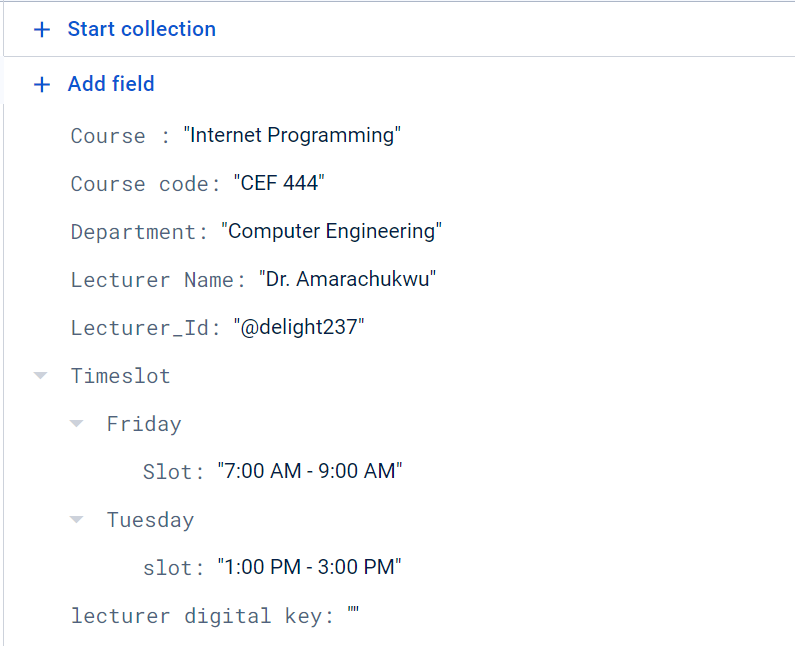
### 5.3.1 Documents

In Cloud Firestore, the unit of storage is the document. A document is a lightweight record that contains fields, which map to values. Each document is identified by a name.

A document representing a user alovelace might look like this:

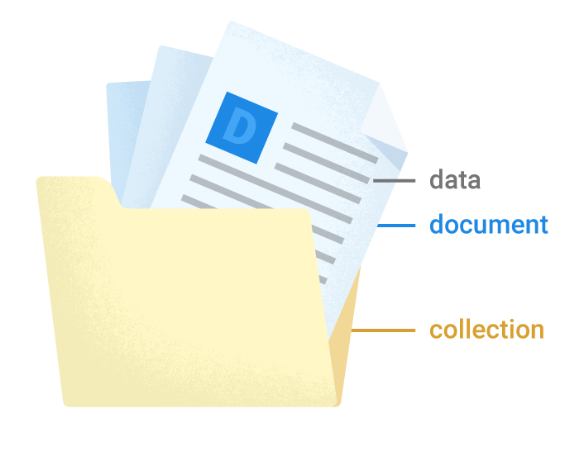


Complex, nested objects in a document are called maps. For example, you could structure the user's name from the example above with a map, like this:



You may notice that documents look a lot like JSON. In fact, they basically are. There are some differences (for example, documents support extra data types and are limited in size to 1 MB), but in general, you can treat documents as lightweight JSON records.

### 5.3.2 Collections



Documents live in collections, which are simply containers for documents. Cloud Firestore is schemaless, so you have complete freedom over what fields you put in each document and what data types you store in those fields. Documents within the same collection can all contain different fields or store different types of data in those fields.

However, it's a good idea to use the same fields and data types across multiple documents, so that you can query the documents more easily.

A collection contains documents and nothing else. It can't directly contain raw fields with values, and it can't contain other collections.

The names of documents within a collection are unique. You can provide your own keys, such as user IDs, or you can let Cloud Firestore create random IDs for you automatically.

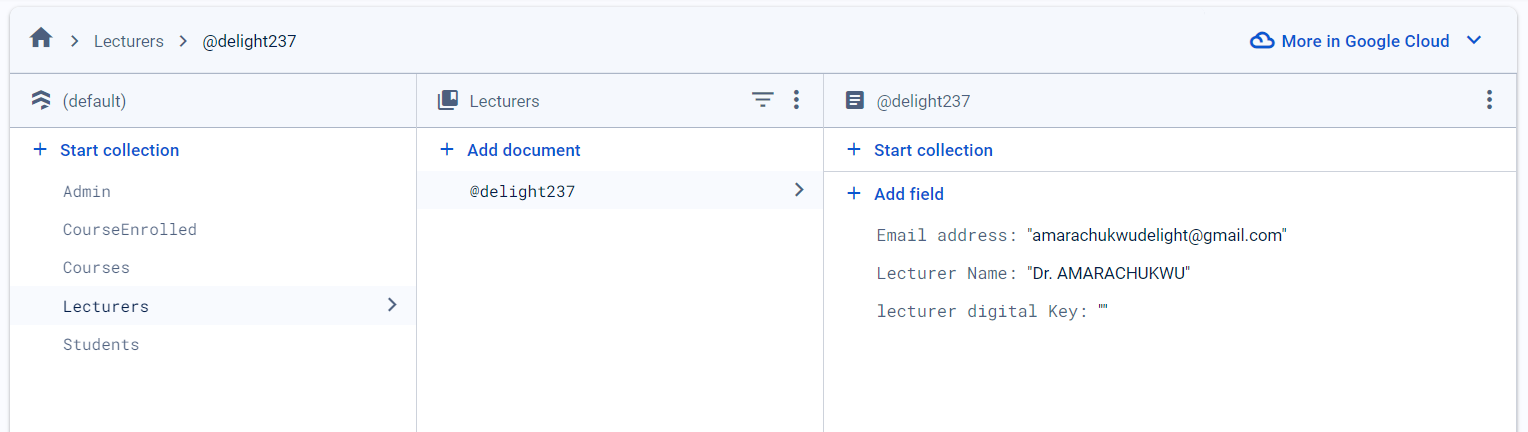
You do not need to "create" or "delete" collections. After you create the first document in a collection, the collection exists. If you delete all of the documents in a collection, it no longer exists.

## 5.4 STRUCTURE OF BIOTRACK DATABASE DOCUMENTS IN FIRESTORE

Our Database consist of six (6) collections, each collection containing information described as follows

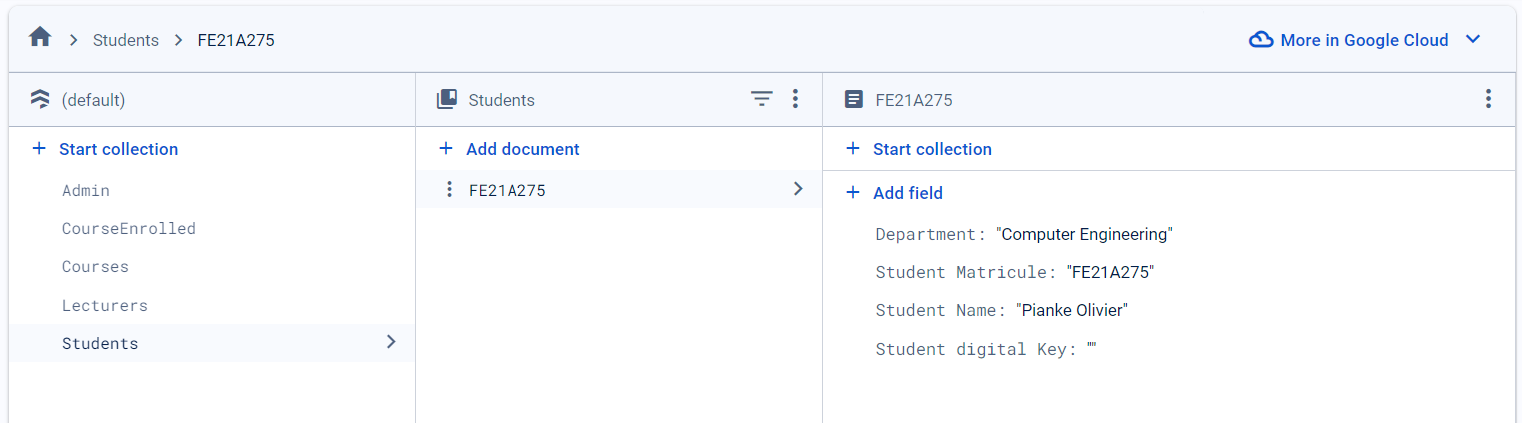
## Collection: Lecturers

This collection includes documents for each lecturer with details such as lecturer ID, name, email, and biometric key.



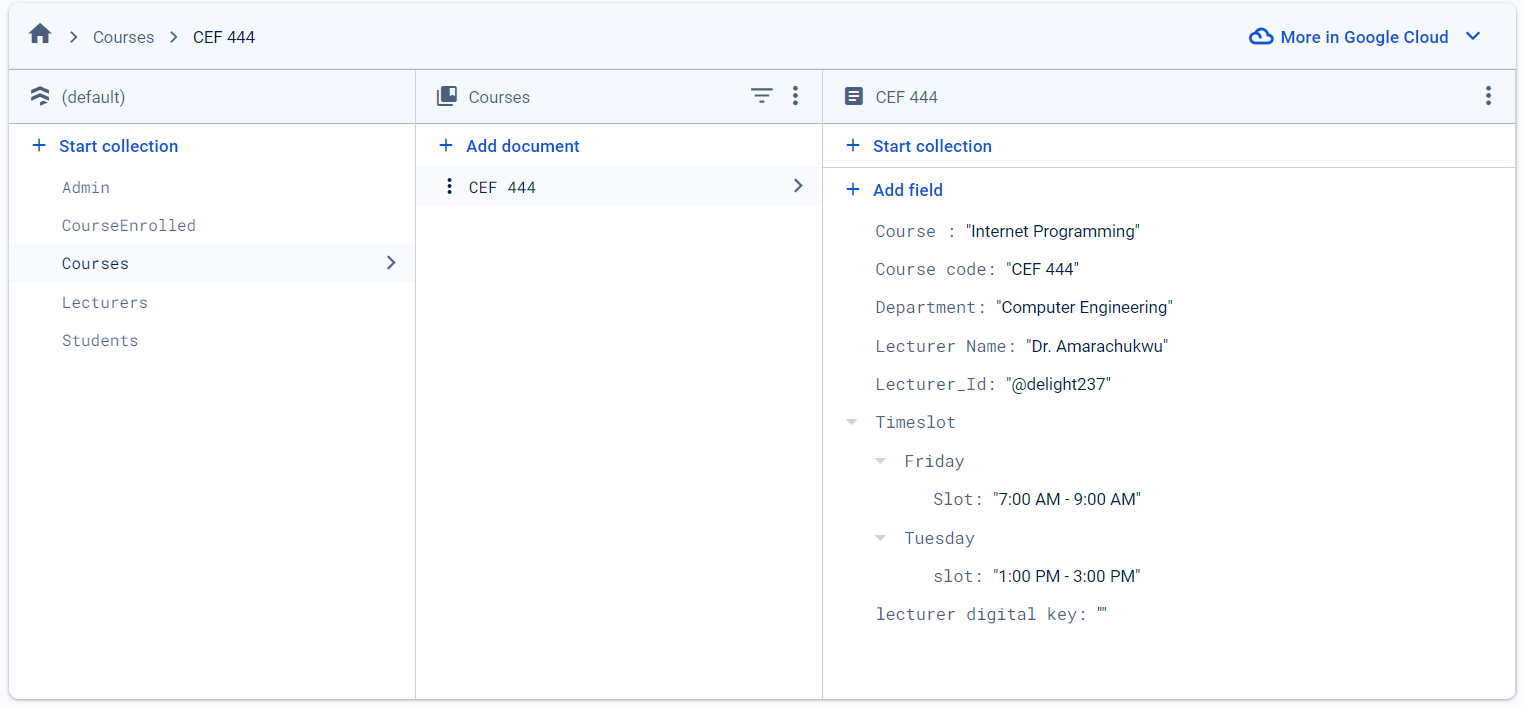
## Collection: Students

This collection includes documents for each Student with details such as Student matricule, name, department and biometric key.



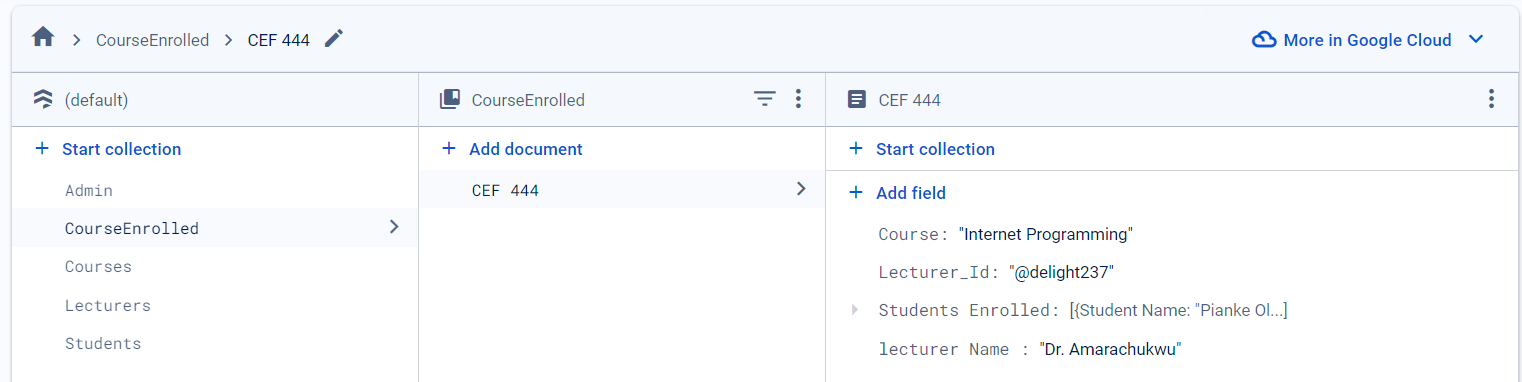
## Collection: Course

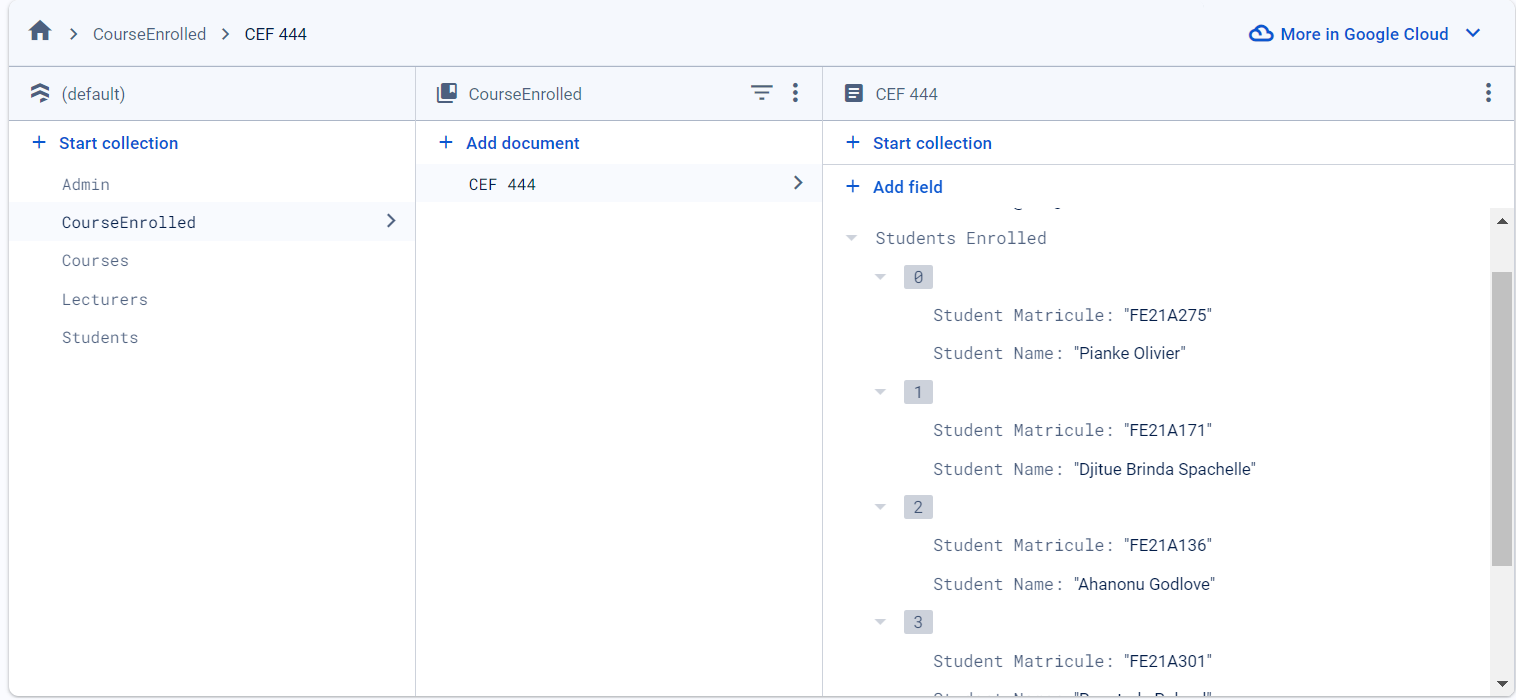
This collection is inferred to store the courses available in the system. Each document represents a unique course with associated details.



## Collection: CourseEnrolled

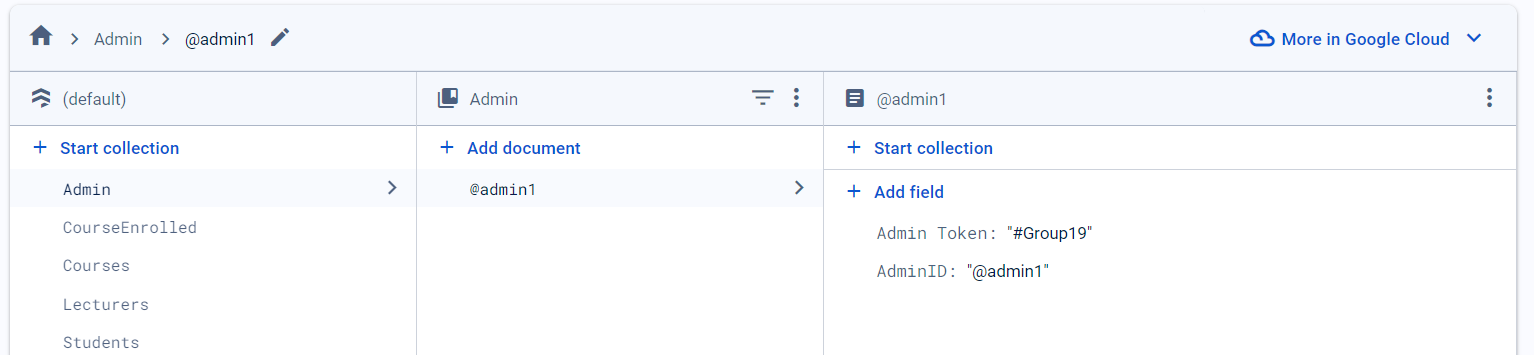
This collection contains documents representing each course a student is enrolled in. Each document within this collection represents a unique course.





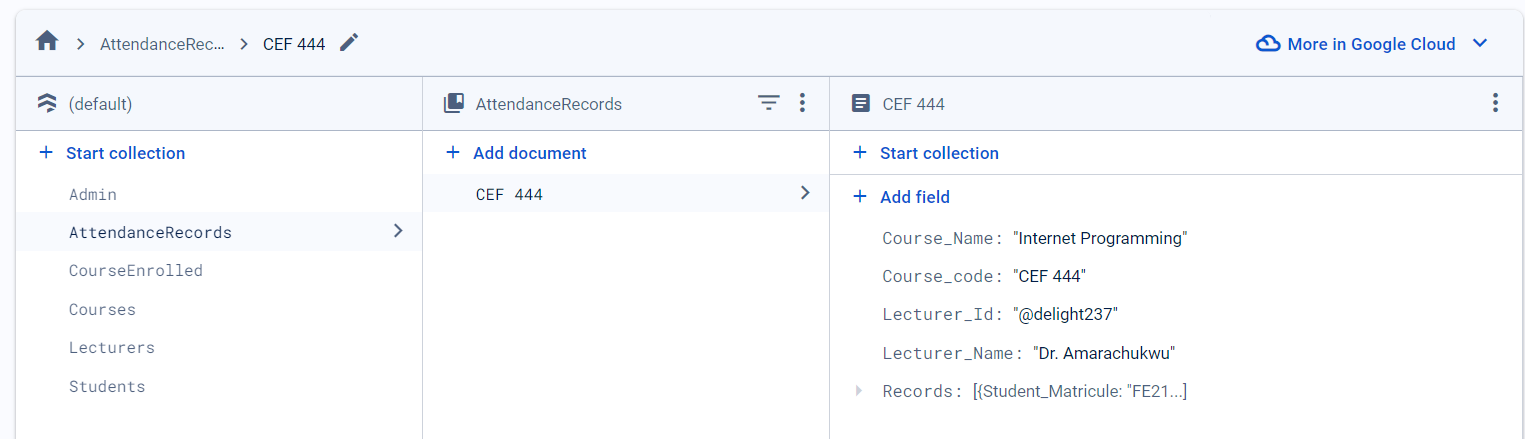
## Collection: Admin

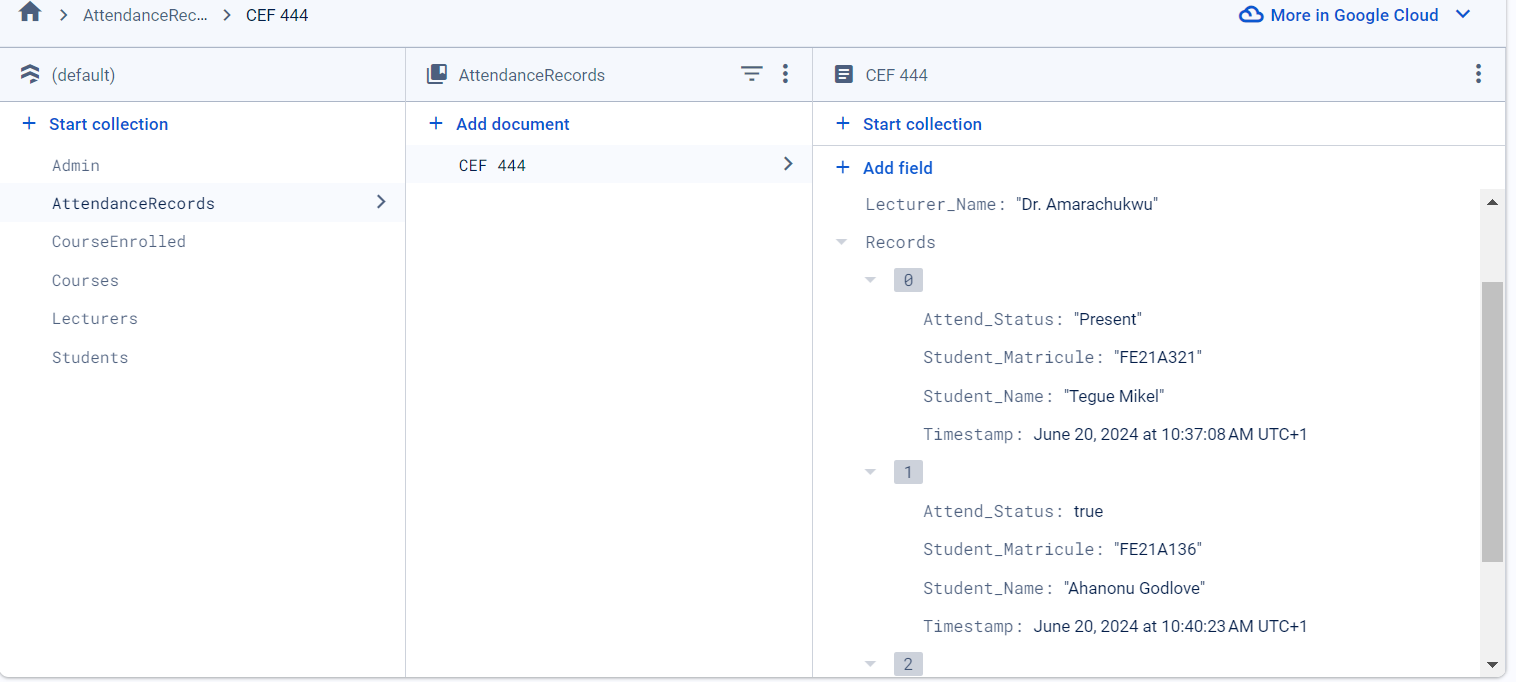
This collection contains the document holding the admin credentials along with the admin token for authentication into the system as super user



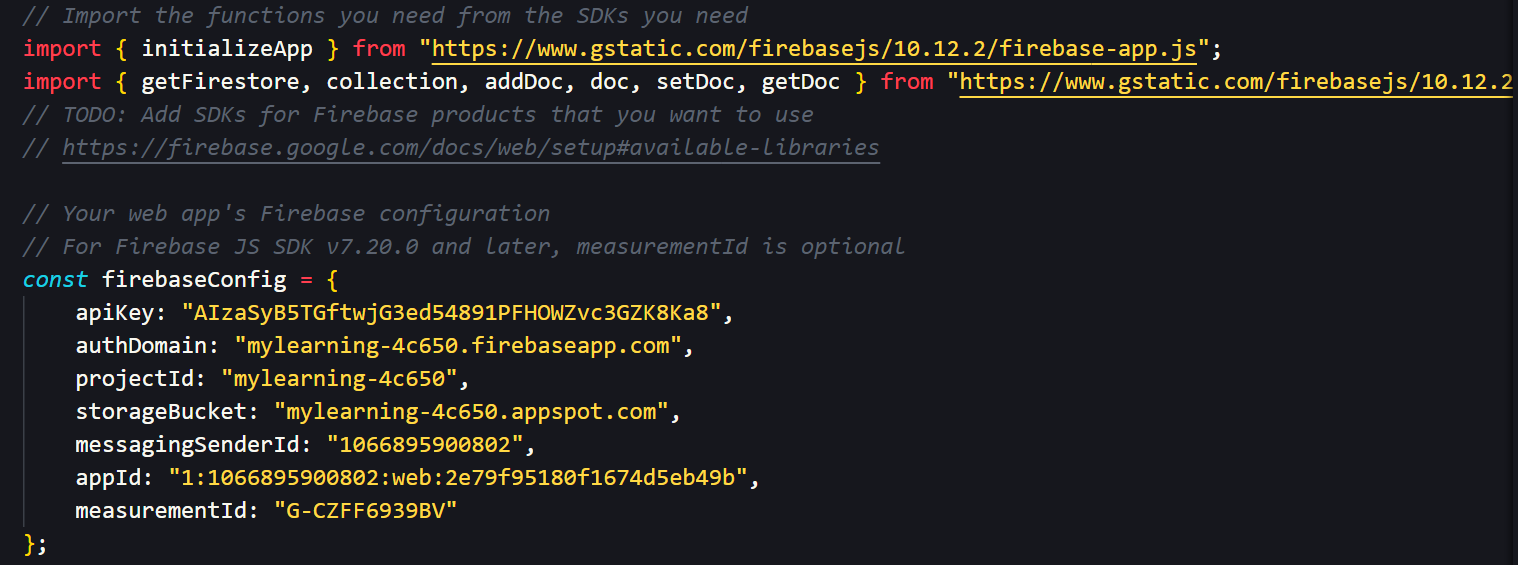
## Collection: Attendance Records

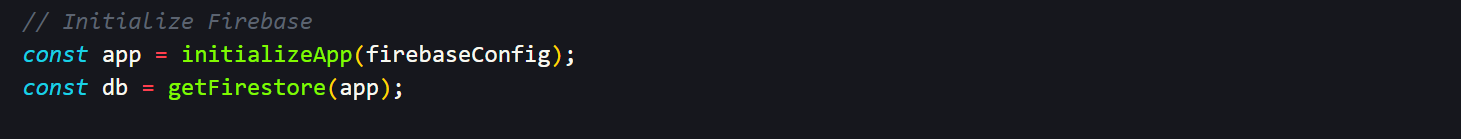
This collection contains records of attendance tracking stored as documents according to the respective ongoing courses



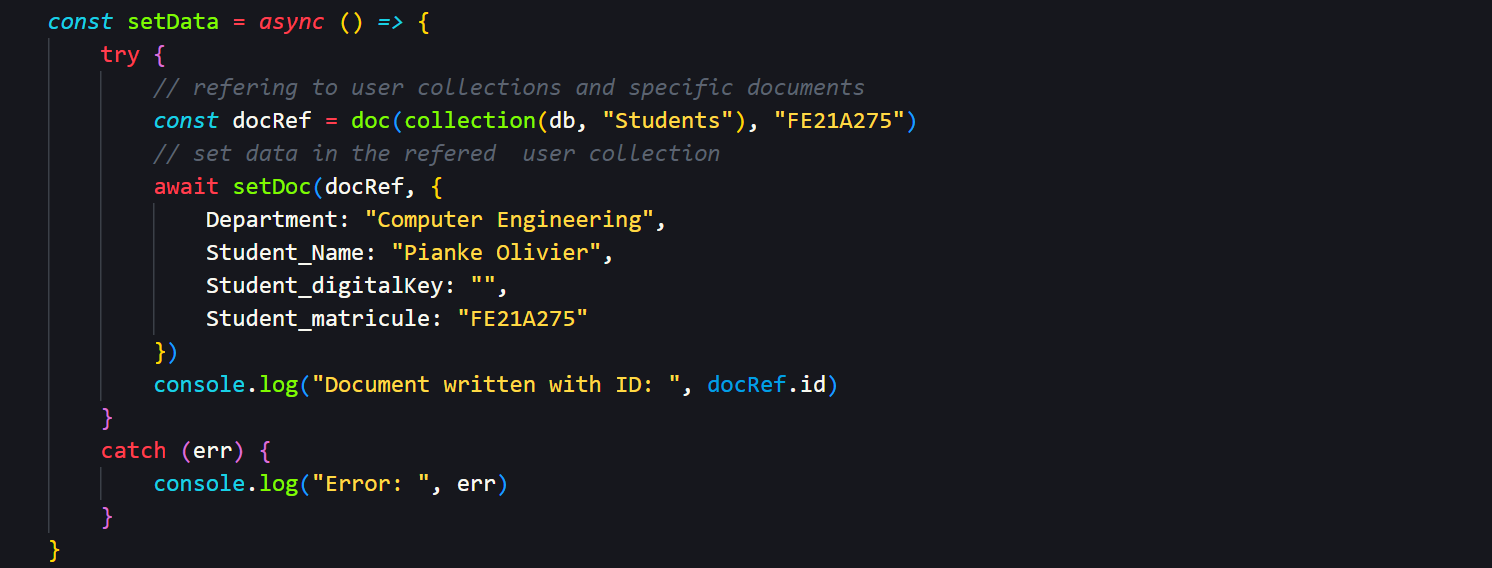


## 5.5 SCRIPTS USED FOR FIREBASE/FIRESTORE CONFIGURATION AND INITIALIZATION

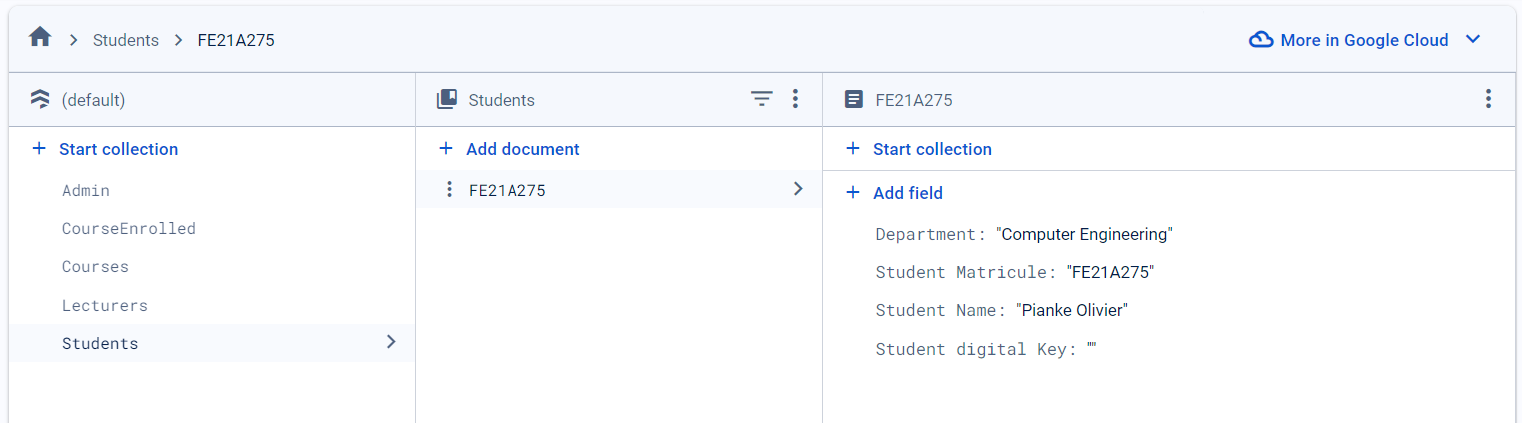




## 5.6 TEST FUNCTION USED TO QUERY THE DATABASE, CREATE AND POPULATE SAMPLE DOCUMENT DATA FOR THE STUDENT’S COLLECTION



**Output**



# CONCLUSION

The implementation of a Biometric Student’s Attendance App, supported by a well-designed NoSQL database, offers a solution for managing student attendance in educational institutions. Through careful database design, leveraging the flexibility and scalability of NoSQL databases like Firestore, the system can handle the dynamic and evolving requirements of modern educational environments.

The database design process emphasized key principles such as maintaining data consistency, reducing redundancy, ensuring faster data retrieval, and enforcing stringent security measures. These principles were critical in establishing a robust foundation for the application, capable of handling large volumes of biometric data and attendance records with high performance and reliability.

The choice of Firestore as the NoSQL database was driven by its real-time data synchronization, scalability, flexible data structure, offline capabilities, robust security features, ease of integration with other Firebase services, and cost-effectiveness. These attributes ensure that the attendance system remains responsive, adaptable, and secure, providing a seamless user experience for administrators, lecturers, and students.

# REFERENCES

* + <https://www.geeksforgeeks.org/database-design-in-dbms/> visited on June 19, 2024
  + <https://www.geeksforgeeks.org/introduction-to-nosql/> visited on June 19, 2024
  + <https://www.mongodb.com/resources/basics/databases/nosql-explained> visited on June 19, 2024
* <https://www.researchgate.net/publication/277890728_NoSQL_Databases_and_Data_Modeling_Techniques_for_a_Document-oriented_NoSQL_Database>

Author [**Robert T Mason**](https://www.researchgate.net/profile/Robert-Mason-21?_tp=eyJjb250ZXh0Ijp7ImZpcnN0UGFnZSI6InB1YmxpY2F0aW9uIiwicGFnZSI6InB1YmxpY2F0aW9uIn19)Regis University, Denver, CO, USA

* <https://www.geeksforgeeks.org/data-modeling-a-comprehensive-guide-for-analysts/> visited on June 20,2024