DAYANANDA SAGAR COLLEGE OF ENGINEERING

(An Autonomous Institute affiliated to VTU, Belagavi, Approved by AICTE & ISO 9001:2008 Certified)

Accredited by National Assessment & Accreditation Council (NAAC) with 'A' grade, Shavige

Malleshwara Hills, Kumaraswamy Layout, Bengaluru-560078.

Project Report

On

"Unified Database API"

"A unified REST API for relational and NoSQL datastores"

Submitted By

Amit Kumar Gupta [1DS16CS704]
Mohammed Ataaur Rahaman [1DS16CS721]
Sanjitha Singh A [1DS16CS739]

[Eighth Semester B.E (CSE)]

Under the guidance of
Mr. Ravichandra H.
Assistant Professor
Dept. of CSE
DSCE, Bangalore

Department of Computer Science and Engineering
Dayananda Sagar College of Engineering
Bangalore-78

VISVESVARAYA TECHNOLOGICAL UNIVERSITY

DAYANANDA SAGAR COLLEGE OF ENGINEERING

Shavige Malleshwara Hills, Kumaraswamy Layout, Bangalore - 560078

Department of Computer Science & Engineering

CERTIFICATE

This is to certify that the project entitled **Unified Database API** is a bonafide work carried out by **Amit Kumar Gupta [1DS16CS704], Mohammed Ataaur Rahaman [1DS16CS721] and Sanjitha Singh A [1DS16CS739]** in partial fulfilment of 8th semester, Bachelor of Engineering in Computer Science and Engineering under Visvesvaraya Technological University, Belgaum during the year 2019-20.

Mr. Ravichandra H (Internal Guide) Professor, Department of CSE, DSCE	**Dr. Ramesh Babu** Vice Principal & Head of Department, CSE, DSCE	**Dr. C P S Prakash** Principal, DSCE
Signature:.....	Signature:.....	Signature:.....

Name of the Examiners:	Signature with date:
1.....	.....
2.....	.....

Abstract

In the last few years, cloud computing has gained a lot of traction. It has spearheaded a huge paradigm shift in the IT industry with respect to management of resources. Computing resources from storage to servers are offered on a pay-per-use basis. With the virtualization of resources and the pricing policies, development and deployment of applications over cloud has become popular by the means of Platform-As-A-Service (PAAS) architecture.

The emergence of cloud computing has also led to the production of a huge amount of data; therefore, a vast array of data-stores to store, access and manipulate this data. Each datastore is specialized for a specific type of data. For a truly dynamic and responsive application, multiple datastores are often used, since a single datastore cannot meet the requirements of complex applications. Each of these datastores has its own data representation model, query language, and interfacing APIs. Learning the semantics of each of these datastores and the syntax of their API is a challenging task for any programmer.

In addition to this, different vendors offering PAAS often provide different datastores. Anytime a developer migrates to a different vendor, he must migrate his data from the existing data store to a new unfamiliar one.

A simple solution to this heterogeneity is a common unified interface that enables the programmer to perform all the necessary database operations using a simple query language. Doing this decouples the actual database operations from the low-level implementation details of the numerous APIs, thus providing a comfortable level of abstraction for the user to work with.

Acknowledgement

It has been a great opportunity to work in the field of Big Data and Database Management where we explored our knowledge of Big Data and how it is very much important in today's world. It wouldn't be possible to work on this project "Unified Database API" without our guide professor **Mr. Ravichandra H** Assistant Professor CSE, DSCE who is an expert in Big Data. We cane a long way designing the complete project to the best way possible keeping in mind the specifications and future scope.

Also, I express my profound thanks to **Dr. Vindhya Malagi** Associate Professor, CSE, DSCE to frame and guide the entire process from helping us understand our project and working towards organizing and planning our every step of actions.

We are also very grateful to our respected Vice Principal, HOD of Computer Science & Engineering, DSCE, Bangalore, **Dr. Ramesh Babu D R**, for his support and encouragement.

We would like to thank and take this opportunity to express our gratitude to **Dr. CPS Prakash**, Principal of DSCE, for permitting us to utilize all the necessary facilities of the Institution.

Lastly, we would also like to thank our parents and friends who have helped us a lot in the project throughout the semester with valuable inputs and motivation. Finally we would like to thank God Almighty for giving us strength, knowledge, ability and opportunity to undertake this research oriented project and to persevere and complete it satisfactorily. Without his blessings, this achievement would not have been possible.

Amit Kumar Gupta [1DS16CS704]

Mohammed Ataaur Rahaman [1DS16CS721]

Sanjitha Singh A [1DS16CS739]

Unified Database API

Ataago

May 2020

List of Figures

List of Tables

Listings

Contents

[This page is intentionally left blank]

Chapter 1

Introduction

1.1 Unified Database API

Unified Database API - UDAPI - is a uniform API that provides a uniform language based on REST requests for interaction with different types of databases. The databases could be relational like MySQL and PostGres; or non-relational like MongoDB and Couch DB.

1.2 Goals of the project

The project has two primary goals: One is to build UDAPI - a cross-database API which takes REST Requests as input and performs the required CRUD (Create, Retrieve, Update and Delete) operations by interacting with the necessary databases. The REST requests are parsed into their respective query languages and the database-specific APIs are called. The API returns the results of the query.

Building UDAPI decouples the query languages from the datastore. It allows migration of databases from one database to another without the overhead of learning the nuances of the new datastore, the lingo, the API syntax and so on

The other goal was to build a sample front-end to demonstrate the functionality of the API. The front-end includes tools for viewing all the databases of a specific user, the entity sets of each database, the entities of each entity set.

1.3 Beneficiaries

The API can be used for Platform-As-A-Service (PAAS) applications by developers. The barebones API can be hit directly with the REST request and the results returned can be directly used in their applications.

The user-interface also allows programmers to view and monitor their different databases of different programming language in a simple dashboard. There are quick options to create databases, delete databases. CRUD operations can be performed for entities and entity sets from the front-end without having to write code. This will increase a programmer's efficiency.

The primary beneficiaries of the project are developers.

1.4 Scope of the project

The API only interacts with two primarily kinds of databases : MySQL and MongoDB.

Therefore the project's goals would be to allow CRUD operations only for these two data-stores. But the API has been built modularly to allow integration of various other data-stores.

Certain database operations such as JOINs are not allowed because they are not supported in NoSQL databases.

1.5 Approach

The API was first prototyped using the Spring framework - a java-based application framework. There were certain limitations which caused it to be migrated to Flask framework - a python-based micro web framework. The Flask framework handles the routes of the API and performs the necessary processing and query translation. The processed and translated queries are wrapped and the low-level communication API for the different databases are invoked. There are numerous other steps that are involved in processing the requests which will be detailed in the other sections of the report. To provide security, authorisation and authentication features are included in the API by means of login. Given below is a block Diagram of the UDAPI System:

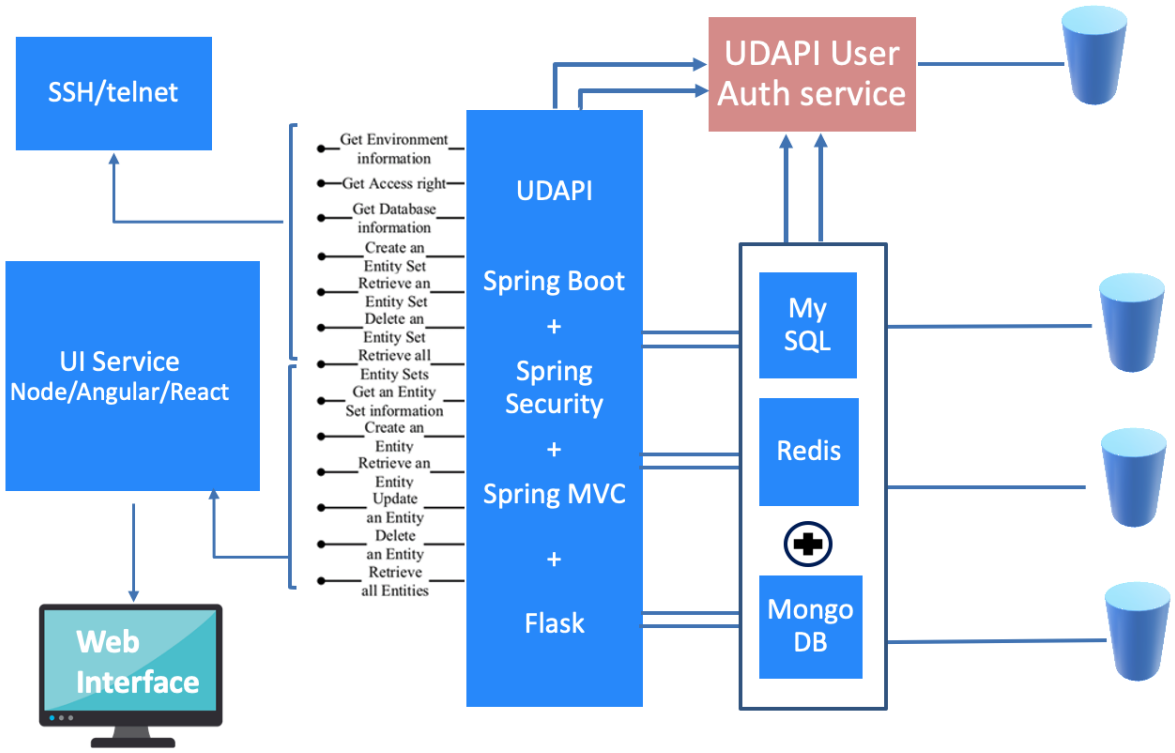

Figure 1.1: UDAPI Block Diagram

The rest of the report deals with the design and development process of UDAPI, the shortcomings, reflections and future work.

Chapter 2

Literature Survey

2.1 Background

With modern applications that collect not only a large amount and large volume of data, but also collect and store a large variety of data, a single database storage system does not suffice. Different types of data are assessed and each of these have different transactional requirements, such as some data needs high availability and some needs to be highly consistent in real time. This cannot be done using the traditional method of having only one type of data store.

This scenario gives rise to a concept called Polyglot Persistence. Polyglot Persistence means that when storing data, it is best to use multiple data stores, chosen based upon the type of data, the static and dynamic specifications of storing and handling data along with the operational requirements.

Having different data stores for different data may introduce the problem of learning and maintaining different technologies but that is well worth it. For example, sensitive data that needs to be highly available would use a relational data store, whereas data which could need eventual consistency would use MongoDB for example.

2.2 RDBMS and Non-RDBMS

One of the main issues of using multiple data-stores is the imminent difference between RDMS and Non-RDBMS databases. The RDBMS databases are a far more structured database. They are rigid. They have fixed schemas. The data is represented in terms of tables, attributes and records. The attribute denotes the type of data and the records contain the value themselves. RDBMS provides ACID properties : Atomicity, Consistency, Isolation and Durability. RBMS is transactional in nature. It is still a primary choice for critical applications. It uses a structured query language called SQL which helps in extracting data and writing complex queries.

Non-RDBMS on the other hand does not have a fixed schema. Data can be stored in different ways. For example, Key-Value pairs, Documents, graphs. They work on CAP properties: Consistency, Availability and Partition-tolerance. These datastores are vertically scalable and are a lot more robust than RDBMS data-stores. It however has a few limitations

in terms of handling complex queries. NoSQL databases are usually open-source which has its own obvious advantages and disadvantages as with anything open-source. For example, cost-efficiency is an advantage but there is no standardisation.

Using both of these could help developers leverage their advantages while letting them minimise the disadvantages.

2.3 REST Requests

REST stands for Representational State Transfer.

It is not a protocol but more of an architectural style for providing communication standards between different computer systems. RESTful applications are stateless. The client and server implementations are done independently and irrespective of each other, so any changes made on either of these ends will not affect the operations of the other component. This increases modularity and lets independent evolution and development of the client and server.

REST requests work on endpoints which are the primary sources of communication between the client and the server. The REST requests occur in a request response fashion. The REST requests are HTTP requests and they contain the following components : a HTTP verb, a request header, a request body and the endpoint.

There are four HTTP verbs:

- **GET** - to retrieve resources
- **POST** - to create a new resource
- **PUT** - to modify an existing resource
- **DELETE** - to delete an existing resource

The request header contains information that is important to the server. The request body contains additional information.

Once the request is processed, the server sends the response. The response contains a status code representing the status of the request whether it was processed accurately or if there was any problem. If there is a payload, the server also includes the content type in the response headers.

This stateless communication process makes UDAPI a light-weight ideal solution independent of the client and server development.

2.4 Cloud Computing

Cloud computing has gained popularity in the last decade. It offers computing resources from storage to servers on a pay-per-use basis. It essentially deals with virtualisation of resources.

Cloud service is provided in three different ways:

1. **Infrastructure-As-A-Service (IAAS)**

   IAAS delivers services like pay-as-you-go cloud computing infrastructure, servers, network, operating systems, and storage, through virtualization technology. This is the most flexible cloud service, but a lot of it relies on the users to manage and configure.

2. **Platform-As-A-Service (PAAS)**

   PAAS is primarily used by developers since it provides a platform for software creation . This means developers don't need to start from scratch when developing applications, giving them the freedom to focus on building the software without worrying about operating systems, software updates, storage, or infrastructure.

3. **Software-As-A-Service (SAAS)**

   SAAS is the most commonly provided cloud service. A pre-developed application is hosted via a cloud for the user to use as they go. They run on the client side without worrying about installing dependencies or any other software of any kind.

UDAPI is primarily aimed at helping PAAS developers. PAAS applications provide a predetermined database. Anytime a developer changes the PAAS provider they will be potentially migrate to a different database they aren't even familiar with. This causes what is known as vendor lock-in : the developer is forced to use a vendor despite being dissatisfied with certain aspects of the service.

UDAPI would eliminate this concern. The user would only need to be familiar with the context of the API. He could use it for any database and could migrate to any provider without hesitation since the API will do all the heavy-lifting and low-level tasks for the developer, including but no limited to migration of databases.

2.5 Big Data

Big data is a term that is used to describe data that is too large, too fast or too complex for traditional database management stores. This data can either be structured or unstructured.

There are three Vs of big data:

1. **Volume:** Large volumes of data is being processed every second from social media feed, IoT devices, business transactions and so on. This vast amount of data is overwhelming for any traditional datastore.
2. **Velocity:** The speed at which the data arrives is overwhelming for traditional data-stores.
3. **Variety:** Data can be structured or unstructured. Not just that, data can be in various different forms, from text to video. All this data has to be handled efficiently.

Variety is where UDAPI finds its use-case. As mentioned under polyglot persistence, different data-stores are well suited for different applications. And it makes a lot of sense to leverage the advantages of these databases instead of settling for something that is based on a compromise. UDAPI makes the process of handling different datastores easier for the developers and saves precious development time and operational cost.

Chapter 3

System Specification

3.1 Specification

The system specification can be illustrated by the following Use case Diagram.

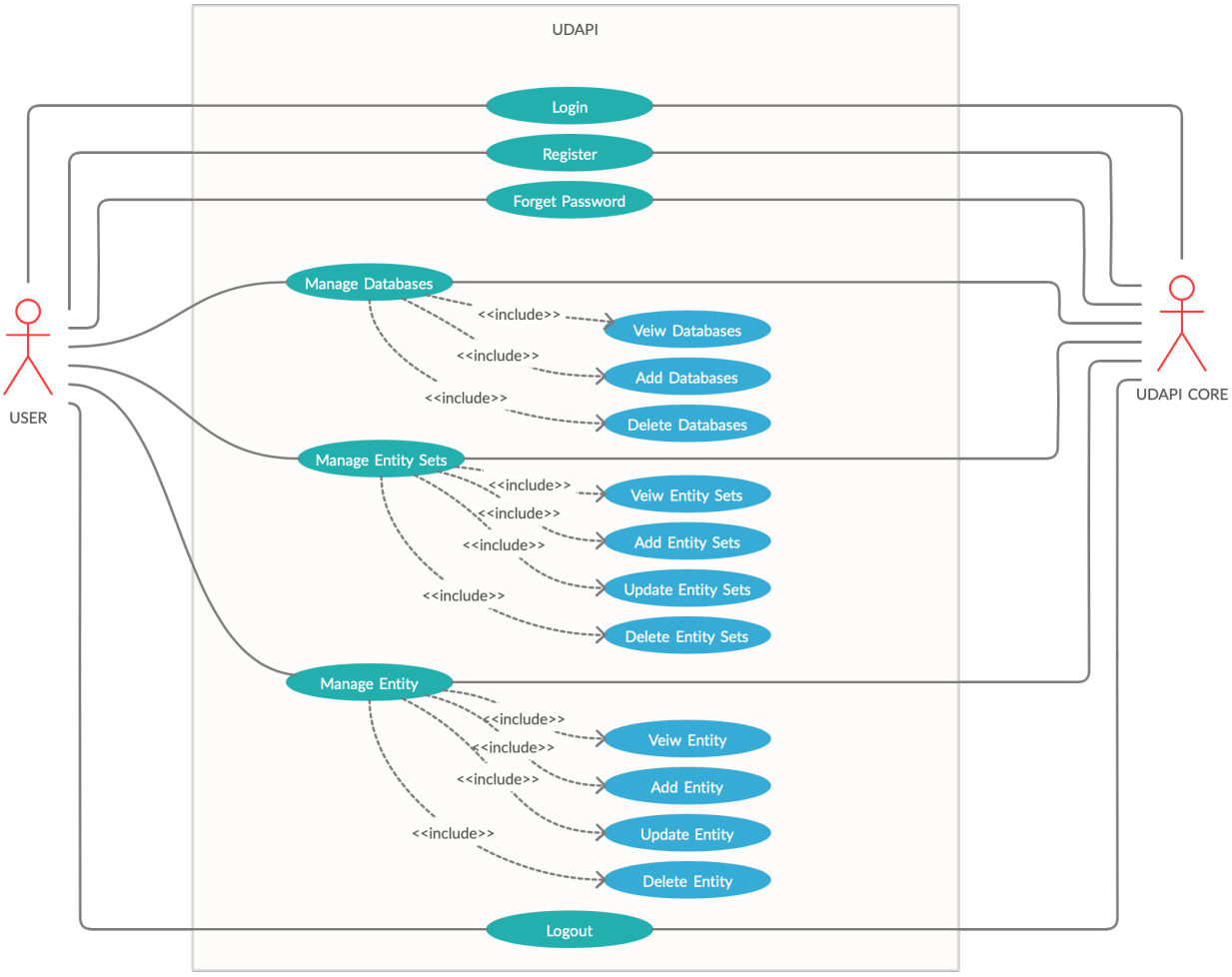

Figure 3.1: UDAPI Use Case Diagram

- Unified Database API is a RESTful API, meaning that it uses REST requests and responses for communication with the client.
- The API allows for a register feature which allows the users to register themselves to use the API. This is essential for providing security vis-a-vis authentication and authorisation.
- The API allows for a login feature which allows the user to login through a username and password ensuring security vis-a-vis authentication and authorisation.
- The API will take one of the four primary HTTP requests : GET, POST, PUT and DELETE.
    1. The GET requests will solely be used for fetching and viewing resources.
    2. The POST requests will solely be used for creating resources.
    3. The PUT requests will solely be used for updating existing resources.
    4. The DELETE requests will solely be used for deleting existing resources.
- These four requests will be responsible for overall handling of CRUD operations.
- The API will use uniform terminologies as a substitute for different database-specific terminology.
- The API will allow Create operations on Databases, Entity Sets and Entities.
- The API will allow Retrieve operations on Databases, Entity Sets and Entities.
- The API will allow Update operations on Databases, Entity Sets and Entities.
- The API will allow Delete operations on Databases, Entity Sets and Entities.
- The API will be accompanied with a front-end web application that will demonstrate the communication interface.
- The web application will include means to perform all the necessary operations on Databases, Entity Sets and Entities in a simple, easy-to-use interface.
- The application will allow Create operations on Databases, Entity Sets and Entities.
- The application will allow Retrieve operations on Databases, Entity Sets and Entities.
- The application will allow Update operations on Databases, Entity Sets and Entities.
- The application will allow Delete operations on Databases, Entity Sets and Entities.
- The web application will provide a necessary layer of abstraction for users who are not familiar with direct API interactions.
- The web application will be highly-responsive and have an intuitive design.
- There will be no noticeable delay or latency while using either the API or the application for CRUD operations.
- The requests will be simple and mindful of the REST requirements so the developer can get started in a matter of minutes.
- The API will provide proper encapsulation to prevent unauthorized access and misuse by binding the users and their datastore together.

Chapter 4

System Architecture

4.1 Authentication and Authorization

Authentication is the process of verifying the user's identity.

Authorization is the process of verifying what operations the user has the privileges to perform.

In UDAPI, authentication is done in two steps.

1. **Creating an account:**

   The user can register for an account with his email ID, a username and a password. There are measures on ensuring that there are no duplicate usernames or emails. Once the user is registered successfully, he has his own account that he can access by logging in. All the REST routes for performing operations require that the user is logged in. A user who has not registered cannot perform any operations.

2. **JWT tokens:**

   JSON Web Token (JWT) defines a compact and self-contained way for securely transmitting information by means of a JSON object. In addition to regular credentials, JWT provides additional authorisation benefits. They can also be used for non-repudiation as these are digitally signed. When the user logs in successfully, a unique JWT will be returned. This is required as part of there request headers for performing any operations in the API. The user will be able to access routes, services, and resources without having to login repeatedly.

If the application is used, JWT is handled internally.

4.2 Uniform Terminologies

There are a few key terminologies that UDAPI uses to ensure uniformity across different databases.

1. **Database:**

   Databases are the primary units that UDAPI uses to store its data. An example of a database would be a college database. Each database is made of multiple entity sets.

2. **Entity Set:**

   An entity set is a unit of database. It is called a table in MySQL and collection in MongoDB. In the college database, entity sets would be professors, students, courses, departments and so on.

3. **Entities:**

   Entities are records containing the values and information. They are stored in Entity Sets. They have attributes that define the nature of data they store. In the college example, a student : "USN" : "1DS16CS000", "Name" : "John Doe", "Department" : "Computer Science" would be an example of an entity in the entity set students.

UDAPI Terminology	MySQL	MongoDB
Databases	Databases	Databases
Entity Sets	Tables	Collections
Entities	Rows	Documents

Table 4.1: UDAPI Terminology

Figure 4.1: UDAPI Resources Model

4.3 RESTful Routes (JSON Agreement)

The description of the routes along with their HTTP request verb, the request header, the request body and the expected response is detailed below:

4.3.1 Authentication

Register

Register a new user

- **Request: POST**
- **Endpoint:** /register
- **Header:**
- **Body:**

```
{
  "email":"ataa2@gmail.com",
  "username":"Ataa2",
  "password":"s ",
  "confirm_password":"s"
}
```

Listing 4.1: Register Body

- **Response:**

```
{
    "message": "ataago.go@gmail.com successfully registered as Ataa!",
    "success": 1
}
```

Listing 4.2: Register Response

Reset Password

Change the Password

- **Request: POST**
- **Endpoint:** /resetpassword
- **Header:**
- **Body:**

```
{
  "email":"ataago7@gmail.com",
  "new_password":"ata",
  "confirm_password":"ata"
}
```

Listing 4.3: Reset Password Body

- **Response:**

```
{
    "message": "Password updated successfully.",
     "success": 1
}
```

Listing 4.4: Reset Password Response

Login

Login with your username and password

- **Request: POST**
- **Endpoint:** /login
- **Header:**
- **Body:**

```
{
        "username": "Ataa",
        "password": "abcdedf"
}
```

Listing 4.5: Login Body

- **Response:**

```
{
    "jwtToken": "eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJ1c2VybmFtZSI6IkF0YWEiLCJleHAiOjE1ODkwOTI5NTB9.PhuOCh5zN8tWh2195OB7TZeBDGETyJolYfHcKx2R__k"
}
```

Listing 4.6: Login Response

4.3.2 Databases

View Databases

List all the databases with <databaseType> : all, mysql, mongodb

- **Request: GET**
- **Endpoint:** /<databaseType>/databases
- **Header:** jwtToken
- **Body:**
- **Response:**

```
{
   "databases": {
       "mongodb": [
           "database6",
           "database7"
       ],
       "mysql": [
            "database1",
            "database2",
           "database4",
       ]
    },
    "success": 1
}

```

Listing 4.7: View Databases Response

Create Databases

Create a new database and add in configs

- **Request: POST**
- **Endpoint:** /<databaseType>/databases
- **Header:** jwtToken
- **Body:**

```
{
  "databaseName":"gomysql9"
}

```

Listing 4.8: Create Databases Body

- **Response:**

```
{
    "message": "Database created successfully.",
    "success": 1
}

```

Listing 4.9: Create Databases Response

Delete Databases

Delete the Database and remove it from configs

- **Request: DELETE**
- **Endpoint:** /<databaseType>/databases/<databaseName>
- **Header:** jwtToken
- **Body:**
- **Response:**

```
{
    "message": "Database: gomysql1 deleted successfully.",
    "success": 1
}

```

Listing 4.10: Delete Databases Response

4.3.3 Entity Sets

View Entity Sets

View all the Entity Sets in the Database

- **Request: GET**
- **Endpoint:** /<databaseType>/databases/<databaseName>
- **Header:** jwtToken
- **Body:**
- **Response:**

```
{
    "entitySets": [
        "dept",
        "emp",
        "salgrade"
  ],
    "success": 1
}

```

Listing 4.11: View Enity Sets Response

Create Entity Sets

Create a new entity set in the databaseName

- **Request: POST**
- **Endpoint:** /<databaseType>/databases/<databaseName>
- **Header:** jwtToken
- **Body:**

```
{
      "entitySetName": "student2",
     "attributes": {
            "id": {
                  "DataType": "INT",
                  "PK": 1,
                  "NN": 1,
                  "AI": 1
            },
             "studname": {
                  "DataType": "VARCHAR(20)",
                 "PK": 1,
                 "NN": 1,
                 "AI": 0
            },
             "grade": {
                  "DataType": "VARCHAR(2)",
                  "PK": 0,
                  "NN": 0,
                  "AI": 0
             }
     }
}


```

Listing 4.12: Create Entity Set Body

- **Response:**

```
{
  "message": "Table Created",
    "success": 1
}

```

Listing 4.13: Create Entity Set Response

Edit Entity Sets

Edit entity set name in the databaseName

- **Request: PUT**
- **Endpoint:** /<databaseType>/databases/<databaseName>/<entitySetName>
- **Header:** jwtToken
- **Body:**

```
{
  "newEntitySetName": "newTableName"
}

```

Listing 4.14: Edit Entity Set Body

- **Response:**

```
{
  "message": "Table Name Altered",
  "success": 1
}

```

Listing 4.15: Edit Entity Set Response

Delete Entity Sets

Delete entity set in the databaseName

- **Request: DELETE**
- **Endpoint:** /<databaseType>/databases/<databaseName>/<entitySetName>
- **Header:** jwtToken
- **Body:**
- **Response:**

```
{
  "message": "Table newTableName deleted.",
  "success": 1
}

```

Listing 4.16: Delete Entity Set Response

4.3.4 Entity

View Entity

View all the entites in the entitySetName

- **Request: GET**
- **Endpoint:** /<databaseType>/databases/<databaseName>/<entitySetName>
- **Header:** jwtToken
- **Body:**
- **Response:**

```
{
    "message": [
     {
          "deptno": 10,
          "dname": "Accounting",
          "location": "New York"
      },
      {
          "deptno": 20,
          "dname": "Research",
          "location": "Dallas"
      },
    ],
    "success": 1
}

```

Listing 4.17: View Entity Response

Add Entity

Create new entity for the entitySetName

- **Request: POST**
- **Endpoint:** /<databaseType>/databases/<databaseName>/<entitySetName>
- **Header:** jwtToken
- **Body:**

```
{
      "entity": {
            "deptno": 12,
            "dname": "Data Scienc",
            "location": "Bangalore"
      }
}

```

Listing 4.18: Add Entity Body

- **Response:**

```
{
    "message": "Inserted into dept successfully.",
    "success": 1
}

```

Listing 4.19: Add Entity Response

Edit Entity

Edit the entity for the entitySetName

- **Request: PUT**
- **Endpoint:**
  /<databaseType>/databases/<databaseName>/<entitySetName>/<primeAttributeName>
- **Header:** jwtToken
- **Body:**

```
{
    "primeAttributeValue": "12",
      "attributeName": "dname",
    "attributeValue": "Data Science"
}

```

Listing 4.20: Edit Entity Body

- **Response:**

```
{
    "message": "updated value of deptno ('12').",
    "success": 1
}

```

Listing 4.21: Edit Entity Response

Delete Entity

Delete an entity for the entitySetName

- **Request: DELETE**
- **Endpoint:**
  /<databaseType>/databases/<databaseName>/<entitySetName> /<primeAttributeName>
- **Header:** jwtToken
- **Body:**

```
{
  "primeAttributeValue":12
}

```

Listing 4.22: Delete Entity Body

- **Response:**

```
{
    "message": "Deleted entity having deptno ('12').",
    "success": 1
}

```

Listing 4.23: Delete Entity Response

4.3.5 Config Data

Schema Details

Get the schema details of all the databases for the user.

- **Request: GET**
- **Endpoint:** /<databaseType>/databases/<databaseName>/schema/<entitySetName>
- **Header:** jwtToken
- **Body:**
- **Response:**

```
{
    "databaseType": "mysql",
    "entitySetName": "newStudent",
    "primary_key": "id",
    "schema": {
        "grade": "VARCHAR(2)",
         "id": "VARCHAR(20)",
        "studname": "VARCHAR(20)"
    }
}
```

Listing 4.24: Schema Details Response

4.3.6 User

User Details

Get the user details who has logged in

- **Request: GET**
- **Endpoint:** /user
- **Header:** jwtToken
- **Body:**
- **Response:**

```
 {
     "success": 1,
     "users": [
      {
           "admin": 1,
           "email": "ataago7@gmail.com",
           "username": "Ataago"
       }
   ]
 }
```

Listing 4.25: User Details Response

4.3.7 Admin

All Users

Get all the users registered in UDAPI

- **Request: GET**
- **Endpoint:** /users
- **Header:** jwtToken
- **Body:**
- **Response:**

```
{
    "success": 1,
    "users": [
      {
          "admin": 1,
          "email": "ataago7@gmail.com",
          "username": "Ataago"
      },
      {
         "admin": 0,
          "email": "ataa@gmail.com",
          "username": "ataa"
      }
    ]
}

```

Listing 4.26: All users Response

Chapter 5

Backend Implementation

5.1 Spring Implementation of UDAPI

We have chosen to make use of the Spring Framework for our UDAPI project. The spring framework is a suite of libraries, frameworks and design patterns for enterprise Java, which is now also extended to C-Sharp, Kotlin, etc. The Spring framework is arguably the most popular Java framework to date and offers continuous support and development by always adding new features and making enterprise code more maintainable, easier to develop, and helps in using industry best practices. The Spring framework is largely based on the Java EE (Enterprise Edition) specification, with some modules deviating in favour of better design choices. The framework is a large collection of multiple modules and a core framework which introduces the concepts of Dependency Injection and Inversion of Control. These are two popular design patterns specified by the gurus of design patterns, the Gang of Four.

We have decided to use the Spring Framework's offerings of DI (Dependency Injection) and IOC (Inversion of Control), as we needed our application to be tested easily and also have minimal boilerplate code. As our application deals with a lot of code weaving with translating our own querying definitions to that of the target database, the codebase was bound to get messy. By using the Inversion of Control design pattern, we eliminate that problem and hand over control of creating and managing the lifecycle of objects to something called the IOC container.

This IOC container is not created by us, but the Spring Framework itself. All we had to do was specify what beans(Objects) we want and when we need them, spring provides them for us at runtime. Some of the examples where we used this concept was the Database Driver Factory, where we need to automatically determine which database driver flow we need to choose based on the database type header. If we were to write the code for this logic manually, it would contain multiple "if-else" ladders and that is not clean and maintainable code. We also use this concept of IOC with separating out UDAPI into different layers such as the Persistance layer, the Business layer, etc.

IOC also allows us to control the scope of our beans such as providing a new instance of the bean every single time (PROTOTYPE) or the same instance of the bean every time it is requested for the lifecycle of the application (SINGLETON). By doing this, we have fine grained control of our application code logic, and we don't have to worry about instantiating

the objects. Dependency Injection is the other feature of Spring core and it goes hand in hand with the IOC framework. It is the part of spring which allows us to inject objects into methods and other @Autowired fields dynamically during runtime. While designing and writing the UDAPI application, we realised that we would need to do a lot of testing and have to test at different layers of our application such as the persistence layer directly due to the nature of our application. This is where the Dependency Injection comes in, helping us in injecting Mock testing objects without having to go through the entire application stack. This has saved us a lot of time, as well as helped us build a robust and testable application. We used a framework called Mockito which is used to create Mock objects to our Spring applications for testability. Examples of mock test objects we used across layers are Json objects provided to different database driver query converters which convert the json object to their queries. If we didn't have this, we would have to test by making a HTTP API call along with JWT tokens and other headers every time as it's normal use case would be. This makes the whole development process faster.

Other than Spring Core, the spring framework has a vast number of other modules in it's suite and it keeps adding more, and refining more. Some of the modules are Spring MVC, Spring JUnit, Spring Security, Spring Data JPA, Spring Data Redis, Spring Boot, Spring Thymeleaf, and many many more.

The modules we used in UDAPI are

1. Spring Boot
2. Spring MVC (Web)
3. Spring JUNIT
4. Spring Data JPA
5. Spring IOC and DI (core)
6. Spring Jackson (For JSON parsing)

Spring Boot offers many advantages such as autoconfiguration, automatic annotation support, Starter dependencies, and a command line interface tool suite. It helps in setting up a fully featured enterprise ready microservice based enterprise project in a matter of minutes. It enforces the Spring Frameworks motto of using the Convention over manual configuration. Only the elements which explicitly needs to be explicitly different from the enterprise standards needs to be specified. It also helps in auto managing the versions of dependencies of other spring modules that we need. This helps in making our application future proof as newer spring modules are always developed with backwards compatibility in mind.

Spring Web (MVC) helps us in creating a fully featured REST based application which is based on the Model View Controller design pattern. It uses Java's enterprise servlet specifications underneath and abstracts it to give us an MVC based pattern. This helps in offering great flexibility and extensibility. The web framework offers everything a web based REST application might need, from web sockets to serving dynamic JSP pages. It also supports a wide array of multimedia formats (multipart, video streaming, etc).

Spring JUnit framework is used to write efficient and robust test cases for our enterprise application, extending and working along the advantages provided by Spring IOC and DI.

Spring Data JPA is a powerful framework to interact with data sources and uses an ORM tool such as Hibernate underneath. It abstracts away a lot of the boilerplate code required with typical ORM tools, and makes handling transactions and complex queries very easy. We use Spring Data JPA to interact with UDAPI's MySql data store to be able to store metadata and used related information. To talk to the actual database drivers however, we opted to use the low level Database driver classes provided by the database vendors for greater control. Transaction control and management is as easy as annotating the transactional methods with @Transactional, however we can get very involved, specific and have a lot of control over every small aspect, which again showcases convention over configuration approach of the spring framework. Data JPA also provides its own Querying language called JPQL which intelligently uses Code defined entity classes.

5.2 Flask Implementation of UDAPI

Flask is a micro web framework written in Python and as such makes use of python's readable, concise coding language. It is a micro-framework because unlike other frameworks Flask requires very few tools, dependencies and library. Unlike stricter frameworks like Ruby on Rails, Flask offers a high level of flexibility. By default, a flask application is very bare-bones. But there are various third-party libraries that are used to make it a useful framework. Two of the main components that are used along with Flask for UDAPI are Werkzeug and Jinja. These were used for the front-end server, and the details will be discussed in the Flask Implementation of Front-end Server section of this report.

The API code was divided into three sections:

1. Generic code that was applicable for all databases
2. MySQL specific code
3. MongoDB specific code

Directory Structure:

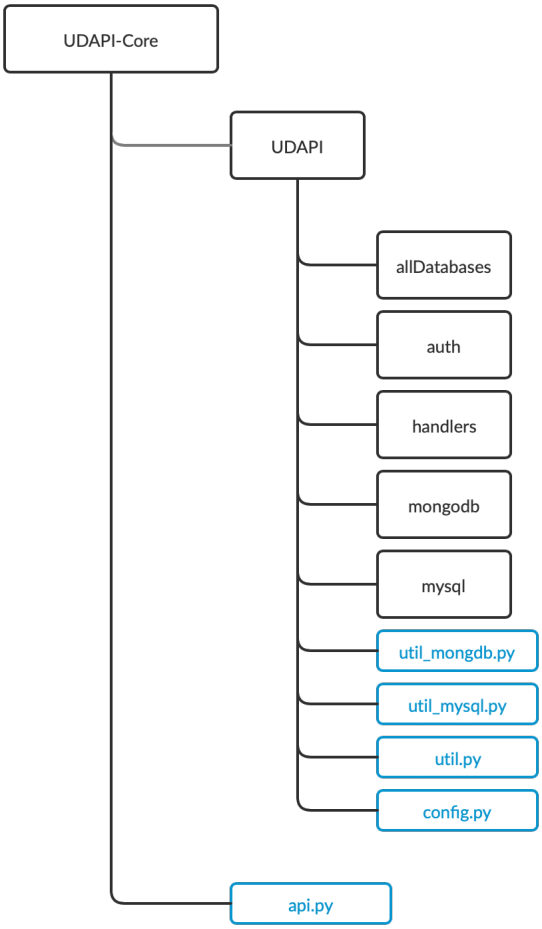

Figure 5.1: UDAPI Core Directory Structure

Anytime a request was made to any of the routes, an initial check would be made for JWT. If there was JWT in the header, the request would go through, or an error would be raised. The JWT was checked using a python decorator called tokenrequired. Python decorators are functions that wrap other function and modify some aspect of the wrapped function. Given below is the wrapper function for this:

```
def token_required(f):
    @wraps(f)
    def decorated(*args, **kwargs):
        token = None

        if 'jwtToken' in request.headers:
            token = request.headers['jwtToken']

        if not token:
            return jsonify(success=0, error_code=401, message="JWT Token is missing!"), 401
        try:
            jwtData = jwt.decode(token, SECRET_KEY)
        except:
            return jsonify(success=0, error_code=401, message="JWT Token is invalid."), 401

        try:
            cnx = mysql.connector.connect(
                host="localhost",
                user="root",
                passwd="password",
                database="udapiDB"
            )
            mycursor = cnx.cursor()
            sql = "SELECT * FROM udapiDB.users WHERE username='" + jwtData['username'] + "';"
            mycursor.execute(sql)
            entities = mycursor.fetchall()
            attributes = [desc[0] for desc in mycursor.description]
            data = []
            for entity in entities:
                data.append(dict(zip(attributes, entity)))
            cnx.close()
            if not data:
                return jsonify(success=0, error_code=401, message="JWT Token is invalid."),
    401
        except mysql.connector.Error as err:
            return jsonify(success=0, error_code=err.errno, message=err.msg)
        return f(jwtData['username'], *args, **kwargs)
    return decorated
```

Listing 5.1: JWT Token Wrapper function

Once the JWT was verified, the respective endpoint would be hit with the request based on the database.

5.2.1 MySQL Implementation

To establish communication from python to MySQL database we need a MySQL driver. We have used the driver "MySQL connector" which has all the basic and essential tools for working with MySQL from Python.

Firstly establishing a connection to the MySQL database we need to specify the username and password also specifying the host.

```
import mysql.connector

mydb = mysql.connector.connect(
  host="localhost",
  user="yourusername",
  passwd="yourpassword"
)

print(mydb)
```

Listing 5.2: Example to create a connection to MySQL server

All the details of the CRUD operations are explained in the following pages.

Databases

1. **Creating a Database**

   Once a connection is established, the process of creating a database is simple. The following example shows how to create a mydatabase

```
import mysql.connector

mydb = mysql.connector.connect(
  host="localhost",
  user="yourusername",
  passwd="yourpassword"
)

mycursor = mydb.cursor()
mycursor.execute("CREATE DATABASE mydatabase")
```

Listing 5.3: Creating a Database example

   In our API we store all the databases created in a separate database in MySQL named as "udapiDB". This database has 2 tables namely, "configs" and "users".

   We use configs table to store and update all the new databases created by a particular user having attributes as, "username", "databaseName" and "databaseType". This configs table helps us to keep track of all the databases created in not only mySQL but also other relational or non relational databases connectivity added in future. This way udapi stands very extensible and scalable for further modules to be added.

   The other user table is used to store all the user credentials and details at the backend server. These two tables in udapiDB can only be accessed with admin connection which is the "root" username.

2. **Viewing all databases**

   Retriving and viewing all the databases was fairly simple and easy as we can fetch all the databases created by an user just by a simple query for the metadata table "udapiDB.configs".

3. **Updating a database name**

   Directly updating a Database name is not possible in MySQL and hence this feature was not added.

4. **Deleting a database**

   Deleting a database in MySQL is done with a simple query by the user connection to the MySQL server. The query to delete the database is "DROP DATABASE <database-Name>;". Care should be taken after deleting a database that the details of the deleted database should be removed from "udapiDB.configs".

Entity Sets

1. **Creating an Entity Set**

   MySQL has a well defined schema to be declared first for a entity set called as a table schema. This could be done after establishing a connection for a user. Given below is a simple example demonstrating how to create a entity set "customers" with its attributes "name" and "address".

```
import mysql.connector

mydb = mysql.connector.connect(
  host="localhost",
  user="yourusername",
  passwd="yourpassword",
  database="mydatabase"
)

mycursor = mydb.cursor()
mycursor.execute("CREATE TABLE customers (name VARCHAR(255), address VARCHAR(255))")
```

Listing 5.4: Creating an Entity Set example

   We see that there are many restrictions and tight rules to be followed with respect to the schema of a relational database like MySQL, hence extra care was taken to dynamically create any entity set with users choice to create attributes with its attribute types.

2. **View all Entity Sets**

   The Entity sets (tables) can be viewed using a querry "SHOW TABLES;". Note that the user should establish the connection to the MySQL server with the users username and password.

3. **Updating Entity Set**

   Renaming a entity set "table" is fairly simple after a connection by the user has been established with a query "ALTER TABLE oldTableName RENAME newTableName;".

4. **Deleting Entity Set**

   Deleting a entity Set (table) in MySQL can be done by the same user connection with a simple querry "DROP TABLE tableName;".

Entity

It should be noted that to access any entity by the user, firstly we have to establish connection to the database with the username and password of the client. Then any such operation on the connected database on an entity can be performed as given below. But if the database doesn't exists for the user then we raise and exception which can be handled by mysql.connector.errorcode module.

1. **Creating an entity**

   Creating a new entity is a tricky one in MySQL, as care should always be taken while inserting the values into the table according to the schema as we can't have a different types of entities for each rows in the tables. This is done by a query "INSERT INTO tableName (attribute1, attribute2) VALUES (value1, value2);".

2. **View all entities**

   Before viewing all the entities, we should make sure that the entity set exists, this situation is easily handled by mysql.connector.errorcode. then we can view all the entities present in an entity set (table) as follows

```
sql = "SELECT * FROM " + entitySetName + ";"
mycursor.execute(sql)
entities = mycursor.fetchall()
```

Listing 5.5: Viewing Entities of and Entity Set

3. **Updating an entity**

   Before updating a given entity care should be taken if the enity set exists for the database connected. Then we use a key value to locate which entity is to be changed in the collection of entities. The query for this is "UPDATE entitySetName SET 'attributeName'='newValue' WHERE 'keyAttribute'='keyValue';"

4. **Deleting an entity**

   Deleting an entity is also similar to the updating entity where one should take care if the entity exists by using the key and its value for the entity. Then this is done by the query "DELETE FROM entitySetName WHERE 'keyAttribute'='keyValue';".

5.2.2 MongoDB Implementation

To establish communication from python to MongoDB, an object-relational mapper (ORM) called PyMongo was used. ORM is a technique that lets the developer query and manipulate data from a database using an object-oriented paradigm. PyMongo is a Python distribution that contains all the essential tools for working with MongoDB from Python.

To establish connection with the MongoDB database, the MongoClient function is used. By default it connects to 'localhost', port : 27017.

```
from pymongo import MongoClient
client = MongoClient()
```

Listing 5.6: Example to establish connection to the mongoDB server

All the details for the CRUD operations are explained as follows.

Databases

1. **Creating a Database**

   Once a connection is established, the process of creating a database is a simple process. The following example shows how to create a test-database.

```
client = MongoClient()
db = client['test-database']
```

Listing 5.7: Example to create a test-database

   But our approach involves adding the database config information to the metadata table. The database data was stored in a global Config table and a local api-config table. It was necessary to have a local config mongo collection since MongoDB create operations are Lazily-evaluated. This means that a database is not created until a document is inserted into at least one of its collections.

   MongoDB, however, allows creation of databases with the same name. The databases were therefore stored by prepending the username to the database name.

   For example, if a user called James created a database called Student, his database would be stored as James-Student. Similarly, if a user called Lily created a database called Student, her database would be stored as Lily-Student. However this only happened in the backend, providing a layer of abstraction.

2. **Viewing all databases**

   The databases were stored in a metadata table and a simple query to fetch the table with the current username was sufficient.

3. **Updating a database**

   Updating a database is not possible in MySQL, therefore this had to be removed from MongoDB as well to maintain the consistency of the API.

4. **Deleting a database**

   Deleting a database is possible through a straightforward command once the database connection is established. But care has to be taken to delete the database from the global and local config databases.

```
client = MongoClient()
client.drop_database('test-database')
```

Listing 5.8: Example to delete a test-database

Entity Sets

1. **Creating an Entity Set**

   Unlike MySQL, MongoDB does not have a schema. But, to accommodate for the schema of MySQL, it was essential to create an abstraction schema. A separate MongoDB collection called Schema was created to store all schemas of any created objects. Entity Sets are called collections in Mongo. To create a collection,

```
client = MongoClient()
db = client['test-database']
coll = db['test-collection']
```

Listing 5.9: Example to create an Entity Set

   In addition to this, a dummy document is created and consequently deleted to make sure the collection is created. This is done to overcome MongoDB's lazy evaluation.

   MongoDB does not place any restrictions on creating collections of the same name. So custom Exceptions had to be written to prevent duplicate Entity Sets and for non-existing Entity Set since MongoDB would automatically create an entity-set or database if it didn't exist.

2. **View all Entity Sets**

   The collection names can be viewed using

```
db.list_collections_names()
```

Listing 5.10: Example to view all Entity sets

3. **Updating Entity Set**

   To rename a MongoDB collection, the rename function would have to be called.

```
previousES.rename('newESName')
```

Listing 5.11: Example to update Entity Set Name

   But the EntitySet would have to be modified in the schema as well.

4. **Deleting Entity Set**

   The collection could be dropped using:

```
db.dropcollection(entitySetName)
```

Listing 5.12: Example to delete an Entity Set

   But, the entitySet had to be deleted from the schema as well.

Entity

1. **Creating an entity**

   Before inserting an entity, first checks would need to be performed on if the database and the collection exist. If they don't then an error can be raised. There is a check for existing entities. If they exist, then the entity is created using,

```
entitySet.insertone({entity})
```

Listing 5.13: Example to create an Entity

2. **View all entities**

   Before viewing all entities, first checks would need to be performed on if the database and the collection exist. If they don't then an error can be raised. If they exist, then the entity is created using the find() function and by passing it no parameters.

```
entitySet.find()
```

Listing 5.14: Example to view all Entities

3. **Updating an entity**

   Before inserting an entity, first checks would need to be performed on if the database, the collection and if the entity itself exists. If they don't then an error can be raised. If they exist, then,

```
entitySet.findoneandupdate({primarykey : primaryKey}, {"$set": newData})
```

Listing 5.15: Example to update an Entity

   Using $set is important to preserve existing data.

4. **Deleting an entity**

   Before inserting an entity, first checks would need to be performed on if the database, the collection and if the entity itself exists. If they don't then an error can be raised. If they exist, then,

```
entitySet.findoneanddelete({primarykey : primaryKey})
```

Listing 5.16: Example to delete an Entity

Chapter 6

Frontend Implementation

6.1 Flask Implementation of UDAPI User Interface

By default, a flask application is very bare-bones. But there are various third-party libraries that are used to make it a useful framework. Two of the main components that are used along with Flask for UDAPI are Werkzeug and Jinja.

- Werkzeug is a toolkit for Web Server Gateway Interface (WSGI) applications, and is the primary utility library.
- Jinja2 is a template engine that is used for rendering dynamic HTML pages.

Along with this, various other small libraries such as WTForms were used. The Directory structure for the Frontend UI is given below:

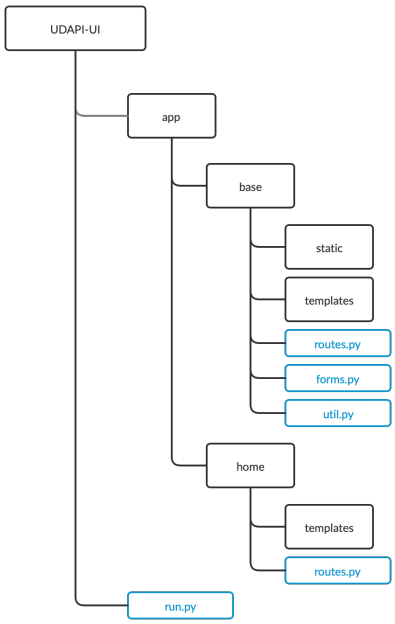

Figure 6.1: UDAPI UI Directory Structure

The front-end was built using HTML, CSS, JavaScript and Bootstrap 4. Bootstrap is a free and open-source CSS framework that is used for building responsive web applications. It provides numerous open-source templates to get started.

The front-end server was used to render the html pages, and act as the middleman between the user and UDAPI. There are two primary routes.

One for displaying a dynamic dashboard that contains a list of all the databases the user has created. Each of these databases can be deleted with the click of a button. The dashboard contains a list of all entitySets under each database with options to create new entity sets, update entity sets and delete them in the same page.

The second route is for handling entity sets and entities. CRUD operations can be performed for both entities and entity sets here. Given below we have the screenshots from the frontend implementation:

Unified Database API's login page, where the user can enter his/her username and password to gain access to the UDAPI.

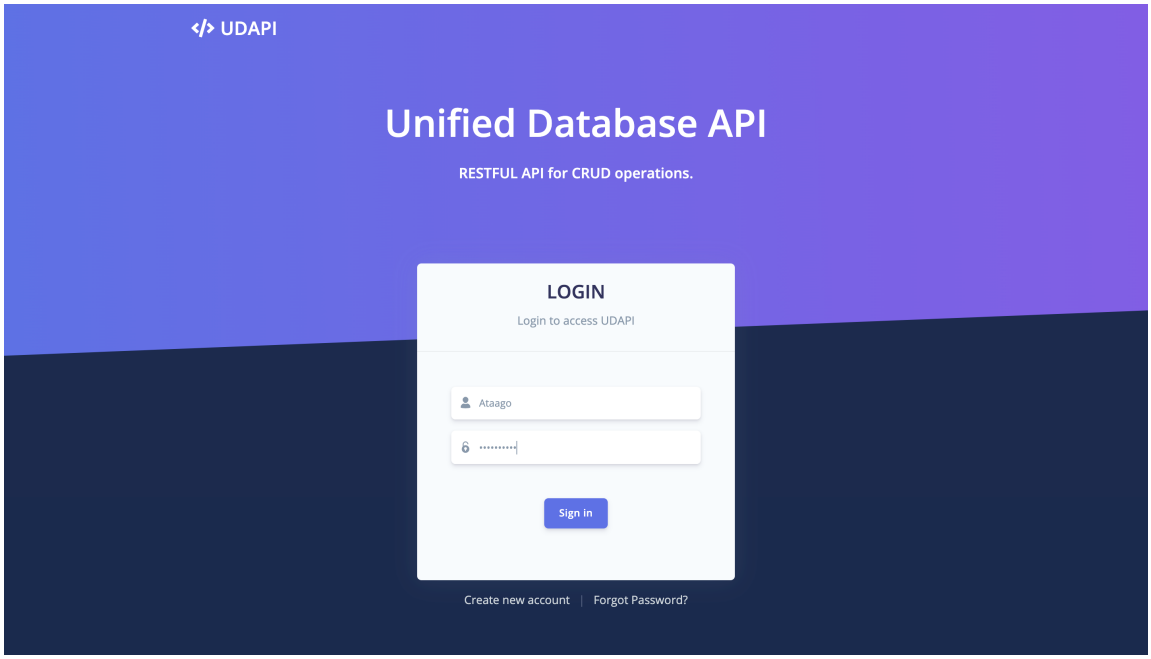

Figure 6.2: UDAPI Login Page

Unified Database API's login page, where the user can enter his/her username and password to gain access to the UDAPI.

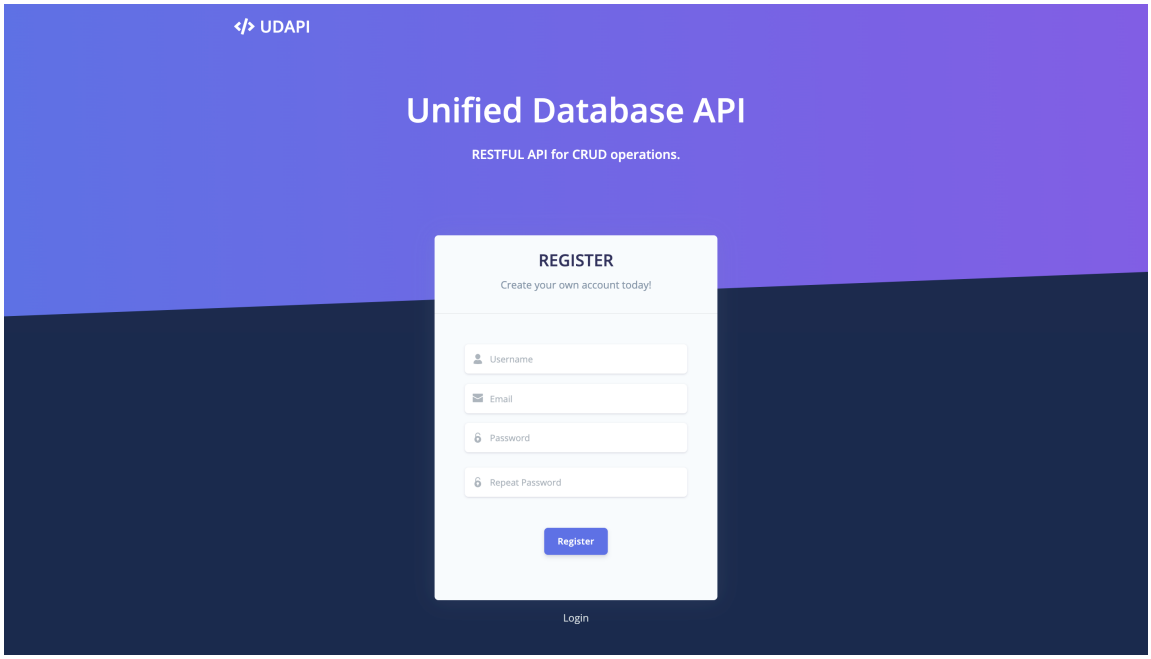

Figure 6.3: UDAPI Register Page

Unified Database API's Password Reset page, if the user forgets his/her password, they could reset it by using their email ID.

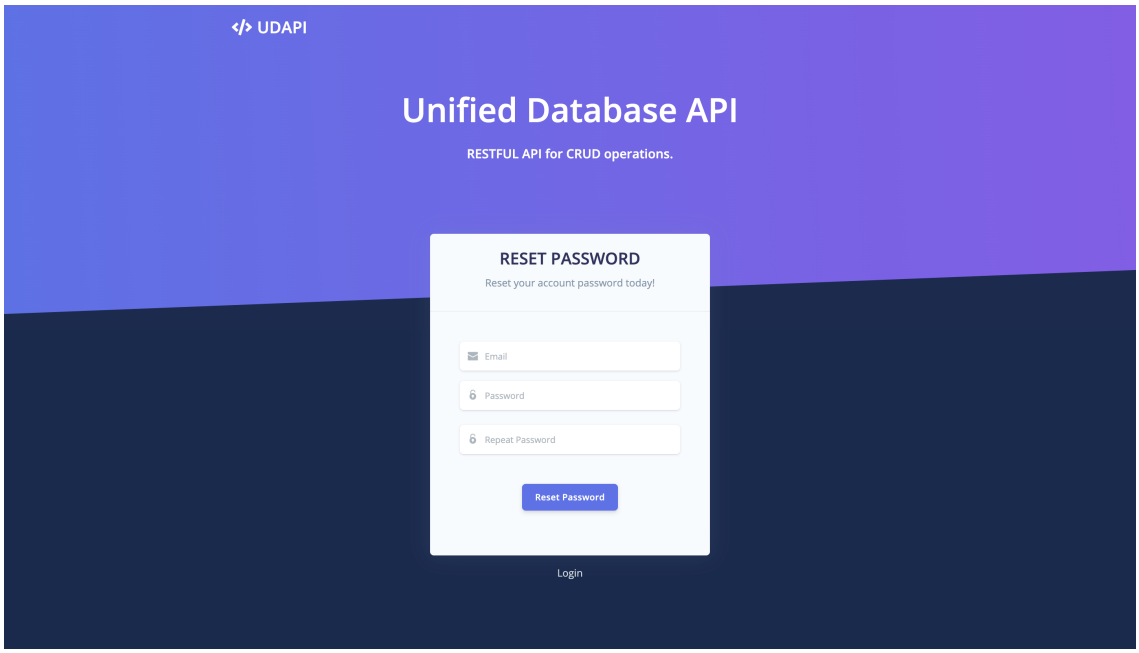

Figure 6.4: UDAPI Password reset Page

Unified Database API's Dashboard, All the databases of the user along with the users details could be seen and managed from this UDAPI home screen.

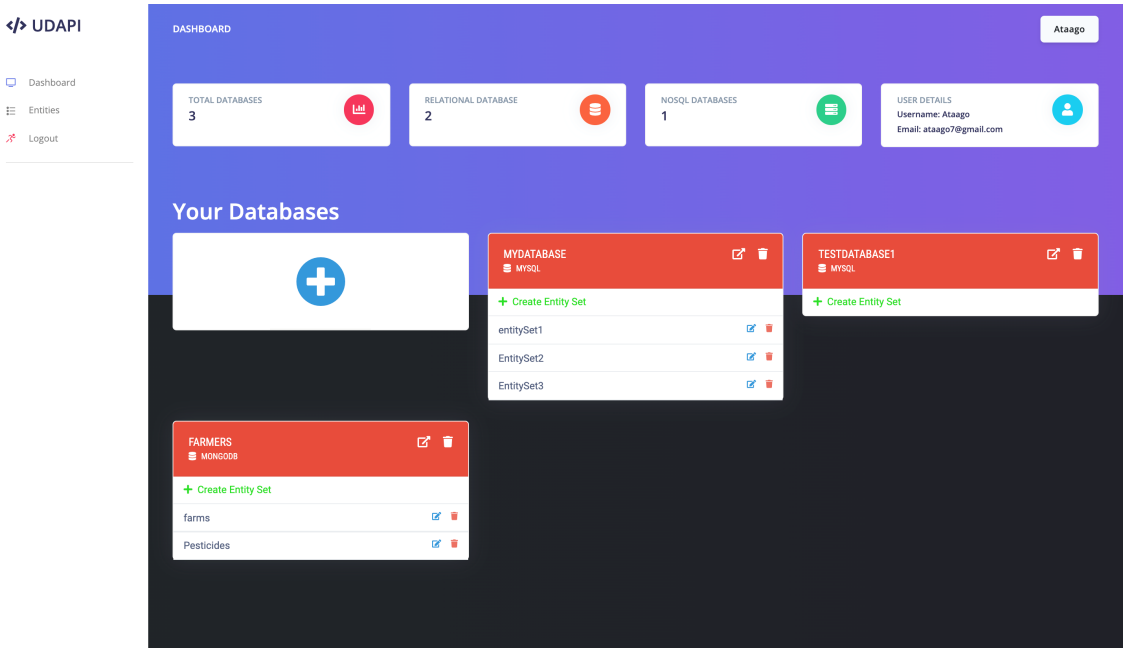

Figure 6.5: UDAPI Dashboard

Creating a new database is as simple as selecting the database type and giving it a name with a click of a button.

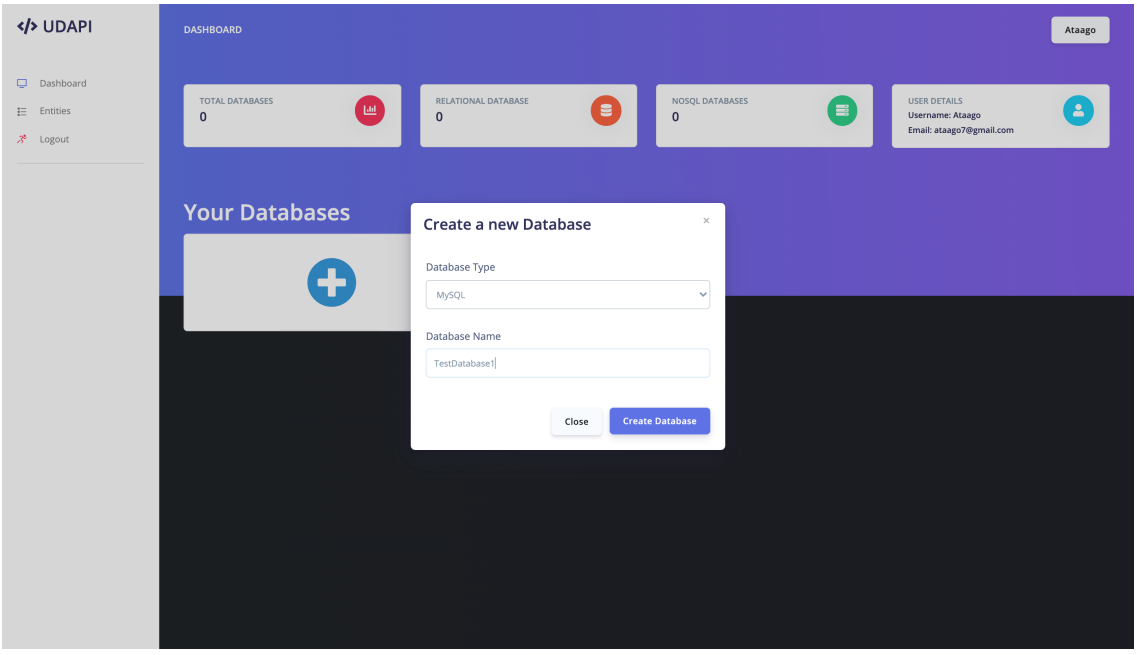

Figure 6.6: Creating a new Database

Deleting a Database is just a click of a button on the UI, but the user should be careful while doing this act.

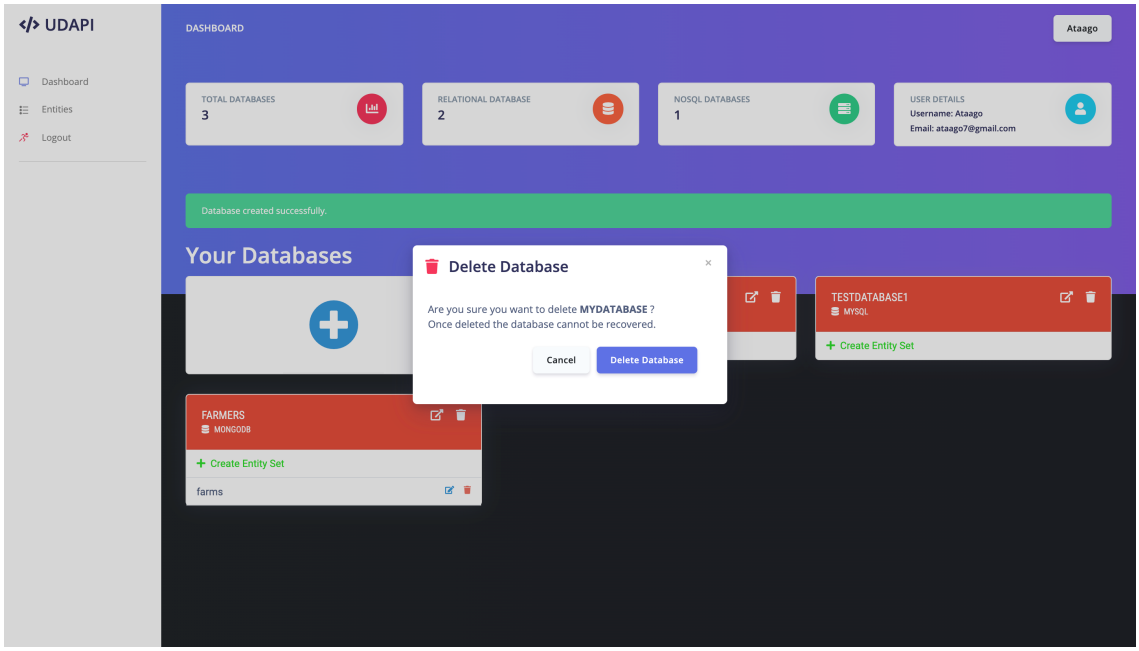

Figure 6.7: Deleting a new Database

Creating a new Entity set for a selected Database is shown in the picture, user can select if the attribute is a Primary key, Auto increment or Not null.

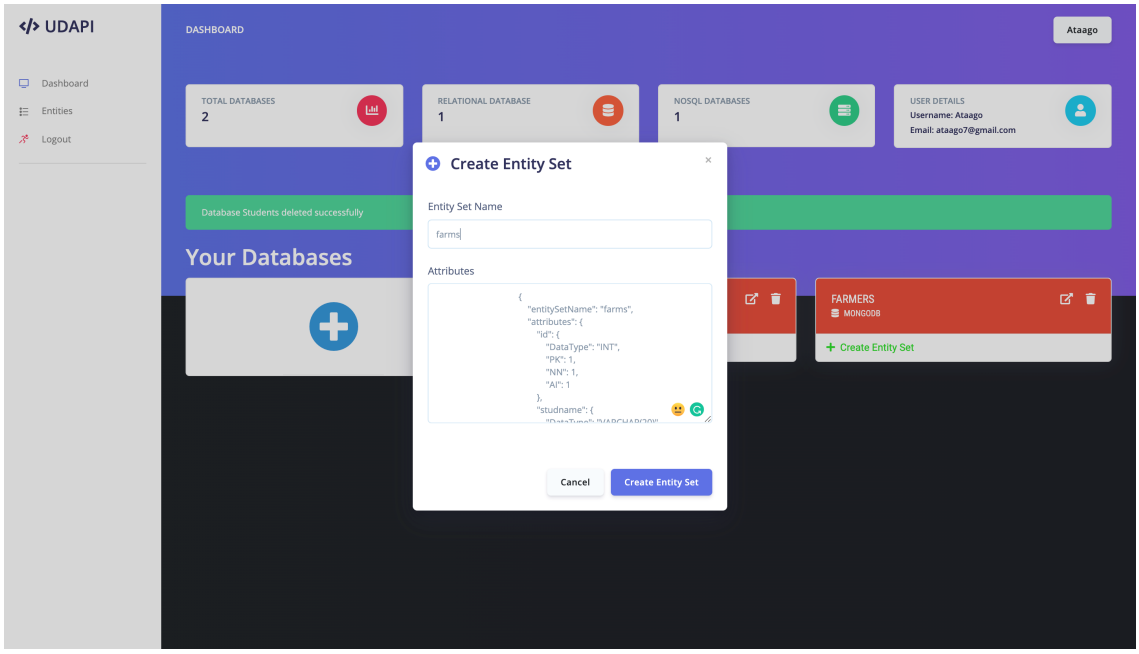

Figure 6.8: Creating an Entity set

Renaming an Entity set is as simple as clicking on rename option and typing in a new name for the Entity set.

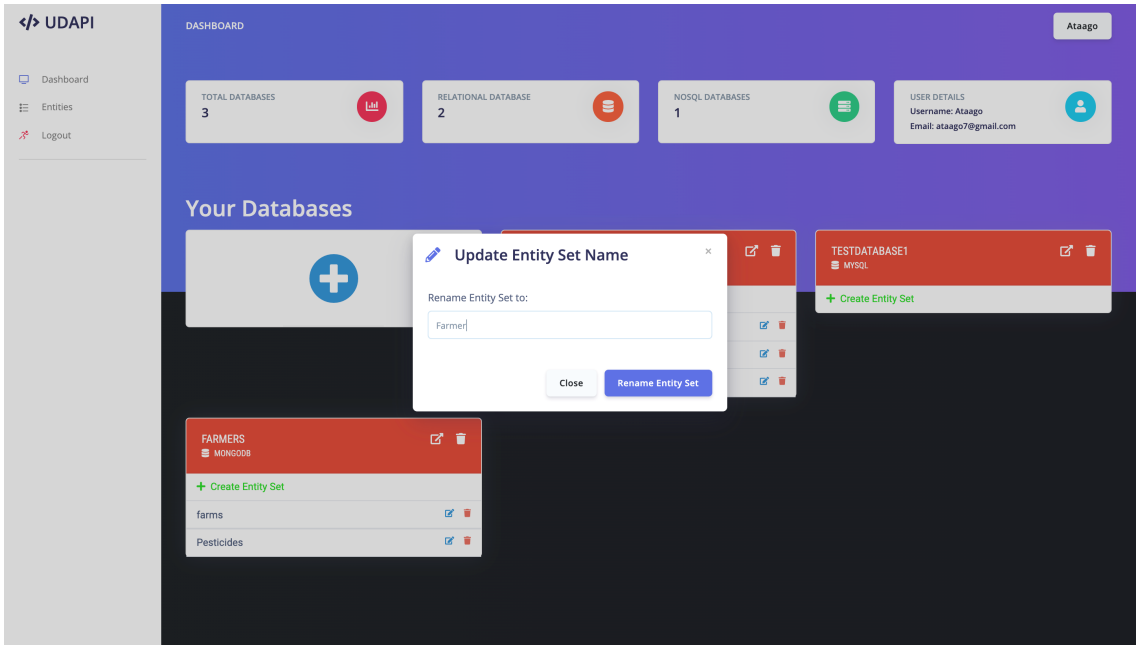

Figure 6.9: Renaming an Entity set

Deleting an Entity Set can be done with a simple click on the trash can beside the Entity set of a Database and confirming the same.

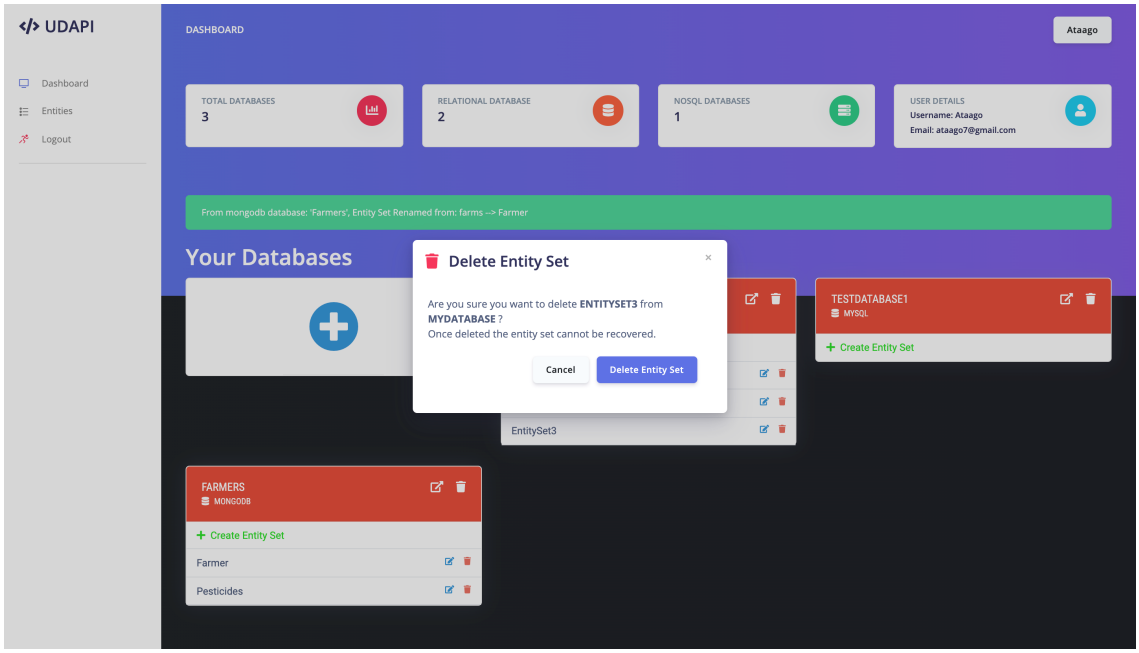

Figure 6.10: Deleting an Entity set

To view the details of the Database, user has to just click on the view button next to the database name in the Database Card on the Dashboard, which will take him/her to the details of the Database.

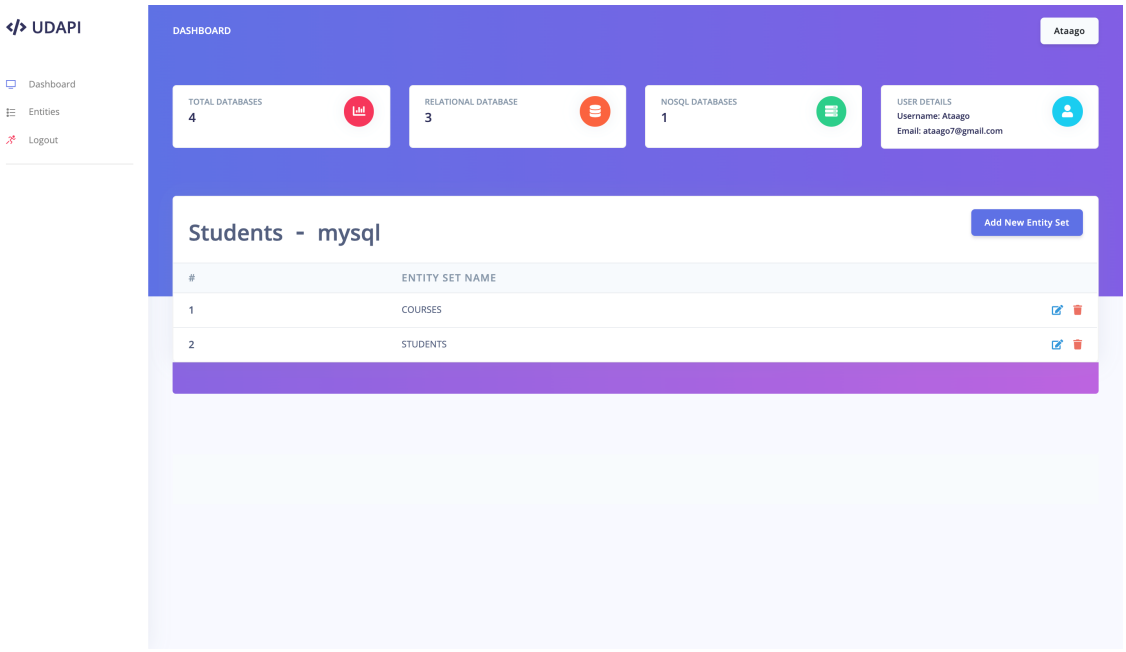

Figure 6.11: Details of Database

To view the contents of an entity set, the user has to click on the entity set name on the screen. This gives a table displaying the Entities of the Entity set selected.

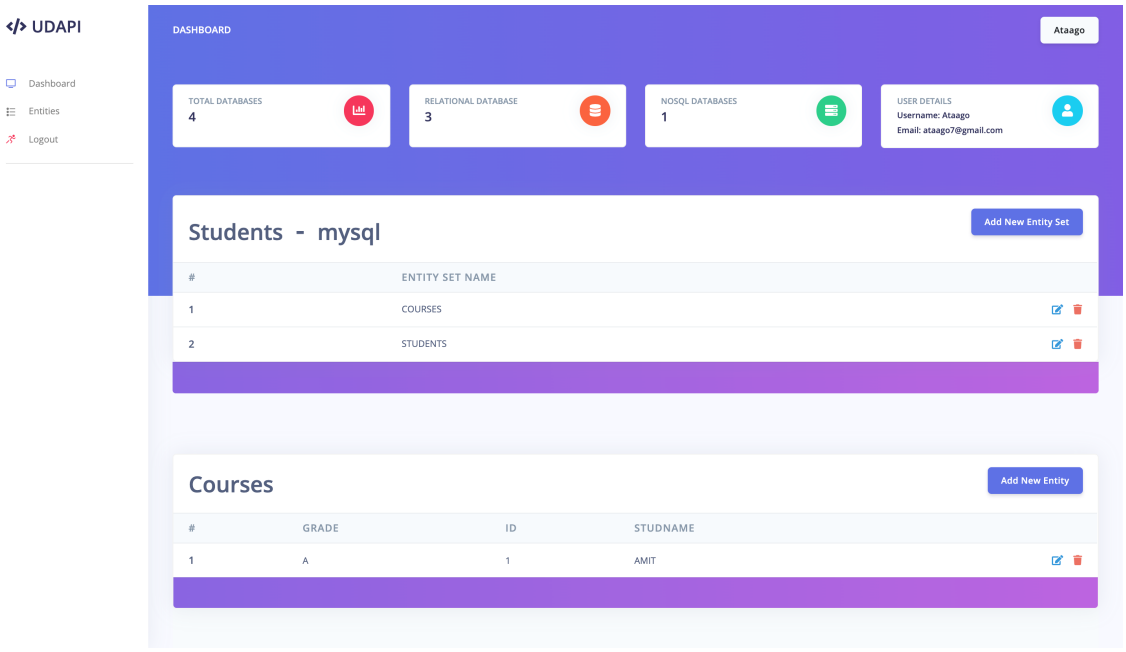

Figure 6.12: Viewing the Entities of an Entity set

To create or add a new Entity for an Entity Set, the user just has to click on the Add New Entity button beside the Entity Set name.

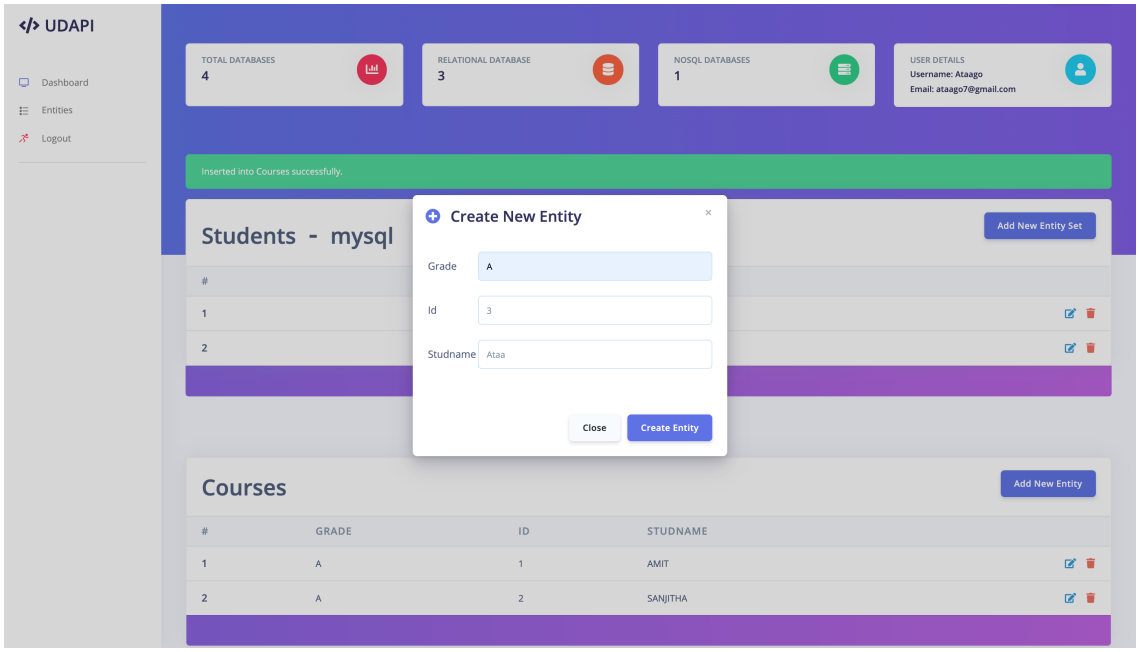

Figure 6.13: Adding a new Entity

Deleting an Entity can be done by just clicking on the trash button beside the Entity which is subjected to be deleted.

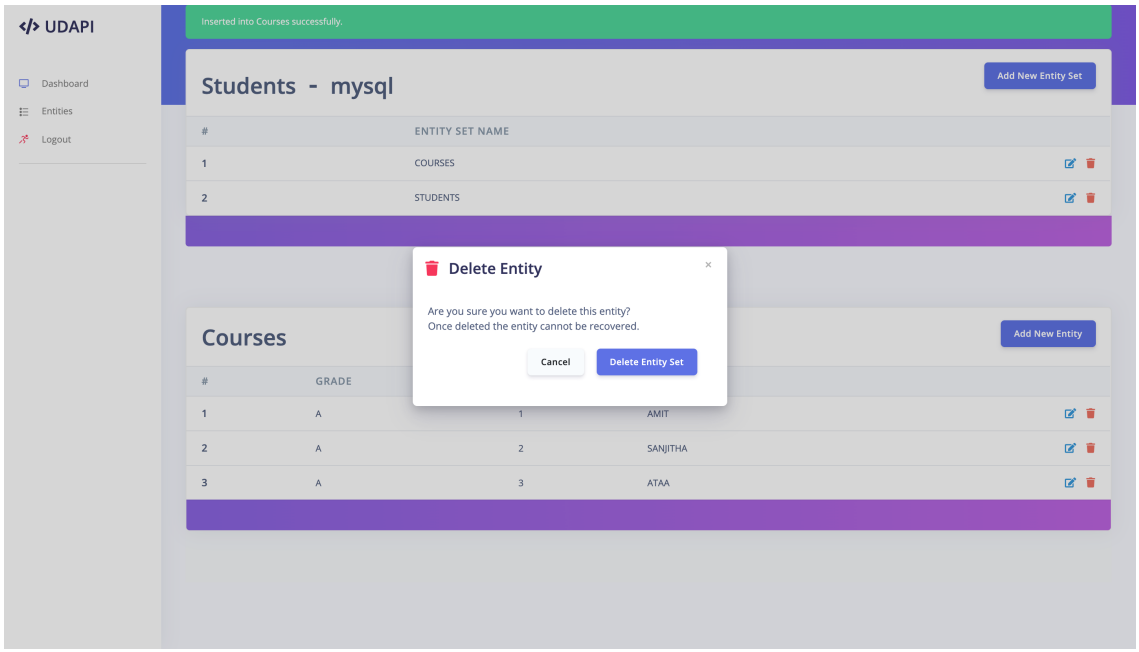

Figure 6.14: Deleting an Entity

Chapter 7

Conclusion

7.1 Future Work

At present, UDAPI only works for two databases : MySQL and MongoDB. Other databases can be supported in future iterations of the development process.

The process of creating entity sets takes JSON requests in the user interface. This could be modified into a much more intuitive form with dynamic input fields that can be added or deleted based on the number of attributes.

The Front-end can be further enhanced to include a POSTMAN like feature where the user could directly interact with UDAPI for quick requests and responses.

Tools can be provided for quick migration of datastore from one datastore to another with the press of a button. We hope to include all of these in the near future.

7.2 Conclusion

UDAPI, adapted from Sellami et. al's ODBAPI has implemented all the features the authors theorised about, and has also included its own enhancements. UDAPI is a streamlined and a unified REST API that allows the execution of CRUD operations on different NoSQL and relational databases. It decouples cloud applications from data stores, making the process of migration much easier. Furthermore, it relieves the developers from having to learn new APIs for different databases. UDAPI also has authentication and authorisation to ensure the security of the users' data and databases. There is still a long way to go but UDAPI is on its way to becoming a reliable one-stop-shop for all database needs.