Document Based (NoSQL) Database

(Capstone Project)

2022 winter

By: Farah Jamal

Supervisor: Motasem Al-Diab & Fahed Jubair.

GitHub LinkedIn | FarahJamal

“We’re here to put a dent in the universe. Otherwise, why else even be here?”

“Be a yardstick of quality. Some people aren't used to an environment where excellence is expected.”

— Steve Jobs

Table of Contents

Introduction

What Is a Database?

Database defined

A database is an organized collection of structured information, or data, typically stored electronically in a computer system. A database is usually controlled by a database management system (DBMS). Together, the data and the DBMS, along with the applications that are associated with them, are referred to as a database system, often shortened to just database.

Data within the most common types of databases in operation today is typically modeled in rows and columns in a series of tables to make processing and data querying efficient. The data can then be easily accessed, managed, modified, updated, controlled, and organized. Most databases use structured query language (SQL) for writing and querying data. oracle

What is Structured Query Language (SQL)?

SQL is a programming language used by nearly all relational databases to query, manipulate, and define data, and to provide access control. SQL was first developed at IBM in the 1970s with Oracle as a major contributor, which led to implementation of the SQL ANSI standard, SQL has spurred many extensions from companies such as IBM, Oracle, and Microsoft. Although SQL is still widely used today, new programming languages are beginning to appear.

Evolution of the database

Databases have evolved dramatically since their inception in the early 1960s. Navigational databases such as the hierarchical database (which relied on a tree-like model and allowed only a one-to-many relationship), and the network database (a more flexible model that allowed multiple relationships), were the original systems used to store and manipulate data. Although simple, these early systems were inflexible. In the 1980s, relational databases became popular, followed by object-oriented databases in the 1990s. More recently, NoSQL databases came about as a response to the growth of the internet and the need for faster speed and processing of unstructured data. Today, cloud databases and self-driving databases are breaking new ground when it comes to how data is collected, stored, managed, and utilized.

Types of databases

There are many different types of databases. The best database for a specific organization depends on how the organization intends to use the data.

Relational databases

- Relational databases became dominant in the 1980s. Items in a relational database are organized as a set of tables with columns and rows. Relational database technology provides the most efficient and flexible way to access structured information.

Object-oriented databases

- Information in an object-oriented database is represented in the form of objects, as in object-oriented programming.

Distributed databases

- A distributed database consists of two or more files located in different sites. The database may be stored on multiple computers, located in the same physical location, or scattered over different networks.

Data warehouses

- A central repository for data, a data warehouse is a type of database specifically designed for fast query and analysis.

NoSQL databases

- A NoSQL, or nonrelational database, allows unstructured and semistructured data to be stored and manipulated (in contrast to a relational database, which defines how all data inserted into the database must be composed). NoSQL databases grew popular as web applications became more common and more complex.

Graph databases

- A graph database stores data in terms of entities and the relationships between entities.
- **OLTP databases.** An OLTP database is a speedy, analytic database designed for large numbers of transactions performed by multiple users.

These are only a few of the several dozen types of databases in use today. Other, less common databases are tailored to very specific scientific, financial, or other functions. In addition to the different database types, changes in technology development approaches and dramatic advances such as the cloud and automation are propelling databases in entirely new directions. Some of the latest databases include

NoSQL Databases

Why successful enterprises rely on NoSQL

- Support large numbers of concurrent users (tens of thousands, perhaps millions)
- Deliver highly responsive experiences to a globally distributed base of users
- Be always available – no downtime
- Handle semi- and unstructured data
- Rapidly adapt to changing requirements with frequent updates and new features

Types of NoSQL Databases

NoSQL Databases are mainly categorized into four types: Key-value pair, Column-oriented, Graph-based, and Document-oriented. Every category has its unique attributes and limitations. None of the above-specified database is better to solve all the problems. Users should select the database based on their product needs.

Types of NoSQL Databases:

- Key-value Pair Based
- Column-oriented Graph
- Graphs based
- Document-oriented

Project explanation

Requirements

Build a document based (NoSQl) Database based on JSON Objects, with cache system that will save data inside the file system.

Software requirements

- Java: Main Programming Language.
- Spring boot: Application Demo.
- Maven: Build Tool.
- Docker & docker-compose containerization.
- React and node.js for web app Demo.

Sequence Diagram

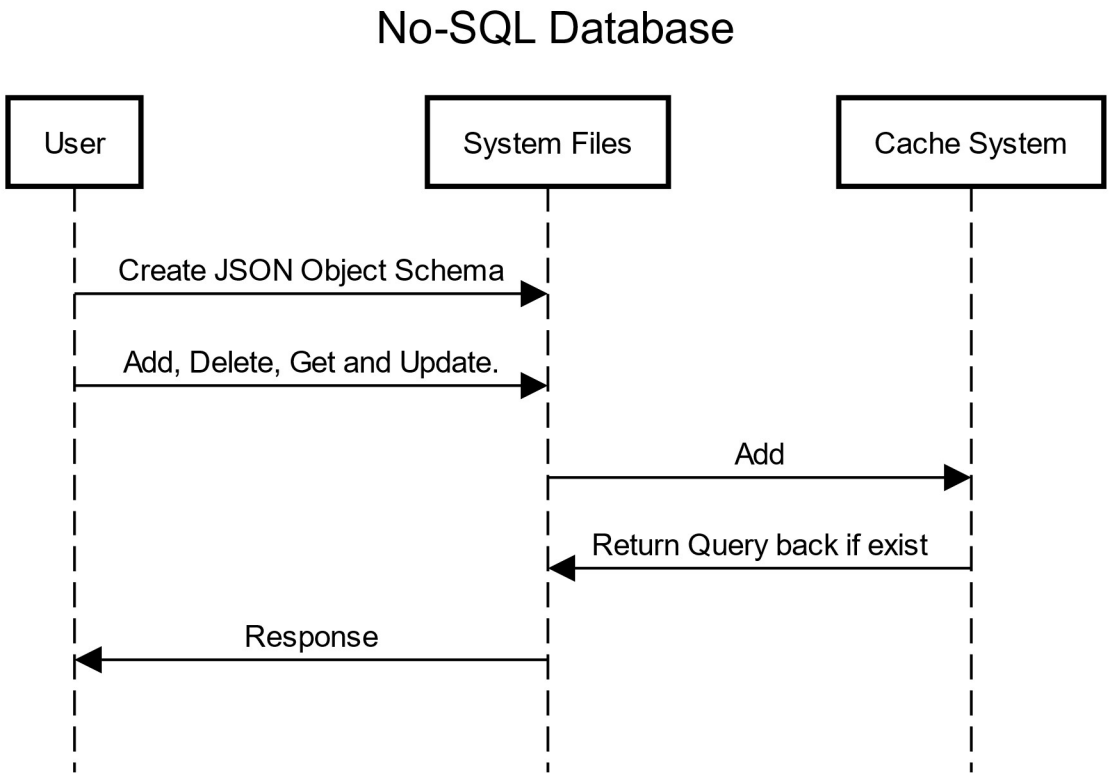


Figure 1- Sequence Diagarm.

1- User can create Database Schema.
2- User can Do full CRUD (create, read, update, delete) on schema data.
3- System should create or add files to the system add it to cache if not exist and return it back if exist.
4- Files system response to the user with status code.

Project Files

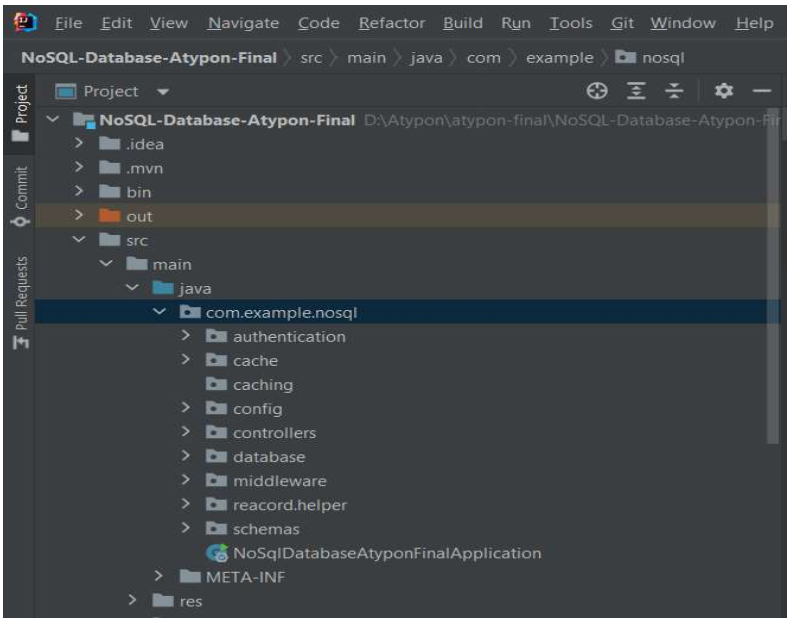


Figure 2- Project structure.

1- Schemas Builder:

a. Schema builders use to build the schema file user can create the schema with json object contains keys and values (name and type) and file name as parameter on the API call.

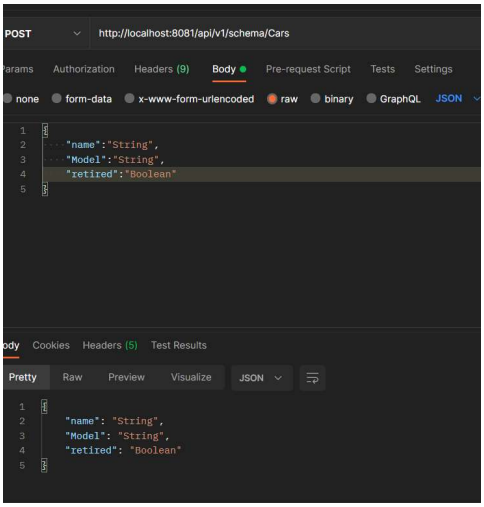


Figure 3- Schema Builder

Every schema built have something called record every record has built-in attributes:

```
{
  "data": [
```

```
    {
      "createdAt": "2022-06-7 at 23:01:59 EEST",
      "name": "Honda",
      "Model": "2019",
      "retired": false,
      "_id": "e9c5120b-b06d-4552-9d92-be584c91facd"
    }
  ]
}
```


Created_At: auto generated date which is showing when the schema has been created.

_id: primary key for the schema record auto generated when the object is created used mostly to get data by Id or delete and update data.

Caching System:

What is cache in programmer's world

according to Wikipedia, a cache is a hardware or software component that stores data so that future requests for that data can be served faster.

Caching Benefits:

- Faster access of data in O (1)
- Computation complexity once for the first time

Caching Types:

- Memory cache
- Database cache
- Disk cache, etc

To create a cache, we can simply use a map / dictionary data structure and we can get the expected result of O (1) for both get and put operation.

But we can't store everything in our cache. We have storage and performance limits.

A cache eviction algorithm is a way of deciding which element to evict when the cache is full. To gain optimized benefits there are many algorithms for different use cases.

- Least Recently Used (LRU)
- Least Frequently Used (LFU)
- First In First Out (FIFO)
- Last In First Out (LIFO) etc.

"There are only two hard things in computer science, Cache invalidation and naming things."

— Phil Karlton

Least Frequently Used (LFU)

In my design, I will use

- *HashMap (ConcurrentHashMap)* to get and put data in O(1)
- Doubly linked list

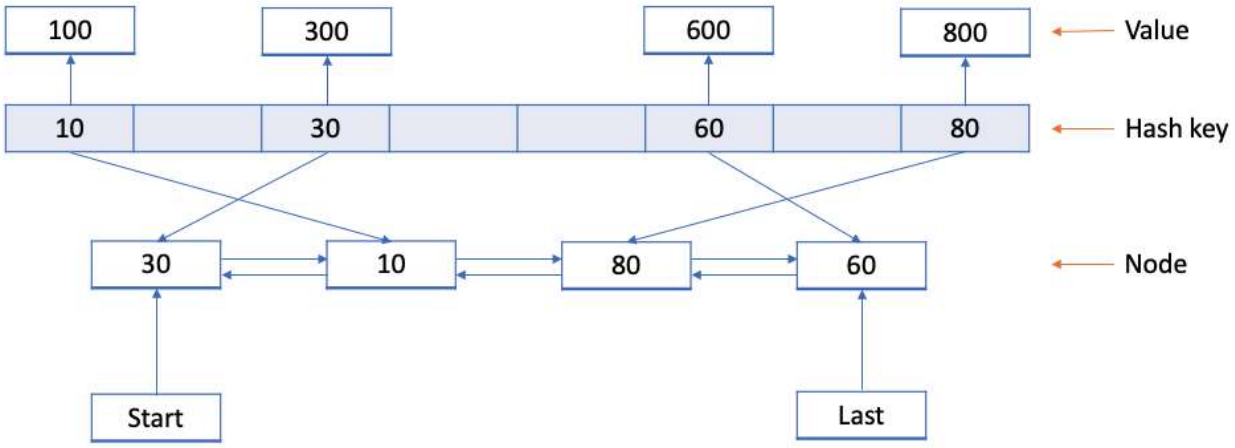


Figure 4- cache implementation strategy

I am using doubly linked list to determine which key to delete and have the benefit of adding and deleting keys in O(1).

Initially I will declare a model to store our key-value pair, hit count and reference node to point previous and next node.

Delete candidate is the least accessed entry.

We have to sort items based on the frequency the nodes being accessed.

To avoid getting deleted, for each accessed items needs to reach top based on their frequency.

- Iterated a loop, which swaps the node if the frequency is greater than it's next node frequency

- The user add record to the database.
- Hash function generates hash code.
- Hash code stored in the cache system or check if it is already stored.
- Cache going to return the result.

Each query in the database read/write will got unique hash code so the hash code will be the key and query results as value.

Anytime user ask for the query he will get it faster from the database.

Fixed size for the cache to make it running faster.

Is this implementation thread safe?

No. To be thread safe

- We can use *ConcurrentHashMap* instead of *HashMap*
- Use synchronized block

Java concurrency (multi-threading)

What is concurrency?

Concurrency is the ability to run several programs or several parts of a program in parallel. If a time-consuming task can be performed asynchronously or in parallel, this improves the throughput and the interactivity of the program.

A modern computer has several CPU's or several cores within one CPU. The ability to leverage these multi-cores can be the key for a successful high-volume application.

Process vs. threads

A *process* runs independently and isolated of other processes. It cannot directly access shared data in other processes. The resources of the process, e.g., memory and CPU time, are allocated to it via the operating system.

A *thread* is a so-called lightweight process. It has its own call stack but can access shared data of other threads in the same process. Every thread has its own memory cache. If a thread reads shared data, it stores this data in its own memory cache.

A thread can re-read the shared data.

A Java application runs by default in one process. Within a Java application you work with several threads to achieve parallel processing or asynchronous behavior.

Race Condition

What is race condition?

A condition in which the critical section (a part of the program where shared memory is accessed) is concurrently executed by two or more threads. It leads to incorrect behavior of a program.

In layman terms, a **race condition** can be defined as, a condition in which two or more threads compete together to get certain shared resources.

For example, if thread A is reading data from the linked list and another thread B is trying to delete the same data. This process leads to a race condition that may result in run time error.

There are two types of race conditions:

1. Read-modify-write
2. Check-then-act

The **read-modify-write** patterns signify that more than one thread first read the variable, then alter the given value and write it back to that variable. Let's have a look at the following code snippet

How to avoid race condition?

There are the following two solutions to avoid race conditions.

- Mutual exclusion
- Synchronize the process

Race condition in NoSQL-database project

Imagine we do multiple updates||delete in our database at the same time it will cause many errors

Imagine we have number record that I want to decrease it and run two threads it should decreased by the number I have added but maybe if it is not synchronized it going to increase one of them twice most times threads avoid it and get correct results but not all the time, so we have to expect the worst case.

Handle Race Condition in NoSQL-database project

As I said if I use **Synchronized** on the block, all code inside this block can be accessed only by one thread at the same time.

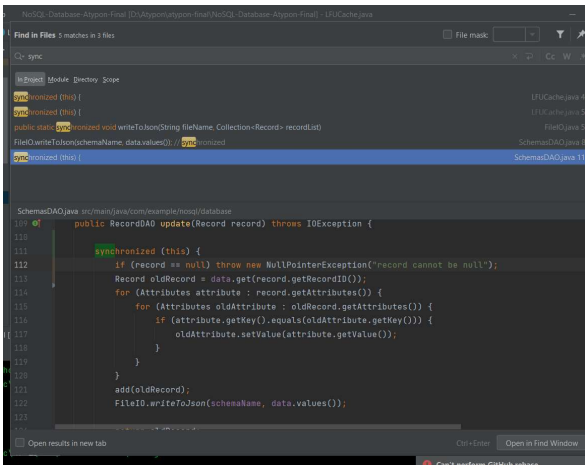


Figure 5- Handle Race condition

- Used **ConcurrentHashMap** which is thread safe version of **HashMap** inside Shemas DAO Class.

```
private ConcurrentHashMap parseFile() {
    ConcurrentHashMap<String, Record> data = new ConcurrentHashMap<>();
    try {
```

The transactions guarantee

ACID

Inherently a transaction is characterized by four properties (commonly referred as ACID) :

1. Atomicity
2. Consistency
3. Isolation
4. Durability

It's very important to understand those, hence we will discuss each and every one of them as follows.

- Atomicity:
  - All changes to data are performed as if they are a single operation. That is, all the changes are performed, or none of them are.

  For example, in an application that transfers funds from one account to another, the atomicity property ensures that, if a debit is made successfully from one account, the corresponding credit is made to the other account.

- Consistency
  - Data is in a consistent state when a transaction starts and when it ends.

  For example, in an application that transfers funds from one account to another, the consistency property ensures that the total value of funds in both the accounts is the same at the start and end of each transaction.

- Isolation
  - The intermediate state of a transaction is invisible to other transactions. As a result, transactions that run concurrently appear to be serialized.

For example, in an application that transfers funds from one account to another, the isolation property ensures that another transaction sees the transferred funds in one account or the other, but not in both, nor neither.

- Durability
  - After a transaction successfully completes, changes to data persist and are not undone, even in the event of a system failure.

For example, in an application that transfers funds from one account to another, the durability property ensures that the changes made to each account will not be reversed.

ACID in NoSQL-database project.

Atomicity:

- Check if record is valid record before update or write so file will not be affected if it is not valid.

```
if (!schema.isValidRecord(record)) {
    System.out.println("false");

    return false;
}
```

Consistency:

- in everything happened on database it will overwrite the file so it will guarantee it is going to stay well structured.

Isolation:

- each crud function has it is own functionality

Durability:

- files still in the system even if the system down.

Security

Advanced Encryption Standard (AES)

- The Advanced Encryption Standard (AES) is a symmetric block cipher chosen by the U.S. government...

Advanced Encryption Standard (AES) is a specification for the encryption of electronic data established by the U.S National Institute of Standards and Technology (NIST) in 2001. AES is widely used today as it is a much stronger than DES and triple DES despite being harder to implement. gfg

Points to remember

- AES is a block cipher.
- The key size can be 128/192/256 bits.
- Encrypts data in blocks of 128 bits each.

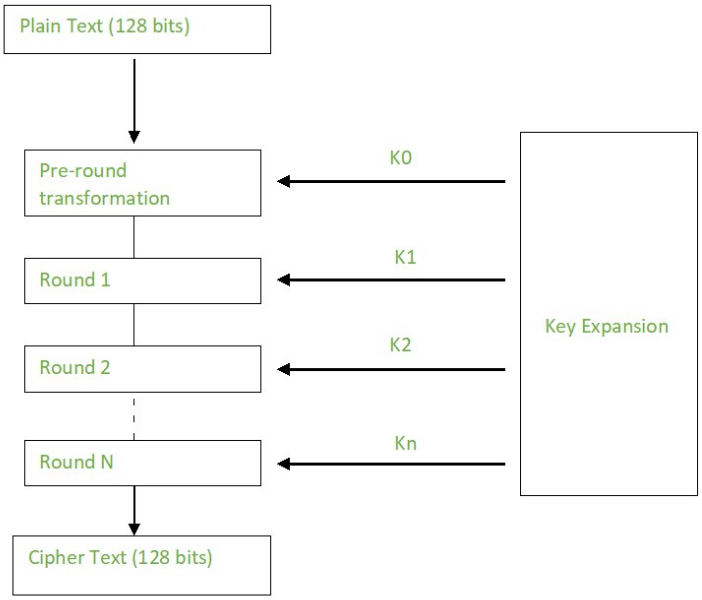


Figure 6- AES security

Security in NoSQL-database project

Middleware:

- it will validate auth header which is contains the access token generally generated after login successfully.

- In the future this app should have more security by make token expired after 1 hour for example so if anyone got your token should not be usable.

Client & Server protocol

Use database

- User can use database usually with GUI / Web app or Restful API
- In this app user will access it using REST API.

REST API

A REST API (also known as RESTful API) is an application programming interface (API or web API) that conforms to the constraints of REST architectural style and allows for interaction with RESTful web services. REST stands for representational state transfer and was created by computer scientist Roy Fielding.

NoSQL-database endpoints

GET endpoints:

/api/v1/test-connection

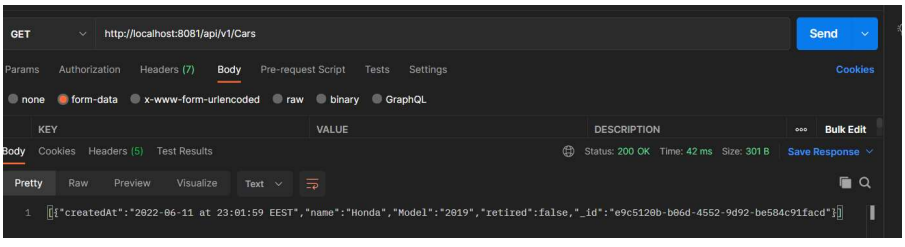
Endpoint to test api connection will return all pc data.

/api/v1/schemas

Endpoint will return all schemas in the system.

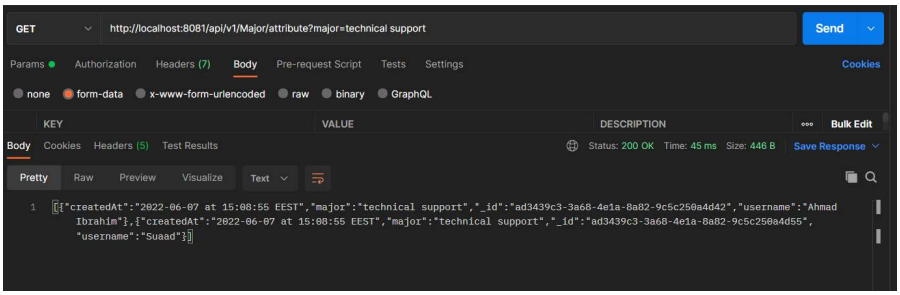
/api/v1/{schema} → schema name

Get data by schema name.



/api/v1/{schema}/attribute

Get data by schema attribute.



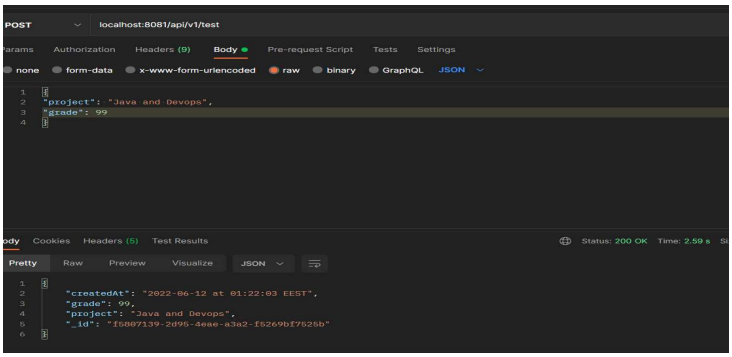
POST endpoints:

/api/v1/login

Endpoint for login and functionality already described.

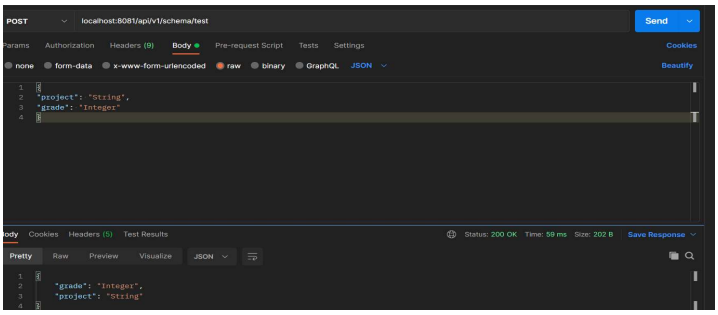
/api/v1/{schema}

Endpoint to add record to already added schema with body as json for record.



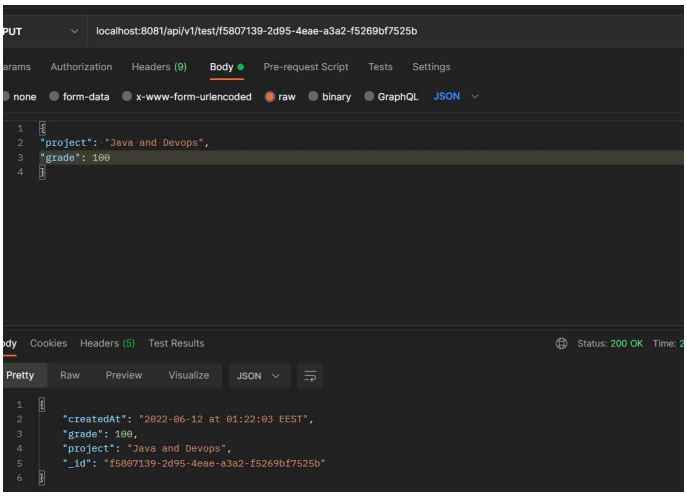
/api/v1/schema/{schema}

Endpoint to create new schema for specific attribute.



PUT endpoints:

/api/v1/{schema}/:id



DELETE endpoints:

/api/v1/{scheam}/:id