

UNIVERSITY OF BUEA  
P.O Box 63,  
Buea\_South West Region  
Cameroon  
Tel: (+237) 674354327  
Fax: (+237) 3332 22 72



**REPUBLIC OF CAMEROON**

PEACE – WORK - FATHERLAND

**FACULTY OF ENGINEERING AND TECHNOLOGY**  
**COMPUTER ENGINEERING DEPARTMENT**  
**CEF 440: INTERNET PROGRAMMING (J2EE) AND MOBILE PROGRAMMING**

**TASK 6:**  
**DATABASE DESIGN AND IMPLEMENTATION**  
**OF A BIOMETRIC STUDENT ATTENDANCE**  
**APPLICATION**

SN	NAMES	MATRICULE
1	NGULEFAC JERRY MBUOH	FE21A265
2	NEBA PRINCEWILL AMBE	FE21A251
3	NKENGBEZA DERICK	FE21A277
4	KAH JOSPEN NGUM	FE21A207
5	NYOCHENBENG ENZO NKENGAFACK	FE21A293

COURSE INSTRUCTOR: NKEMENI VALERY

Academic year 2023/2024

Presented by Group 11

## Contents

1. INTRODUCTION .....	4
1.1. Introduction to the project.....	4
1.2. Objectives and scope of the project .....	4
1.3. Motivation for choosing NoSQL database.....	4
2. LITERATURE REVIEW .....	5
2.1. Review of relevant literature on NoSQL databases .....	5
2.2. Comparison with traditional (SQL) databases .....	6
2.3. Analysis of different types of NoSQL databases (document-oriented, key-value stores, column-family stores, graph databases).....	8
2.4. Advantages and disadvantages of using NoSQL databases .....	11
<b>2.4.1. Advantages</b> .....	11
<b>2.4.2. Disadvantages</b> .....	12
3. METHODOLOGY .....	13
3.1. Overview .....	13
3.2. Justification for choosing MongoDB.....	13
3.3. Designing the Database Schema.....	14
3.4. Data Modelling .....	14
4. DATABASE DESIGN.....	15
4.1. Overview of the designed schema .....	15
4.2. Conceptual database Design .....	15
<b>4.2.1. Entities and Attributes:</b> .....	15
4.3. Logical and Physical Database Design (collections, documents, tables, or key structures) .	16
4.4. How data is organized to support specific use cases and query patterns .....	18
5. IMPLEMENTATION.....	19
5.1. Description of the implementation process.....	20
5.2. Tools and technologies used .....	22
5.3. Challenges encountered and how they were addressed .....	23
5.4. Performance considerations and optimizations made .....	25
6. TESTING AND EVALUATION .....	25
6.1. Overview of testing Strategies used.....	26
6.2. Test cases and scenarios tested (e.g., performance benchmarks, scalability tests) .....	26
6.3. Functional Testing .....	26

6.4.	Performance Testing .....	27
6.5.	Security Testing .....	27
6.6.	Results and Discussions .....	28
7.	CONCLUSION .....	28
7.1.	Summary of key findings .....	28
7.2.	Future considerations and improvements.....	28
8.	APPENDICES .....	28

## 1. INTRODUCTION

### 1.1. Introduction to the project

This project for a Biometric student attendance system is aim at facilitating the attendance taking of students for both the students and instructors. The requirement gathering, Analysis, Design are all important but what should be done after that? The design and implementation of a database for such a system and other systems in general is very important. It is the database that actually holds all of the information needed to actually make/keep the system up and running. So, the next question to ask ourself is which type of database to use for our system.

### 1.2. Objectives and scope of the project

The scope of Database design and implementation encompasses:

- ❖ Going back and analyze your project goals and application requirements to actually know which type of data is suitable for storing which piece of information.
- ❖ How data or the entities in the application actually relate to each other.
- ❖ Which Database Management System (DBMS) is best fit for the application, is a Relational DBMS or not?
- ❖ Design of the data schemas
- ❖ Implementation of the database management
- ❖ Integrating with RESTFul API
- ❖ Testing and evaluation

### 1.3. Motivation for choosing NoSQL database

- ❖ **Scalability**
  - Horizontal Scalability: NoSQL databases are designed to scale out by adding more servers, which is essential for handling large volume of users.
  - **Flexible Schema:** As the system evolves, the database schema can be easily modified without significant downtime or complex migrations, accommodating changes in data structure effortlessly.
- ❖ **High Performance**
  - **Low Latency:** NoSQL databases can deliver high read and write performance, which is critical for real-time data processing and immediate attendance updates.
  - **Optimized for Read/Write Operations:** They are often optimized for the types of read and write operations that are common in attendance tracking applications.

### ❖ **Availability and Reliability**

- **Distributed Architecture:** Many NoSQL databases are designed with a distributed architecture, providing high availability and fault tolerance. This ensures that the application remains operational even if some nodes fail.
- **Replication and Consistency:** Built-in replication features ensure data is consistently backed up and available across multiple nodes.

### ❖ **Real-time Data Processing**

- **Eventual Consistency Models:** These models allow for efficient real-time data processing and synchronization across distributed systems, which is beneficial for real-time attendance tracking.

### ❖ **Cost-Effectiveness**

- **Commodity Hardware:** NoSQL databases can run on commodity hardware, reducing the overall cost of deployment and maintenance.
- **Open-Source Options:** Many NoSQL databases are open source, offering a cost-effective solution without licensing fees.

### ❖ **Complex Queries and Analytics**

- **Indexing and Querying:** Advanced indexing and querying capabilities allow for efficient retrieval of specific biometric records and attendance logs.
- **Analytics Integration:** NoSQL databases can integrate with data analytics tools to provide insights into attendance patterns, student behavior, and system performance.

### ❖ **Security**

- **Flexible Security Models:** NoSQL databases offer various security features, such as encryption at rest and in transit, fine-grained access controls, and integration with identity management systems, which are crucial for protecting sensitive data.

## 2. LITERATURE REVIEW

### 2.1. Review of relevant literature on NoSQL databases

The term NoSQL database is typically used to refer to any non-relational database. Some say NoSQL stands for non-SQL while others say it stands for not only SQL. Either be the case, most agree that NoSQL databases store data in more natural and flexible way.

NoSQL, as opposed to SQL is a database management approach whereas SQL is just a query language, similar to the query languages of NoSQL databases. It was developed in the beginning as a response to web data, the need for processing unstructured data, and the need for faster processing.

## 2.2. Comparison with traditional (SQL) databases

There are a variety of differences between relational database management systems and non-relational databases. One of the key differences is the way data is modelled in the database. Some key differences of each feature are listed below:

### ➤ Data Model

#### ❖ *SQL Databases:*

- **Relational Model:** Data is organized into tables (relations) consisting of rows and columns.
- **Fixed Schema:** Requires a predefined schema with enforced data types and relationships.
- **ACID Transactions:** Ensures atomicity, consistency, isolation, and durability.

#### ❖ *NoSQL Databases:*

- **Varied Models:** Includes document, key-value, column-family, and graph databases.
- **Flexible Schema:** Allows for dynamic schema design, suitable for unstructured and semi-structured data.
- **BASE Properties:** Emphasizes basically available, soft state, and eventual consistency.

### ➤ Scalability

#### ❖ *SQL Databases:*

- **Vertical Scalability:** Scaling up involves increasing the capacity of a single server (more CPU, RAM).
- **Horizontal Scalability:** More challenging; involves partitioning and sharding.

#### ❖ *NoSQL Databases:*

- **Horizontal Scalability:** Designed to scale out by adding more servers, distributing data across nodes efficiently.
- **Efficient Distribution:** Built-in support for data replication and distribution.

### ➤ Query Language

#### ❖ *SQL Databases:*

- **Structured Query Language (SQL):** Standard language for querying and managing data.
- **Join Operations:** Supports complex joins to combine data from multiple tables.

❖ *NoSQL Databases:*

- **Varied Query Languages:** Each NoSQL database may have its own language or API (e.g., MongoDB uses a JSON-like query language).
- **Limited Joins:** Typically, less support for joins; encourages denormalization.

➤ **Data Integrity and Consistency**

❖ *SQL Databases:*

- **Strong Consistency:** Ensures data integrity through ACID transactions.
- **Constraints:** Enforces foreign keys, unique constraints, and other relational constraints.

❖ *NoSQL Databases:*

- **Eventual Consistency:** Some NoSQL systems provide eventual consistency for better performance and availability.
- **Application-Level Integrity:** Relationships and constraints are often managed at the application level.

➤ **Performance**

❖ *SQL Databases:*

- **Optimized for Complex Queries:** Performs well with complex queries and transactional operations.
- **Overhead:** Schema enforcement and transactional consistency can introduce performance overhead.

❖ *NoSQL Databases:*

- **High Throughput:** Designed for high read/write throughput and low latency.
- **Trade-offs:** May sacrifice some consistency or complex querying capabilities for performance.

➤ **Use Cases**

❖ *SQL Databases:*

- **Structured Data:** Ideal for structured data with clear relationships.

- **Transactional Systems:** Suitable for applications requiring complex queries and transactions (e.g., financial systems).

❖ *NoSQL Databases:*

- **Unstructured Data:** Suitable for unstructured and semi-structured data (e.g., JSON documents).
- **Big Data and Real-Time Applications:** Ideal for applications requiring scalability and high performance (e.g., social media, IoT).

➤ **Examples**

- ❖ **SQL Databases:** MySQL, PostgreSQL, Oracle Database, Microsoft SQL Server.
- ❖ **NoSQL Databases:** MongoDB (document), Redis (key-value), Cassandra (column-family), Neo4j (graph).

### 2.3. Analysis of different types of NoSQL databases (document-oriented, key-value stores, column-family stores, graph databases)

➤ **Document-oriented database:**

A Document oriented database (for example MongoDB) stores data in documents similar to JSON 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. It offers a flexible data model much suited for semi-structured and typically unstructured data sets. They also support nested structures hence making it easy to represent complex relationships or hierarchical data. Below is an example of how data is stored in such database.

```
1  {
2    "_id": "12345",
3    "name": "foo bar",
4    "email": "foo@bar.com",
5    "address": {
6      "street": "123 foo street",
7      "city": "some city",
8      "state": "some state",
9      "zip": "123456"
10   },
11   "hobbies": ["music", "guitar", "reading"]
12 }
```



➤ **Key-value databases:**

This is a simpler type of NoSQL 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. An example of such database is Amazon DynamoDB. Below is an example of how data is stored in such database.

```
1 Key: user:12345
2 Value: {"name": "foo bar", "email": "foo@bar.com", "designation": "software
  developer"}
```

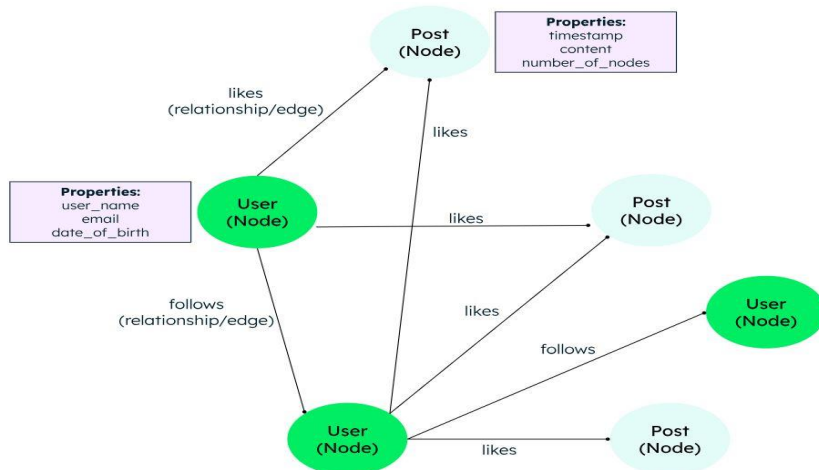
➤ **Wide-column stores:**

This type of database stores data in tables, rows, and dynamic columns. 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 such database include Apache Cassandra and HBase. Below is an example of how data is stored in such database.

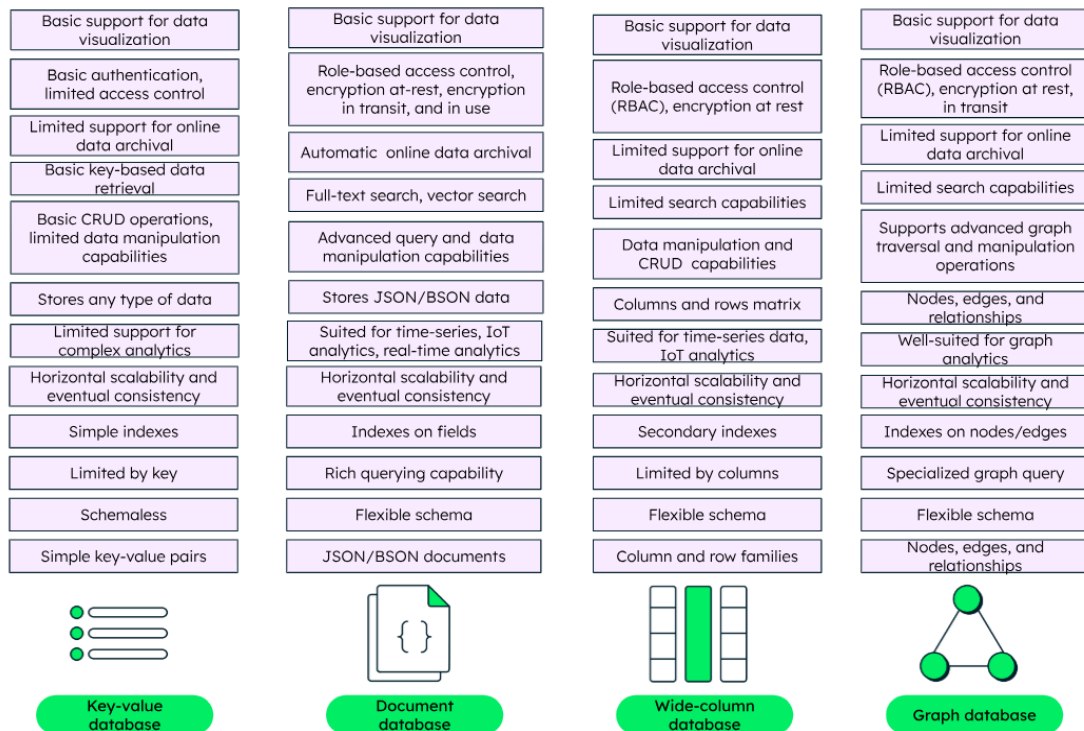
name	id	email	dob	city
Foo bar	12345	foo@bar.com		Some city
Carn Yale	34521	bar@foo.com	12-05-1972	

➤ **Graph databases:**

Here, data is stored 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 such databases are Neo4J and Amazon Neptune. MongoDB also provides graph traversal capabilities using the \$graphLookup stage of the aggregation pipeline. Below is an example of how data is stored in such data



## SUMMARY



## Top advantages of using NoSQL database

### 2.4. Advantages and disadvantages of using NoSQL databases

#### 2.4.1. Advantages

While SQL databases have been around a lot longer than NoSQL databases, NoSQL has many advantages depending on what you are looking for. Here are some advantages of using NoSQL databases.

##### ➤ Flexible scalability

- ❖ NoSQL databases are highly scalable and can be modified to meet the unique scaling needs of any business. They can be scaled both horizontally and vertically which provides a clear advantage over SQL databases.
- ❖ The lack of structure in the data is what allows NoSQL database to scale horizontally. Because NoSQL databases require less structuring than SQL, every item is self-contained and independent. As a result, each object is kept on separate servers and need not to be linked.

##### ➤ Flexible data types

- ❖ It allows you to store and retrieve data with only limited or no requirements for the predefined schema. This means that your application can quickly adapt as new

types of information are added without adjusting table structures, modifying indexes, and so forth.

➤ Large amount of data storage

- ❖ Many NoSQL databases can handle extensive datasets, making them ideal for big data applications, Internet of Things (IoT), and other real-time analytics. Also, there is no single point of failure in such database hence, there is no limit to how much data it can hold.

➤ Simplicity and less code

- ❖ Many NoSQL database management systems require only a few lines of code, which is ideal for developers who want to get started quickly. For example, MongoDB is a NoSQL database management system that allows developers to store data in flexible structures and retrieve it with code written in the language of their choice.

➤ Less ongoing database maintenance

- ❖ NoSQL databases don't require the same level of ongoing database administration as traditional relational databases because they can automatically partition and replicate information across nodes. This means you won't need additional IT infrastructure, software licenses, or expensive hardware like standard SQL which can be a significant financial burden for your business.

#### **2.4.2. Disadvantages**

While there are some obvious advantages to NoSQL, they do come at a cost. The main disadvantages of NoSQL are;

➤ Queries are less flexible

- ❖ NoSQL databases are more flexible when storing a wide variety of data structures, but lack the complex query functionality found in SQL. This means that you won't use many types of standard queries with this type of database management system.

➤ Less mature

- ❖ Since SQL has been around a lot longer, it is generally universal and more mature. One of the most significant disadvantages is that NoSQL may be challenging to find solutions to problems.

➤ NoSQL isn't designed to scale by itself.

- ❖ While there are ways to scale out your application using NoSQL database management systems, their design limits the amount of traffic they can accommodate by themselves. Instead, you'll need additional infrastructure components like load balancing which will increase both hardware costs as well as operational overhead.

### 3. METHODOLOGY

#### 3.1. Overview

##### ➤ **System Requirements.**

- ❖ The Biometric student attendance application requires a high performant, scalable and high fault tolerant database.
- ❖ So, coming up with a proposal such as a NoSQL database, the above requirement can be met.

##### ➤ **Database Design.**

- ❖ The main entities in our database will be Students, Courses and Instructors.
- ❖ The derived entities will be gotten from the course code field from the Courses table(collection). Each course code is gotten from its course and then used to create its own collection will then act as the general session for that course.
- ❖ Each document(record/row) will be used as an individual attendance session since it will be possible to include an array of students as a field in that row who attended that lecture.

##### ➤ **Data Collection and Storage.**

- ❖ The way attendance data is actually collected, processed and stored makes a NoSQL database to be used because they are generally more performant than SQL databases as a variety of data can be stored with NoSQL.

#### 3.2. Justification for choosing MongoDB

For our system, the most favorable NoSQL database to use will be the document type because of the following reasons.

- **Schema Flexibility:** Each document has its own structure making it flexible enough to handle all sorts of data.
- **Nested Data Support:** Documents can contain nested data structures and arrays, making it easier to represent complex relationships or dependencies within a single document.

- **Rich-Querying:** Complex queries are performed using the structure and content of the documents. This is advantageous for retrieving and manipulating nested data.
- **Horizontal Scalability:** Document-Oriented databases are designed for horizontal scalability, allowing them to distribute data across multiple servers. This supports high availability, fault tolerance and high transaction rate.
- **Consistency Models:** Strong consistency (Updates are immediately made visible to all users) and Eventual Consistency (Updates propagates asynchronously).

### 3.3. Designing the Database Schema

The schema of a document-oriented NoSQL database is characterized by its flexible and dynamic nature, where documents (equivalent to rows in SQL) can have varying structures within the same collection. This flexibility is well-suited for agile development practices.

- **Dynamic Schema:** Document-oriented databases like MongoDB do not enforce a rigid schema like SQL databases. Instead, they support dynamic schemas, where each document (equivalent to a row or record in a relational database) can have its own structure.
- **Document Structure:** A document is a JSON-like (or BSON in MongoDB's case) structure that can contain key-value pairs, nested documents, or arrays. The schema of a document is defined by its fields and their data types, but different documents within the same collection (equivalent to a table in SQL) can have different structures.
- **Flexible Data Model:** Unlike SQL databases where tables must adhere to a predefined schema with fixed columns and data types, document-oriented databases allow documents to evolve independently. This flexibility accommodates changes in application requirements without requiring schema migrations or downtime.

### 3.4. Data Modelling

Data modeling is the process of designing the structure and relationships of data to support the intended business processes and requirements of an application or system.

- **Denormalization:**
  - ❖ **Purpose:** Optimizes read performance by reducing the number of joins required in queries.

- ❖ **Process:** Involves adding redundant data (for example, duplicating some data from related tables) to improve query performance, especially for analytical or reporting databases.
- ❖ **Outcome:** Faster query execution at the expense of increased storage space and potential update anomalies, suitable for read-heavy applications.
- **Indexing:**
  - ❖ **Purpose:** Faster retrieval of data by providing a quick lookup mechanism.
  - ❖ **Process:** Identify fields that are frequently used in queries for filtering. These columns are best candidates.
  - ❖ **Outcome:** Faster Queries that involve indexed columns execute faster as the database can locate them quicker.

## 4. DATABASE DESIGN

### 4.1. Overview of the designed schema

The database schema for the mobile attendance system is designed to support efficient storage, retrieval, and management of attendance-related data. The schema encompasses various entities such as Users, Records, Sessions, Courses, Notifications, Feedback, Settings, and Reports. The system is designed using a NoSQL database (Cloud MongoDB) to leverage its flexibility and scalability features.

*Table 1: Overview Of database Schemas*

Collections	Number of fields	Field of array type	Array having objects as items
<b>Users</b>	9	1	No
<b>Courses</b>	4	0	1
<b>Sessions</b>	5	0	1

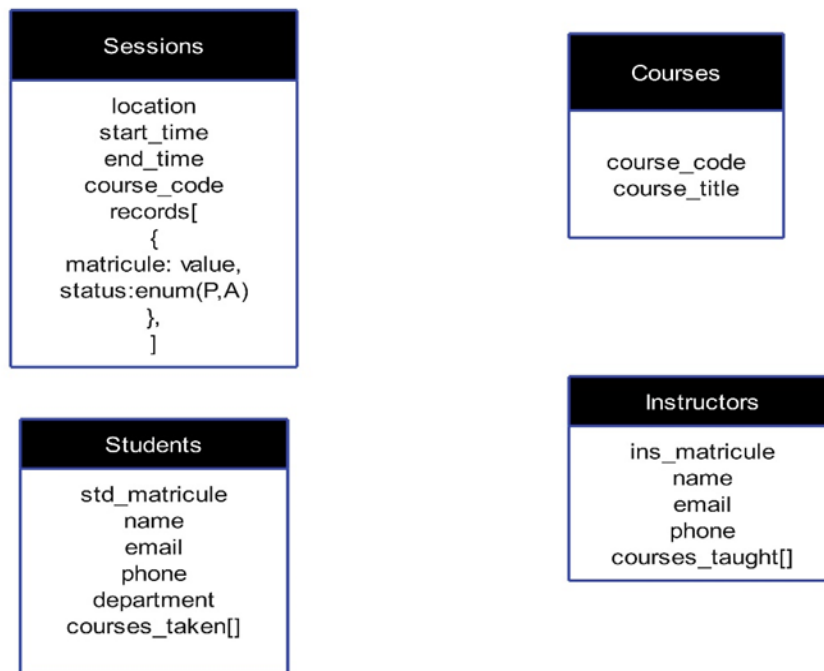
### 4.2. Conceptual database Design

ERD diagrams helps us visualize our database schema at a conceptual level for which is very important in order to pick up some quick aspects noticeable in our database.

#### 4.2.1. Entities and Attributes:

- **Course:** title, code, and level, which are required and Strings, schedules (array of objects containing day and time slot)
- **User:** Name, Email, password, telephone, Role (Admin, Instructor, Student), matriculation number, department, courses
- **Session:** Date, start Time and End time which are Dates, Location, Course (Object reference to course and required), Records (array of objects attendance record objects)

- **AttendanceRecords:** userID, Status (Present, Absent)
- **Notification:** UserID, Type (Push, Email, SMS), Message, Timestamp
- **Report:** UserID, DateGenerated, Content, type
- **Feedback:** userId (object, reference to User, required), message, timestamp
- **Settings:** user, notification preferences, account information, security settings which are all objects



#### 4.3. Logical and Physical Database Design (collections, documents, tables, or key structures)

In the design and implementation of our database, we have several main collections: User, Courses, Instructors, and Sessions. Each collection serves a specific purpose and is structured to efficiently support the application's functionalities.

##### ➤ User Collection

The User collection stores information about all users within the system. This includes students, instructors, and admins. The collection contains fields such as name, matricule, email, phone, department, courses, and role. The courses field represents an array of courses taken by students or taught by instructors. Each document in this collection represents a single user entity.

##### ➤ Courses Collection



The Courses collection holds information about all the courses offered. This includes details such as the course title, code, level, and schedule. Each document in this collection represents a single course entity.

➤ **Sessions Collection**

The Sessions collection is critical for attendance tracking and related queries. Each session document includes fields such as the date, deadline, location, course, and records. The records field is an array of objects, each containing a student matricule and their attendance status (present or absent). Each document represents a single session entity, showing all students and their attendance status for that particular session.

➤ **Report Collection**

The Report collection stores metadata about reports generated for users, including the user ID, report type, and generation date.

➤ **Feedback Collection**

The Feedback collection holds user feedback messages, associated with a user ID and timestamp.

➤ **Settings Collection**

The Settings collection manages user-specific settings, such as notification preferences and account information.

➤ **Notification Collection**

The Notification collection stores user notifications, including type, message, timestamp, and read status.

**Table 2: All Collections**

<b>User Collection</b> <pre>{   "_id": ObjectId,   "name": String,   "email": String,   "password": String,   "tel": String,   "userType": String,   "studentId": String,   "instructorId": String,   "courseCodes": [String],   "department": String }</pre>	<b>Course Collection</b> <pre>{   "_id": ObjectId,   "title": String,   "code": String,   "level": Number,   "schedule": [     {       "dayOfWeek": String,       "startTime": String,       "endTime": String     }   ] }</pre>	<b>Session Collection</b> <pre>{   "_id": ObjectId,   "date": Date,   "deadline": Date,   "location": String,   "course": ObjectId,   "records": [     {       "status": String,       "student": ObjectId     }   ] }</pre>	<b>Settings Collection</b> <pre>{   "_id": ObjectId,   "userID": ObjectId,   "notificationPreferences": {     email: Boolean,     sms: Boolean,     push: Boolean,   },   "accountInformation": {},   "securitySettings": {     loginAlerts,     twoFactorAuth   },   "privacySettings": {     "datasharing": Boolean   } }</pre>
<b>Notification Collection</b> <pre>{   "_id": ObjectId,   "user": ObjectId,   "type": String,   "message": String,   "timestamp": Date,   "read": Boolean }</pre>	<b>Notification Collection</b> <pre>{   "_id": ObjectId,   "user": ObjectId,   "type": String,   "message": String,   "timestamp": Date,   "read": Boolean }</pre>	<b>Feedback Collection</b> <pre>{   "_id": ObjectId,   "userID": ObjectId,   "message": String,   "timestamp": Date }</pre>	
		<b>Report Collection</b> <pre>{   "_id": ObjectId,   "userID": ObjectId,   "reportType": String,   "generatedDate": Date }</pre>	

#### 4.4. How data is organized to support specific use cases and query patterns

##### ❖ User Management

User data is stored in the User collection, allowing for efficient retrieval and management based on email and userType. This organization supports various user-related operations such as authentication, authorization, and profile management.

##### ❖ Course Scheduling

Course details and schedules are managed in the Course collection, while session details are stored in the Session collection. Sessions reference courses via the course field, enabling easy retrieval of course-specific session information.

##### ❖ Attendance Tracking

Attendance records are embedded within the Session documents in the records array, facilitating quick access to attendance data for specific sessions.

### ❖ **Reporting**

Report metadata is stored in the Report collection, linked to users through the userID field. This structure allows for generating and retrieving user-specific reports efficiently.

### ❖ **Feedback Management**

Feedback messages are stored in the Feedback collection, associated with users via the userID field. This setup supports the submission and viewing of feedback related to specific users.

### ❖ **User Settings**

User settings are managed in the Settings collection, with each document linked to a user via the userID field. This allows for easy updates and retrieval of user-specific settings.

### ❖ **Notifications**

Notifications are stored in the Notification collection, linked to users through the user field. This design supports efficient querying and updating of user notifications, including marking them as read or unread.

The data is stored and organized using BSON which is just a data interchange format use by MongoDB databases which is similar to JSON. Each collection is basically made up of documents which are basically JavaScript objects.

MongoDB supports a variety of query patterns and manipulate data from collections. Below are some examples:

*Table 3: Use Cases and Queries*

Use case	Query	Code
Retrieval of user data	Retrieval based of email and role	<code>db.User.find({ email: 'email', userType: "student"})</code>
Course Scheduling	Reference courses via course fields in sessions for ease retrieval	<code>db.Session.findById(sessionId).populate(course, 'title')</code>
Attendance Tracking	Embeds records within sessions	<code>Session.findById(sessionId).populate('records.student', 'name')</code>

## 5. IMPLEMENTATION

In this chapter, we detail the process of implementing the database schema, including the tools and technologies used, challenges encountered, and performance optimizations made.

### 5.1. Description of the implementation process

The implementation process began with defining the data requirements and relationships between different entities. After establishing the conceptual design, we moved on to creating the logical and physical schema using MongoDB due to its flexibility and scalability for handling document-based data.

#### ❖ **Schema Definition:**

- ✓ We defined the schemas for each collection (User, Courses, Sessions, Instructors, Reports, Feedback, Settings, Notifications) using Mongoose, a MongoDB object modeling tool designed to work in an asynchronous environment.

#### ❖ **Data Validation and Constraints:**

- ✓ Mongoose schemas were used to enforce data validation and constraints at the application level, ensuring that only valid data is saved to the database.
- ✓ Validation logic for user input, such as email format, phone number format, and password strength, was incorporated.
- ✓ Courses: Validated course codes to ensure they are unique and follow a specific format.
- ✓ Sessions: Ensured session dates and times are valid and deadlines are correctly set.
- ✓ Reports: Validated report types and ensured they are generated within appropriate contexts.
- ✓ Feedback: Checked that feedback messages are provided and timestamps are correctly recorded.
- ✓ Settings: Validated notification preferences and security settings to ensure they comply with specified formats and constraints.

#### ❖ **Relationships and References:**

- ✓ Relationships between collections were established using ObjectId references, allowing for efficient data retrieval and maintaining data integrity.
- ✓ Example:
  - Session collection references the Course and User collections to link attendance records to specific sessions and students.

- Report collection references the User collection to associate reports with specific users.
- Feedback collection references the User collection to link feedback to users.
- Settings collection references the User collection to store user-specific settings.
- Notification collection references the User collection to send notifications to specific users.

#### ❖ **Indexing:**

- ✓ Indexes were created on frequently queried fields to improve query performance. For example, all collections had the **\_id** field for indexing and referencing automated by Mongoose but also, they were identifiers explained below.
- ✓ Additional indexes were created on:
  - **User Collection:** Index on the email field for fast retrieval during authentication and user lookup.
  - **Courses Collection:** Index on the code field to quickly find courses by their code.
  - **Sessions Collection:** Index on the course and date fields to efficiently retrieve sessions for specific courses and dates.
  - **Reports Collection:** Index on the userID and generatedDate fields to quickly access reports for specific users and within date ranges.
  - **Feedback Collection:** Index on the userID and timestamp fields for efficient retrieval of feedback provided by users.
  - **Settings Collection:** Index on the userID field to quickly access and update user-specific settings.
  - **Notification Collection:** Index on the userID and createdAt fields to efficiently manage and deliver notifications.

#### ❖ **API Development:**

- ✓ The API development process was a critical part of our project, aimed at facilitating seamless interaction between the client applications and the database.

- ✓ Built using Express.js, the API supports a range of functionalities including user management, course management, session tracking, report generation, feedback submission, settings configuration, and notification handling
- ✓ Authentication and authorization mechanisms were implemented to secure the API.
- ✓ Key aspects of the API development included
  - **Authentication and Authorization:** Secure endpoints for user registration (auth/signup) and login (/auth/login), ensuring only authenticated users can access sensitive information. A POST request with some user data and password would respond with users' information and access tokens
  - **User Management:** Endpoints for retrieving (/users/:id), and updating (/users/:id) user details. Similar to auth routes above but PATCH request with just the users' information that needs to be updated
  - **Course Management:** Endpoints (/courses) for retrieving all courses using GET and creating new courses with POST.
  - **Session Management:** Endpoints (/sessions) for listing and creating sessions, enabling efficient attendance tracking. Instructors can initiate sessions and students mark their attendance
  - **Report Management:** Endpoints for listing (/reports) and generating (/generate-report) reports which generates report based on specified parameters and exporting (/export-report) reports that takes report data and export in specified format.
  - **Feedback Management:** Endpoints for viewing (/feedbacks) and submitting (/feedbacks) feedback.
  - **Settings Management:** Endpoints for retrieving (/settings/:userID) and updating (/settings/:userID) user settings.
  - **Notification Management:** Endpoints for listing (/notifications) and sending (/notifications) notifications.

## 5.2. Tools and technologies used

- **MongoDB:** A NoSQL database used to store the collections, offering scalability and flexibility for managing document-based data.

- **Mongoose:** An ODM (Object Data Modeling) library for MongoDB and Node.js, used to define schemas and interact with the database. It helps enforce schema validation and manage relationships between collections.
- **Node.js:** A JavaScript runtime environment used for server-side development, enabling the creation of scalable and high-performance applications.
- **Express.js:** A web application framework for Node.js, used to build the API and handle routing, middleware, and HTTP requests.
- **JWT (JSON Web Tokens):** A library for token-based authentication used to securely transmit information between parties. Employed for user authentication, providing stateless and scalable session management.
- **bcrypt:** A library for hashing passwords, ensuring secure storage of user credentials by converting plaintext passwords into hashed values.
- **validator:** A library used to validate and sanitize user inputs, ensuring that only properly formatted and secure data is processed.
- **Nodemailer:** A module for Node.js applications to send emails. It was used to implement the email notification system, enabling the application to send automated emails for various notifications and alerts.
- **SMS Integration:** Integration with an SMS gateway service to send SMS notifications. This ensures that users receive timely updates and notifications via SMS, enhancing communication and engagement. Libraries such as **twilio** or **nexmo** can be used for this purpose.

By incorporating these tools and technologies, the implementation ensures a robust, secure, and efficient system that meets the needs of the application and its users. Each tool and technology were chosen for its specific strengths and ability to handle particular aspects of the application's requirements.

### 5.3. Challenges encountered and how they were addressed

#### ❖ **Data Validation:**

- **Challenge:** Ensuring all user inputs are valid and secure.
- **Solution:** Used the **validator** library for comprehensive input validation and sanitization for instance email, phone format and password strength. Additionally, **Mongoose schema** validation ensured that only valid data (required fields, correct format and data types) is saved to the database.

#### ❖ **Authentication and Password Security**

- **Challenge:** Storing user passwords securely.
- **Solution:** Implemented password hashing using **bcrypt** to store hashed passwords, ensuring they are not stored in plaintext.
- ❖ **Performance Optimization:**
  - **Challenge:** Ensuring the database performs efficiently under load and retrieval especially for large data sets and complex queries.
  - **Solution:** Implemented **indexing** on frequently queried fields and optimized query patterns to reduce load times and improve responsiveness.
- ❖ **Error Handling:**
  - **Challenge:** Managing errors gracefully and providing meaningful feedback to users.
  - **Solution:** Implemented comprehensive error handling in the API, ensuring that errors are logged and meaningful messages are returned to the client.
- ❖ **Notification System**
  - **Challenge:** Send emails and SMS messages to users.
  - **Solution:** Used Nodemailer for sending emails. This enabled the application to send automated emails for notifications, alerts, and updates to users.
- ❖ **Handling Relationships to avoid over nesting**
  - **Challenge:** Managing relationships between different collections and ensuring data consistency across them.
  - **Solution:** Utilized ObjectId references to establish relationships between collections, such as linking sessions to specific courses and users. This approach ensured that related data could be efficiently retrieved by populating referenced documents and maintained data consistency. This simplifies queries and user engagement.
- ❖ **Environment Configuration**
  - **Challenge:** Configuring and deploying the application in different environments while ensuring consistency and reliability.
  - **Solution:** Used environment variables to manage configuration settings, ensuring that sensitive information (such as database credentials and API keys) was not hardcoded in the application.



By addressing these challenges with targeted solutions, the implementation process ensured the development of a robust, secure, and efficient system that met the application's requirements and provided a seamless user experience.

#### 5.4. Performance considerations and optimizations made

In addition to indexing and the above solutions that could improve performance, several optimizations were made during development to maintain optimal performance

##### ❖ **Efficient Querying:**

- Designed queries to minimize the amount of data retrieved and processed, using projection and aggregation pipelines where appropriate to improve query speed. That is **populate method selectively** to fetch only necessary fields and **lean method for read-only queries** to return plain JS instead of documents

##### ❖ **Caching:**

- Implemented in-memory caching strategies using layers like **Redis** for frequently accessed data, reducing the load on the database and improving response times.

##### ❖ **Data Partitioning:**

- Considered data partitioning strategies to handle large datasets, although this was not implemented initially, it remains a potential optimization for future scalability.
- Implemented **pagination** for queries returning large datasets, such as user lists or session records, to limit the number of documents returned in a single query.

##### ❖ Frequent write operations can cause performance degradation if not handled efficiently.

- Implemented **asynchronous processing** for non-critical write operations, allowing the application to handle other requests while the writes are being processed in the background.

## 6. TESTING AND EVALUATION

The testing and validation phase ensures robustness, reliability, and security of the system.

This phase involved various testing methodologies, including functional testing, unit testing, integration testing, and end-to-end testing. The primary goals were to verify that all functionalities work as expected, handle edge cases, and ensure that the system performs well under various conditions.

## 6.1. Overview of testing Strategies used

Our testing strategy involved:

- **Functional Testing:** To verify that the system works as expected from an end-user perspective.
  - Tools Used: Postman, Thunder Client, Jest, Supertest.
- **Performance Testing:** To evaluate the system's performance under load.
  - Tools Used: Jest, Supertest.
- **Security Testing:** To identify and mitigate potential vulnerabilities.
  - Tools Used: Manual security reviews, jest, supertest.

## 6.2. Test cases and scenarios tested (e.g., performance benchmarks, scalability tests)

### 6.3. Functional Testing

Performance tests were conducted to evaluate the system's responsiveness and stability under load.

#### ❖ Example Functional Test for User Signup:

- **Success Case:** Ensure that user signup endpoint successfully creates a new user with valid input.
- **Failure Case:** Validate error handling when invalid data is provided.

#### ❖ Example to Mark Attendance for an ongoing session:

- **Success Case:** Verify that marking attendance for an ongoing session updates the session records correctly.
- **Failure Case:** Check error handling when trying to mark attendance for a non-existent session or invalid user.

#### ❖ Example to get attendance data in real-time for an ongoing session:

- **Success Case:** Confirm that retrieving attendance data for an ongoing session provides accurate information.
- **Failure Case:** Handle scenarios where the session or attendance data is unavailable.

#### ❖ Example of Sending Notification:

- **Success Case:** Ensure that notifications are sent successfully to users when triggered by specific events (e.g., session creation, attendance marking).
- **Failure Case:** Validate notification failure handling in case of errors (e.g., email server issues).

#### 6.4. Performance Testing

Performance tests were conducted to evaluate the system's responsiveness and stability under load.

- ❖ **Example Performance Test: Simulate 1 course and 1 session creation plus 100 signups and 100 attendances checked concurrently took 40s**
  - **Observation:** The system was able to handle creating a course, session creation, 100 signups, and attendance marking for 100 users concurrently within 40 seconds.
  - **Result:** The performance was satisfactory under the specified load.
- ❖ **Test Case: Simulate 1000 concurrent users signing up. 45s**
  - **Observation:** The system managed to handle 1000 concurrent signups within 45 seconds.
  - **Result:** The performance meets the requirement for handling a large number of concurrent users during sign-up processes.

#### 6.5. Security Testing

We conducted to evaluated how the system will behave to unauthorized access and malicious input

- ❖ **Unauthorized Access:**
  - **Outcome:** Unauthorized attempts to access restricted endpoints were effectively blocked with appropriate error responses (e.g., status code 401 Unauthorized), ensuring protection against unauthorized access.
- ❖ **Malicious Input Handling:**
  - **Outcome:** Input validation and sanitation procedures successfully mitigated common security threats such as XSS and SQL injection attacks, ensuring the integrity of data processing.
- ❖ **Data Protection:**
  - **Outcome:** Sensitive data, both in transit and at rest, was adequately protected through encryption mechanisms (e.g., HTTPS for data in transit, database-level encryption for data at rest), adhering to security best practices.
- ❖ **Error Handling:**
  - **Outcome:** Error messages were carefully crafted to avoid exposing sensitive information and were handled gracefully to prevent information leakage, maintaining confidentiality and system resilience.

## 6.6. Results and Discussions

- ❖ The testing and evaluation phase confirmed that the system:
  - Successfully met functional requirements across key features such as user signup, attendance marking, real-time data retrieval, and notification delivery.
  - Demonstrated satisfactory performance under load, handling concurrent operations within acceptable response times and maintaining system stability.
  - Identified and addressed potential vulnerabilities effectively, ensuring robust protection against unauthorized access, malicious inputs, and data breaches.
- ❖ Based on the conducted tests:
  - All functional requirements were met as per specifications, with all critical scenarios tested successfully.
  - Performance benchmarks aligned closely with expected outcomes of at most 5s per action per user, validating the system's capability to scale and handle concurrent user operations effectively.
  - Security measures implemented were found to be effective, meeting or exceeding industry standards and regulatory requirements for data protection and system security.

## 7. CONCLUSION

### 7.1. Summary of key findings

We realized sensible information such as fingerprints is not allowed to be stored in any database. Only images of the fingerprints can be stored and readily available for the public for educational purposes.

Instead, we are going to use already stored and hashed fingerprints in the mobile device for our attendance tracking.

### 7.2. Future considerations and improvements

For future considerations, the backend awaits client HTTP request so depending on the type or load of the request sent, we are considering adding or modifying some of the end points to reduce the burden on the server of need be.

## 8. APPENDICES

Github Repo for database implementation code: <https://github.com/N-Jerry/Group11/tree/server>

API, Test Cases and screenshots

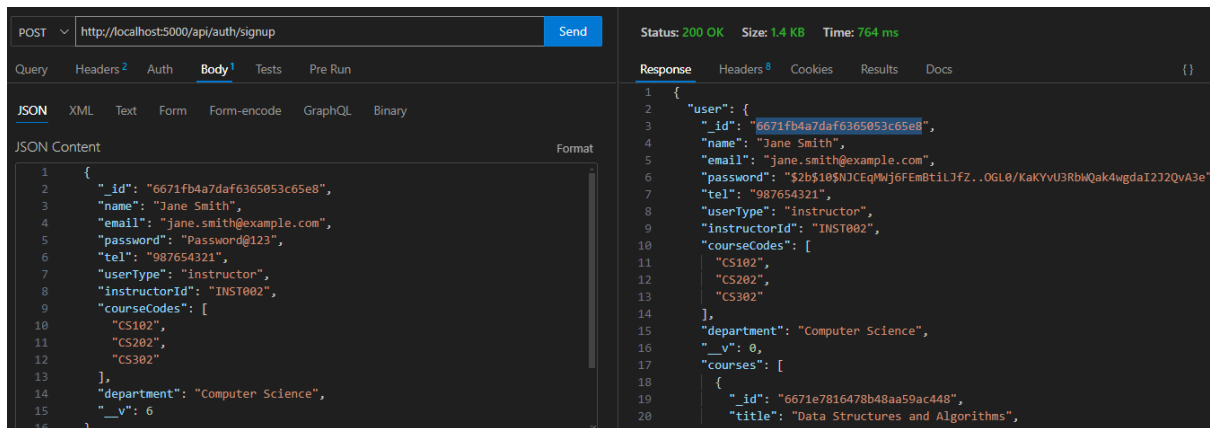


Figure 1: API signup request

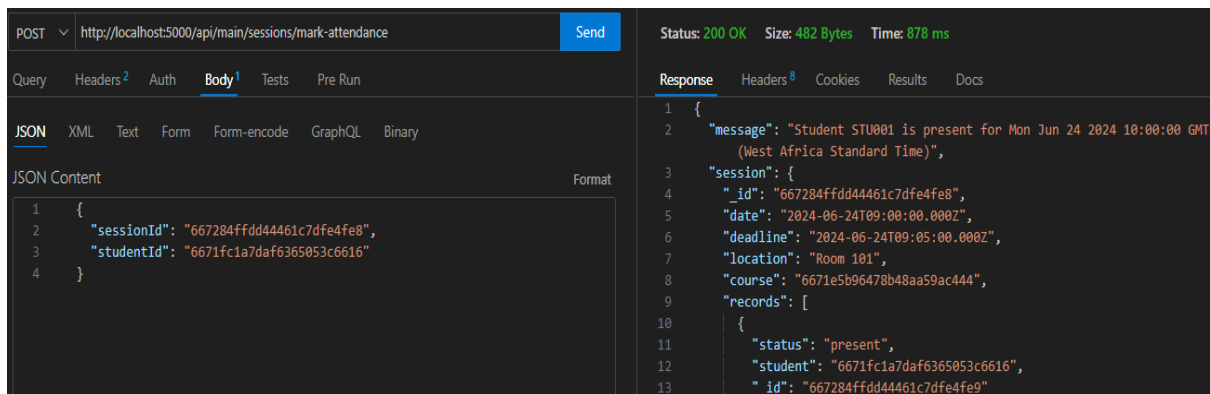


Figure 2: Mark Attendance request

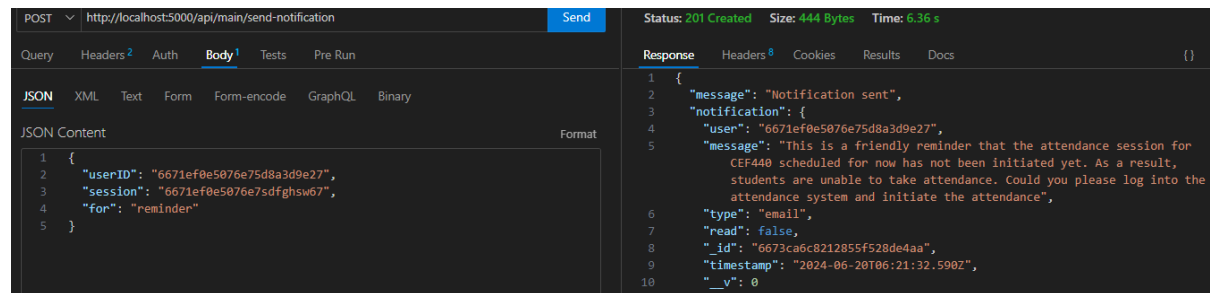


Figure 3: Send Notification Request

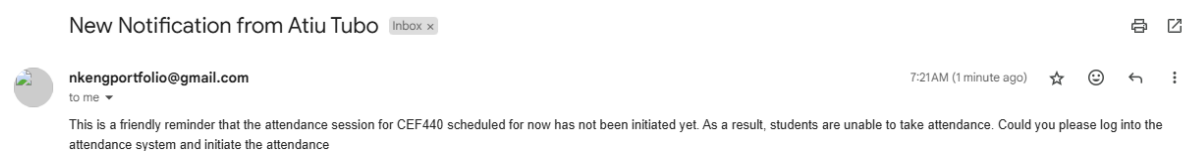


Figure 4: Receive Reminder Email

```
Status: 200 OK   Size: 10.38 KB   Time: 548 ms

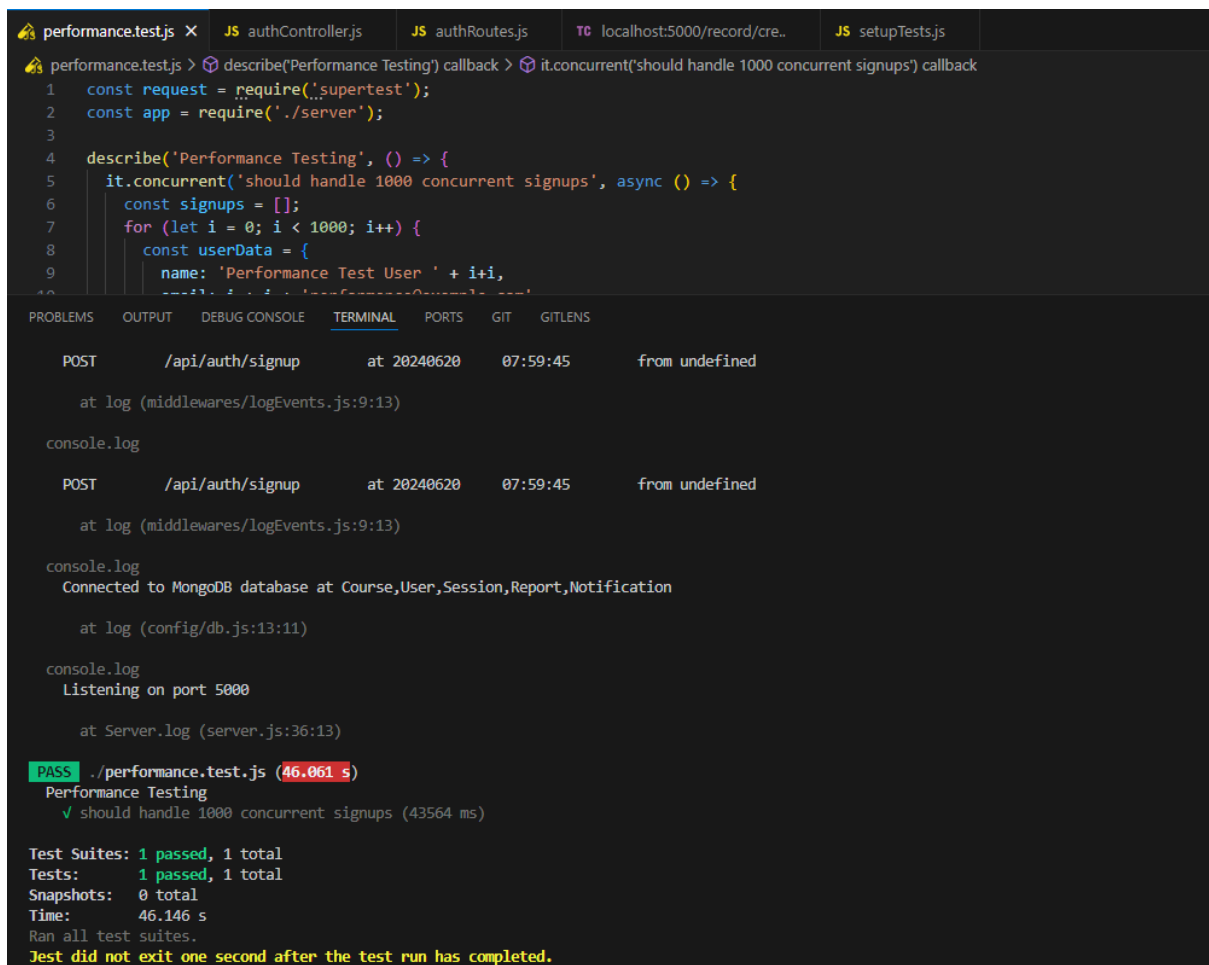
Response  Headers 8  Cookies  Results  Docs
1  [
2    {
3      "_id": "667284ffdd44461c7dfe4fe8",
4      "date": "2024-06-24T09:00:00.000Z",
5      "deadline": "2024-06-24T09:05:00.000Z",
6      "location": "Room 101",
7      "course": {
8        "_id": "6671e5b96478b48aa59ac444",
9        "code": "CS101"
10     },
11     "records": [
12       {
13         "status": "present",
14         "student": {
15           "_id": "6671fc1a7daf6365053c6616",
16           "studentId": "STU001"
17         },
18         "_id": "667284ffdd44461c7dfe4fe9"
19       },
20     ]
21   }
```

Figure 5: Monitor Attendance in real Time

```
PASS ./performance.test.js (43.88 s)
Performance Testing - Attendance Marking
  ✓ should mark all users present for a session (31716 ms)

Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   0 total
Time:        43.969 s
```

Figure 6: Simulate 1000 signups and mark attendance concurrently



The image shows a VS Code editor with a file named `performance.test.js` open. The code defines a performance test suite using Jest and Supertest. It sets up a server and a MongoDB database, then runs a concurrent test where 1000 signups are simulated simultaneously. The terminal output shows the test passing successfully, with a total time of 46.146 seconds.

```
performance.test.js X JS authController.js JS authRoutes.js TC localhost:5000/record/cre.. JS setupTests.js

performance.test.js > describe('Performance Testing') callback > it.concurrent('should handle 1000 concurrent signups') callback
1 const request = require('supertest');
2 const app = require('./server');
3
4 describe('Performance Testing', () => {
5   it.concurrent('should handle 1000 concurrent signups', async () => {
6     const signups = [];
7     for (let i = 0; i < 1000; i++) {
8       const userData = {
9         name: 'Performance Test User ' + i+i,
10        // ...
11      };
12    }
13  });
14 }

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS GIT GITLENS

POST /api/auth/signup at 20240620 07:59:45 from undefined
at log (middlewares/logEvents.js:9:13)

console.log

POST /api/auth/signup at 20240620 07:59:45 from undefined
at log (middlewares/logEvents.js:9:13)

console.log
Connected to MongoDB database at Course,User,Session,Report,Notification
at log (config/db.js:13:11)

console.log
Listening on port 5000
at Server.log (server.js:36:13)

PASS ./performance.test.js (46.061 s)
Performance Testing
  ✓ should handle 1000 concurrent signups (43564 ms)

Test Suites: 1 passed, 1 total
Tests: 1 passed, 1 total
Snapshots: 0 total
Time: 46.146 s
Ran all test suites.
Jest did not exit one second after the test run has completed.
```

Figure 7: Simulate 1000 signups concurrently

```
1  const request = require('supertest');
2  const app = require('./server'); // Adjust the path to your server file
3
4  describe('Security Testing', () => {
5    it('should prevent XSS attacks', async () => {
6      // Test XSS attack through input field
7      const maliciousInput = `<script>alert('XSS Attack!');</script>`;
8      const response = await request(app)
9        .post('/api/main/courses')
10       .send({ userInput: maliciousInput })
11       .expect(500);
12    });
13
14    it('should restrict unauthorized access', async () => {
15      // Test unauthorized access to a restricted endpoint
16      const response = await request(app)
17        .delete('/api/auth')
18        .expect(401);
19
20      // Assert that unauthorized access is properly restricted
21      expect(response.body.error).toBe('Authorization token required');
22    });
23  });
```

at log (middlewares/logEvents.js:9:13)

**PASS** ./security.test.js

Security Testing

- ✓ should prevent XSS attacks (104 ms)
- ✓ should restrict unauthorized access (66 ms)

Test Suites: 1 passed, 1 total  
Tests: 2 passed, 2 total  
Snapshots: 0 total  
Time: 2.114 s, estimated 3 s

Figure 8: Security Testing