# Duolingo DataBase Model

Marlon Yecid Riveros Guio 20231020208
Brayan Estiven Aguirre Aristizabal 20231020156

***Abstract -This paper presents a simplified approach to understanding the Entity-Relationship Model (ERM) underlying Duolingo's database system. By deconstructing key entities such as users, courses, lessons, achievements, and progress tracking. We provide a clear, intuitive ER diagram that captures the essence of the data structure. Alongside this, we introduce fundamental concepts of relational algebra, including essential operations like selection, projection, and joins, which are critical for effective database querying. Finally, we demonstrate a practical implementation of this database schema using Python, leveraging the Faker library to generate realistic test data and employing the Data Access Object (DAO) pattern to streamline data access logic.***

## I. Introduction

Language learning has been one of the most learned topics in the world for some time now. Acquiring languages makes it easier to travel around the world, acquire different opportunities, etc. which opens up countless opportunities. Nowadays, learning languages is not very difficult because with the development of technology and the Internet, learning a language is as easy as accessing your cell phone or computer and asking for a language course. Thanks to events like these, various companies have dedicated themselves to making contact with a second language more accessible, which is why various platforms have emerged that allow access to them. Currently, the platform that stands out the most in this area is duolingo, which has gained fame and has specialized in language teaching.

Modern applications demand robust database systems that can efficiently capture, store, and manipulate data to meet evolving user needs. In platforms like Duolingo, a clear understanding of the underlying data architecture is crucial not only for maintaining performance and scalability but also for enhancing user engagement. This paper aims to bridge the gap between theoretical database design and practical implementation by examining Duolingo's database structure through a simplified lens.

The first section of this paper focuses on the Entity-Relationship Model (ERM) by breaking down Duolingo's complex database into a simplified ER diagram by using reverse engineering. Here, we highlight the critical entities—users, courses, lessons, achievements, and progress tracking—and illustrate the relationships among them. This visualization helps demystify the data structure and provides a foundational understanding that is essential for both students and professionals in the field.

Building on this foundation, we introduce the core principles of relational algebra, a mathematical framework that underpins database operations. By exploring operations such as selection, projection, and joins, we demonstrate how these concepts are used to formulate queries that extract and manipulate data effectively. This theoretical background is integral to appreciating how modern databases function at a fundamental level.

The final section of the paper is dedicated to a hands-on implementation using Python. We utilize the Faker library to generate realistic test data, creating a simulated environment that mirrors the practical challenges faced when managing a live database. Additionally, we implement the Data Access Object (DAO) pattern to encapsulate the data access logic, promoting modularity and ease of maintenance in the codebase

Overall, this paper offers a comprehensive look at the interplay between database design and practical application. By simplifying complex concepts and demonstrating their real-world implementation, we aim to provide valuable insights that are applicable to Duolingo's system.

## II. Methodology

For the realization of this project, a methodology known as the Entity Relationship Model (ERM) was used, an efficient methodology that included an in-depth analysis of the application to recognize the main components that are necessary for the construction of a solid and scalable database. Then, the main entities and their attributes were defined, ensuring that each element necessary for the functionality of the application was represented. After this, the relationships between the entities were verified to describe the interactions within the application, defining which ones were related to each other and defining the types of relationships that each of these presented. Next, a first sketch of what would be the first entity relationship diagram was made.

Once the first outline was finished, a review of all the analysis carried out up to that point was carried out, in which small errors were corrected and the new entities were defined, in charge of breaking the "many to many" relationships that were presented in the first entity relationship diagram. The entity-relationship diagram was then validated by reviewing each component and testing the consistency of the relationships between entities, ensuring that all system interactions were correctly represented. The design was then refined by implementing each entity's data structure and properties, so that the model could be even more robust. The final result was a solid, efficient and adaptable model, ready to be implemented as a database for optimal management of users, courses, lessons and other components of the Duolingo system.

After finalizing the entity-relationship model, the next step was to set up a structured environment to deploy the database using Docker. Docker containers were used to ensure a consistent and reproducible environment, preventing issues related to differences in configurations across different systems. A Docker Compose file was created to define the database service, specifying the image, ports, volumes, and environment variables necessary for the database to function properly.

With the containerized database environment in place, an SQL script was developed to create the required tables. This script included the creation of tables according to the previously defined entity-relationship model, ensuring the integrity of the data through appropriate constraints such as primary keys, foreign keys, unique constraints, and indexes.

Finally, a Python application was developed to interact with the database and perform all the necessary CRUD (Create, Read, Update, Delete) operations for each table.

## III. METHOD AND MATERIALS

Firstly, a technical discussion was held to reach an agreement on what the main components of the application would be. The process began with a deep analysis of Duolingo's core functionality to identify the components needed for a robust and scalable database. Key features analyzed included:

1. User registration and authentication.
2. Storage of user information
3. Catalog of available languages to learn.
4. Structuring lessons by language.
5. Classification of lessons by difficulty levels.
6. Storage of lesson content.
7. Show Conversations and expressions
8. Interactive Exercises
9. Achievement and Reward System
10. Show the Progress

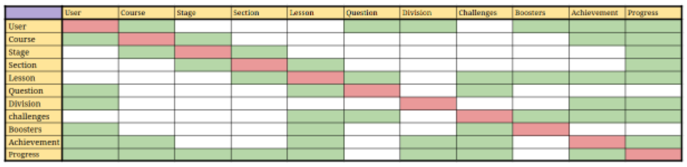
Once the essential components were identified, the main entities or tables that will form part of the database structure were defined. These entities include:

1. User
2. Course
3. Stage
4. Section
5. Lesson
6. Question
7. Division
8. Challenges
9. Boosters
10. Achievement
11. progress

In the next step, the attributes or characteristics of each of the identified entities were detailed. For example, the attributes for the User Table were:

- User: id_PK, name, nickname, joinDate, division_FK, timesInTop, email, password, courses_FK, achievements_FK, followed, followers, gemsNumber, livesNumber, boostersNumber, isPremium

Detailed and exhaustive definitions were carried out on the interactions and connections that exist between the various entities involved:



| | User | Course | Stage | Section | Lesson | Question | Division | Challenges | Boosters | Achievement | Progress |
|---|---|---|---|---|---|---|---|---|---|---|---|
| User | 🟥 | 🟩 | | | | 🟩 | 🟩 | | 🟩 | 🟩 | 🟩 |
| Course | 🟩 | 🟥 | 🟩 | | | | | | | 🟩 | 🟩 |
| Stage | | 🟩 | 🟥 | 🟩 | | | | | | | 🟩 |
| Section | | | 🟩 | 🟥 | 🟩 | | | | | | 🟩 |
| Lesson | | | | 🟩 | 🟥 | 🟩 | | 🟩 | 🟩 | 🟩 | 🟩 |
| Question | 🟩 | | | | 🟩 | 🟥 | | 🟩 | | | |
| Division | 🟩 | | | | | | 🟥 | | | 🟩 | 🟩 |
| challenges | | | | | 🟩 | 🟩 | | 🟥 | 🟩 | 🟩 | 🟩 |
| Boosters | 🟩 | | | | 🟩 | | | 🟩 | 🟥 | | |
| Achievement | 🟩 | 🟩 | | | 🟩 | | 🟩 | 🟩 | | 🟥 | 🟩 |
| Progress | 🟩 | 🟩 | 🟩 | 🟩 | 🟩 | | 🟩 | 🟩 | | 🟩 | 🟥 |

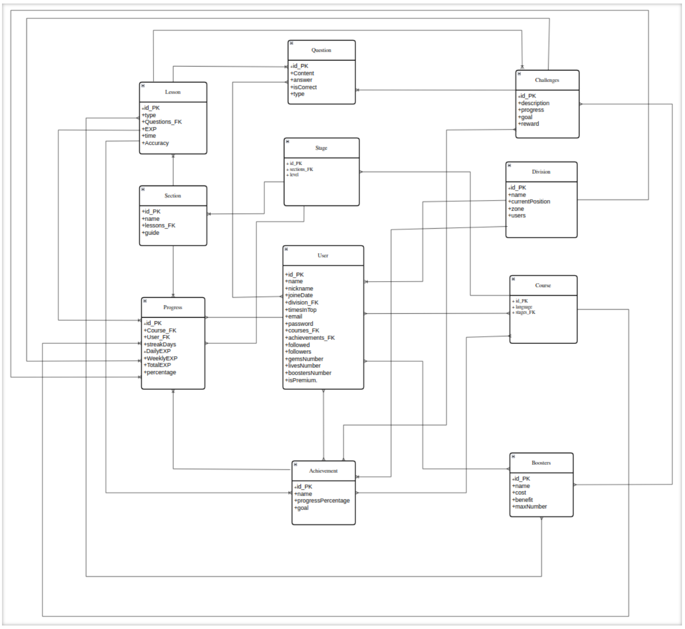
A example of sentence that helped us find existing relationships between the User and Course Tables were:

- User-Course: One user can take many courses and a course can be completed by many users.

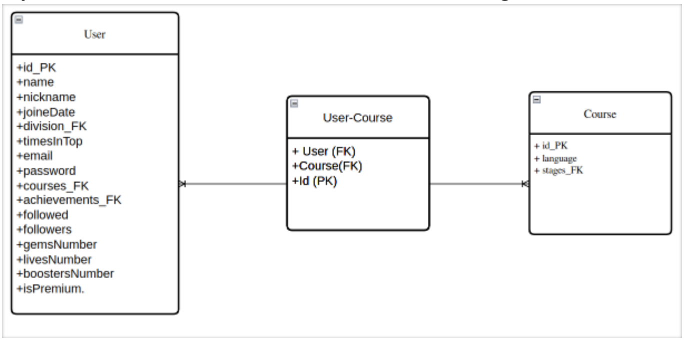
We continue with the classification of the relationships for the 25 relationships that were found, in the case of the User Table and the Course table:

- User-Course: Many to many (* → *)

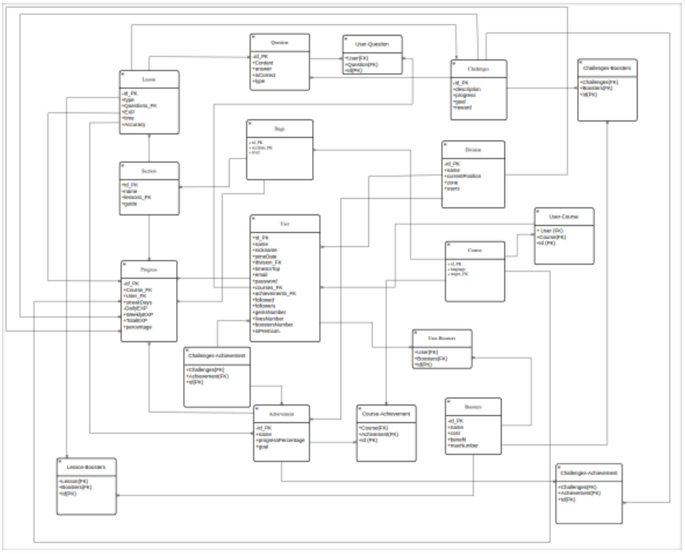
It should be noted that this process was done in the same way for the other tables.
Then the first entity relationship diagram was designed, the result is shown in the image:



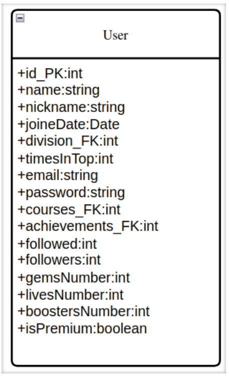
Once this was finished, the existing "many to many" entities were broken up. To do this, an additional entity was created, and the two original entities were connected to it with a "one to many" relationship, and taking as attributes in the new entity two foreign keys that would be our original entities.



Then we make a second diagram with the new entities, like this:



Finally, the data structure was categorized and the properties were added to each attribute of our entities.



To define the data structure we look at what function this attribute is fulfilling in the system, so we categorize it by "int, String, Date, etc."

And the restrictions and properties were provided to these attributes according to the functionalities analyzed above.

**User**

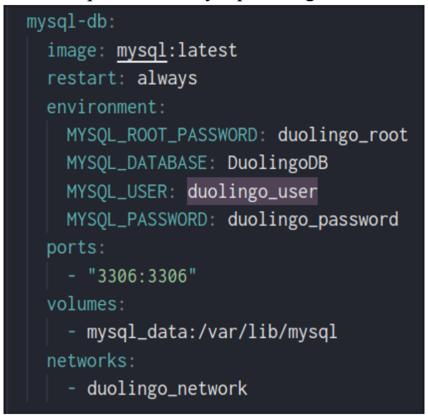
- id_PK: INT (Primary Key, Auto Increment, Not Null)
- name: VARCHAR(50) (Not Null)
- nickname: VARCHAR(50) (Not Null, Unique)
- joinDate: DATE (Default CURRENT_DATE())
- division_FK: INT
- timesInTop: INT (Default 0)
- email: VARCHAR(75) (Not Null, Unique)
- password: VARCHAR(32) (Not Null)
- courses_FK: INT (Not Null)
- achievements_FK: INT
- followed: INT (Default 0)
- followers: INT (Default 0)
- gemsNumber: INT (Default 0)
- livesNumber: INT (Default 5)
- boostersNumber: INT (Default 0)
- isPremium: BOOLEAN (Default False)

Once the MER was finished, A docker-compose.yml file was created to define the required services. The main service in this setup is the database, which runs inside a container and is configured with the necessary environment variables for proper operation. The official PostgreSQL image and MySql image were used to ensure stability and compatibility.

In the configuration, environment variables were set to define the user, password, and database name. A volume was also configured to store data persistently for each service, avoiding information loss if the container is stopped or restarted. In addition, a specific port was assigned to allow external connection from the host system, in this case we used the port 3306 to MySql and the port 5432 to PostGresSQL

In summary, the use of Docker Compose simplifies the deployment and management of the database services, providing a consistent and isolated environment for development and testing. By leveraging official images, environment variables, persistent volumes, and port mappings

An example of the MySql configuration is:



```
mysql-db:
  image: mysql:latest
  restart: always
  environment:
    MYSQL_ROOT_PASSWORD: duolingo_root
    MYSQL_DATABASE: DuolingoDB
    MYSQL_USER: duolingo_user
    MYSQL_PASSWORD: duolingo_password
  ports:
    - "3306:3306"
  volumes:
    - mysql_data:/var/lib/mysql
  networks:
    - duolingo_network
```

Before implementing the Duolingo database in SQL, a theoretical analysis was performed using relational algebra operations. This methodology allowed us to verify that the MER correctly captured the functional requirements of the application, ensuring that future queries and operations on the data would be consistent and efficient.

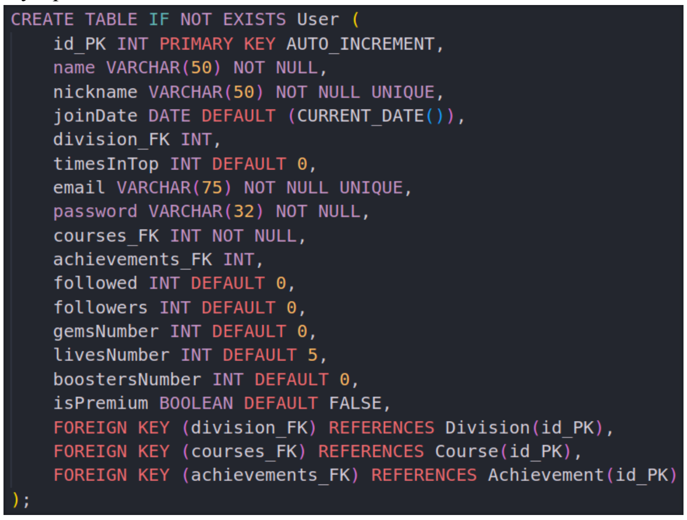
Relational algebra provides a formal framework for defining operations on relationships in an abstract way. In this way, queries and data manipulations that a user might request are simulated. Two simple examples adapted to the Duolingo context are shown below.

- $\pi_{\text{nickname, email}}\left(\sigma_{\text{isPremium}=\text{True}}(\text{User})\right)$
- $\pi_{id\_PK,\, name}\left(\sigma_{timesInTop\geq 3}(\text{User})\right)$

These examples illustrate some of the requests that a user can perform on the database. For instance, one query retrieves premium users by selecting their nickname and email when the attribute isPremium is True, while another query identifies standout users by filtering those whose timesInTop value is greater than or equal to 3 and then extracting their id_PK and name.

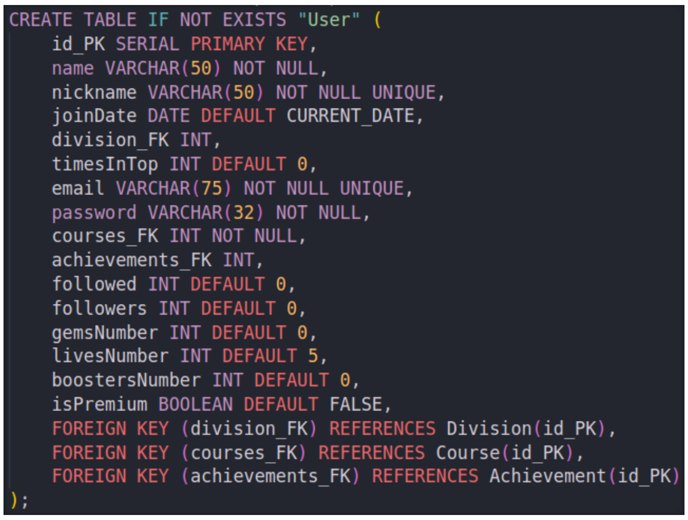
Once we have checked the MER structure with relational algebra, we write a SQL script where we create all our tables, inserting their attributes, with their respective data types and restrictions.

The first image below is showing the creation of the User Table for MySql



```
CREATE TABLE IF NOT EXISTS User (
    id_PK INT PRIMARY KEY AUTO_INCREMENT,
    name VARCHAR(50) NOT NULL,
    nickname VARCHAR(50) NOT NULL UNIQUE,
    joinDate DATE DEFAULT (CURRENT_DATE()),
    division_FK INT,
    timesInTop INT DEFAULT 0,
    email VARCHAR(75) NOT NULL UNIQUE,
    password VARCHAR(32) NOT NULL,
    courses_FK INT NOT NULL,
    achievements_FK INT,
    followed INT DEFAULT 0,
    followers INT DEFAULT 0,
    gemsNumber INT DEFAULT 0,
    livesNumber INT DEFAULT 5,
    boostersNumber INT DEFAULT 0,
    isPremium BOOLEAN DEFAULT FALSE,
    FOREIGN KEY (division_FK) REFERENCES Division(id_PK),
    FOREIGN KEY (courses_FK) REFERENCES Course(id_PK),
    FOREIGN KEY (achievements_FK) REFERENCES Achievement(id_PK)
);
```

The second image below is showing the creation of the User Table for PostgreSQL

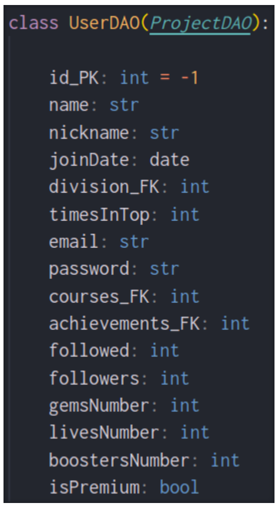


```
CREATE TABLE IF NOT EXISTS "User" (
    id_PK SERIAL PRIMARY KEY,
    name VARCHAR(50) NOT NULL,
    nickname VARCHAR(50) NOT NULL UNIQUE,
    joinDate DATE DEFAULT CURRENT_DATE,
    division_FK INT,
    timesInTop INT DEFAULT 0,
    email VARCHAR(75) NOT NULL UNIQUE,
    password VARCHAR(32) NOT NULL,
    courses_FK INT NOT NULL,
    achievements_FK INT,
    followed INT DEFAULT 0,
    followers INT DEFAULT 0,
    gemsNumber INT DEFAULT 0,
    livesNumber INT DEFAULT 5,
    boostersNumber INT DEFAULT 0,
    isPremium BOOLEAN DEFAULT FALSE,
    FOREIGN KEY (division_FK) REFERENCES Division(id_PK),
    FOREIGN KEY (courses_FK) REFERENCES Course(id_PK),
    FOREIGN KEY (achievements_FK) REFERENCES Achievement(id_PK)
);
```

After setting up the database environment, we proceeded with the development of the Python application, which was responsible for handling the backend operations. The development process followed a structured approach to ensure modularity, maintainability, and efficiency.

**Implementing the DAO Pattern**

The first step in building the application was defining the core classes using the Data Access Object (DAO) pattern. This pattern was chosen to separate the database logic from the application logic, improving code reusability and making the system easier to maintain. Each entity in the database was represented as a class, encapsulating the attributes to interact with the database in an organized manner. These classes defined the structure of the objects used in the application and served as an intermediary between the database and the rest of the system. In this case we'll show you the code for User:

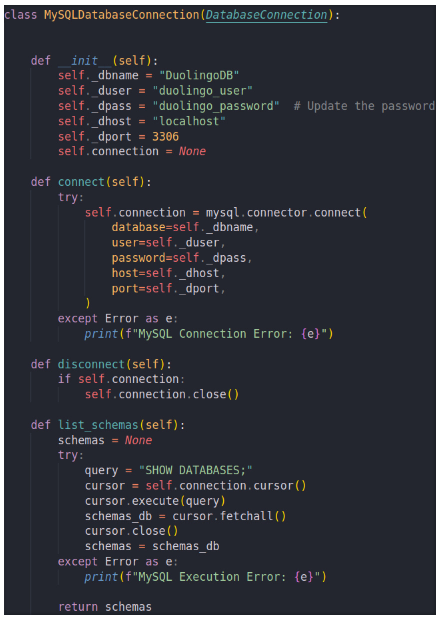


```
class UserDAO(ProjectDAO):

    id_PK: int = -1
    name: str
    nickname: str
    joinDate: date
    division_FK: int
    timesInTop: int
    email: str
    password: str
    courses_FK: int
    achievements_FK: int
    followed: int
    followers: int
    gemsNumber: int
    livesNumber: int
    boostersNumber: int
    isPremium: bool
```

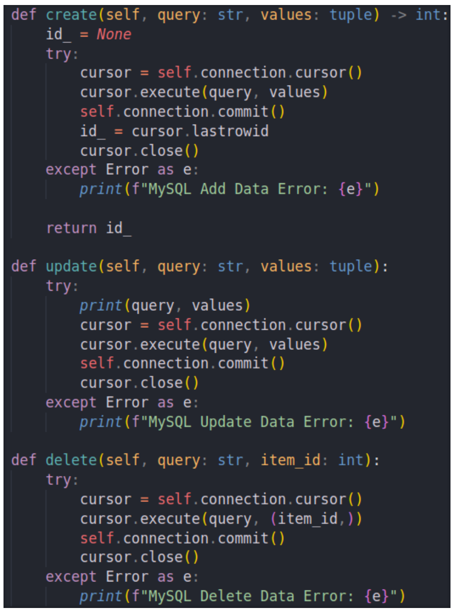
**Creating the 'Connections' Package**

Once the entity classes were in place, the next step was to implement a package called 'connections'. This package was responsible for managing database connections efficiently. It included a module that established and maintained a persistent connection to the PostgreSQL database using SQLAlchemy. By centralizing the connection logic, we ensured that the database could be accessed consistently across different parts of the application, reducing redundancy and potential errors related to database handling.

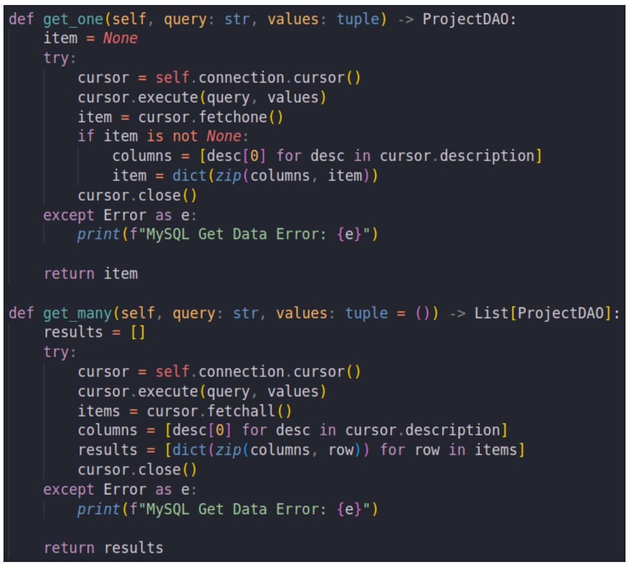
The next 3 images show the code for the MySql Connection, using the library mysql.connector



```
class MySQLDatabaseConnection(DatabaseConnection):


    def __init__(self):
        self._dbname = "DuolingoDB"
        self._duser = "duolingo_user"
        self._dpass = "duolingo_password"  # Update the password
        self._dhost = "localhost"
        self._dport = 3306
        self.connection = None

    def connect(self):
        try:
            self.connection = mysql.connector.connect(
                database=self._dbname,
                user=self._duser,
                password=self._dpass,
                host=self._dhost,
                port=self._dport,
            )
        except Error as e:
            print(f"MySQL Connection Error: {e}")

    def disconnect(self):
        if self.connection:
            self.connection.close()

    def list_schemas(self):
        schemas = None
        try:
            query = "SHOW DATABASES;"
            cursor = self.connection.cursor()
            cursor.execute(query)
            schemas_db = cursor.fetchall()
            cursor.close()
            schemas = schemas_db
        except Error as e:
            print(f"MySQL Execution Error: {e}")

        return schemas
```

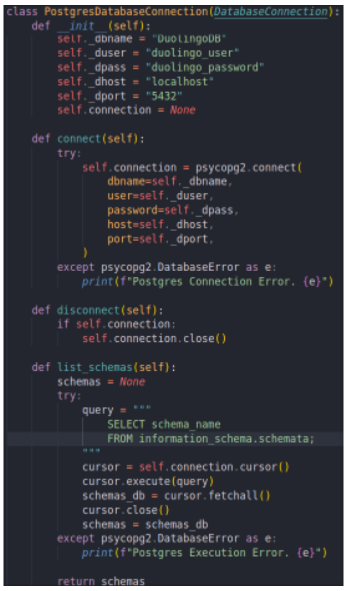


```
def create(self, query: str, values: tuple) -> int:
    id_ = None
    try:
        cursor = self.connection.cursor()
        cursor.execute(query, values)
        self.connection.commit()
        id_ = cursor.lastrowid
        cursor.close()
    except Error as e:
        print(f"MySQL Add Data Error: {e}")

    return id_

def update(self, query: str, values: tuple):
    try:
        print(query, values)
        cursor = self.connection.cursor()
        cursor.execute(query, values)
        self.connection.commit()
        cursor.close()
    except Error as e:
        print(f"MySQL Update Data Error: {e}")

def delete(self, query: str, item_id: int):
    try:
        cursor = self.connection.cursor()
        cursor.execute(query, (item_id,))
        self.connection.commit()
        cursor.close()
    except Error as e:
        print(f"MySQL Delete Data Error: {e}")
```

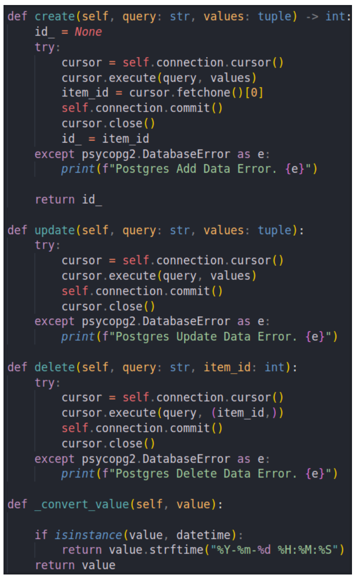


```python
def get_one(self, query: str, values: tuple) -> ProjectDAO:
    item = None
    try:
        cursor = self.connection.cursor()
        cursor.execute(query, values)
        item = cursor.fetchone()
        if item is not None:
            columns = [desc[0] for desc in cursor.description]
            item = dict(zip(columns, item))
        cursor.close()
    except Error as e:
        print(f"MySQL Get Data Error: {e}")

    return item

def get_many(self, query: str, values: tuple = ()) -> List[ProjectDAO]:
    results = []
    try:
        cursor = self.connection.cursor()
        cursor.execute(query, values)
        items = cursor.fetchall()
        columns = [desc[0] for desc in cursor.description]
        results = [dict(zip(columns, row)) for row in items]
        cursor.close()
    except Error as e:
        print(f"MySQL Get Data Error: {e}")

    return results
```

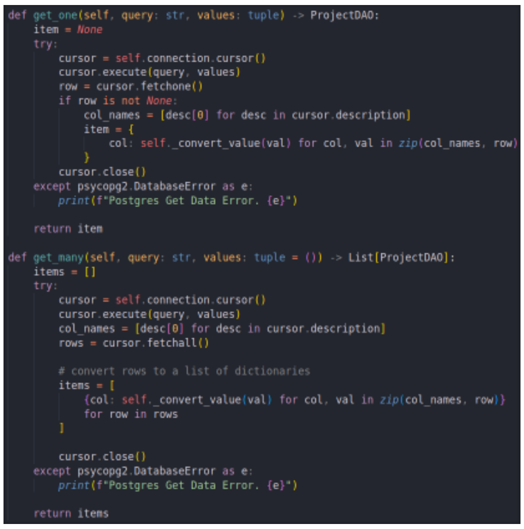
And the next images are showing the same connection but using PostgreSQL:



```python
class PostgresDatabaseConnection(DatabaseConnection):
    def __init__(self):
        self._dbname = "DuolingoDB"
        self._duser = "duolingo_user"
        self._dpass = "duolingo_password"
        self._dhost = "localhost"
        self._dport = "5432"
        self.connection = None

    def connect(self):
        try:
            self.connection = psycopg2.connect(
                dbname=self._dbname,
                user=self._duser,
                password=self._dpass,
                host=self._dhost,
                port=self._dport,
            )
        except psycopg2.DatabaseError as e:
            print(f"Postgres Connection Error. {e}")

    def disconnect(self):
        if self.connection:
            self.connection.close()

    def list_schemas(self):
        schemas = None
        try:
            query = """
                SELECT schema_name
                FROM information_schema.schemata;
            """
            cursor = self.connection.cursor()
            cursor.execute(query)
            schemas_db = cursor.fetchall()
            cursor.close()
            schemas = schemas_db
        except psycopg2.DatabaseError as e:
            print(f"Postgres Execution Error. {e}")

        return schemas
```



```python
def create(self, query: str, values: tuple) -> int:
    id_ = None
    try:
        cursor = self.connection.cursor()
        cursor.execute(query, values)
        item_id = cursor.fetchone()[0]
        self.connection.commit()
        cursor.close()
        id_ = item_id
    except psycopg2.DatabaseError as e:
        print(f"Postgres Add Data Error. {e}")

    return id_

def update(self, query: str, values: tuple):
    try:
        cursor = self.connection.cursor()
        cursor.execute(query, values)
        self.connection.commit()
        cursor.close()
    except psycopg2.DatabaseError as e:
        print(f"Postgres Update Data Error. {e}")

def delete(self, query: str, item_id: int):
    try:
        cursor = self.connection.cursor()
        cursor.execute(query, (item_id,))
        self.connection.commit()
        cursor.close()
    except psycopg2.DatabaseError as e:
        print(f"Postgres Delete Data Error. {e}")

def _convert_value(self, value):

    if isinstance(value, datetime):
        return value.strftime("%Y-%m-%d %H:%M:%S")
    return value
```

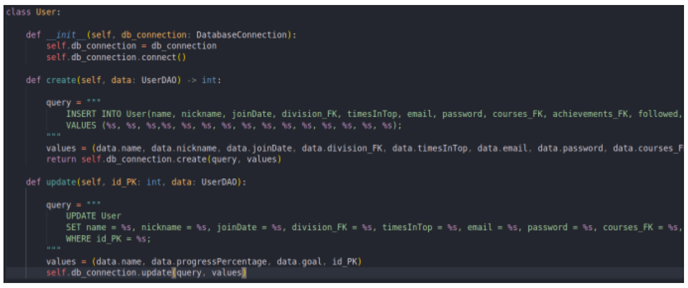


```python
def get_one(self, query: str, values: tuple) -> ProjectDAO:
    item = None
    try:
        cursor = self.connection.cursor()
        cursor.execute(query, values)
        row = cursor.fetchone()
        if row is not None:
            col_names = [desc[0] for desc in cursor.description]
            item = {
                col: self._convert_value(val) for col, val in zip(col_names, row)
            }
        cursor.close()
    except psycopg2.DatabaseError as e:
        print(f"Postgres Get Data Error. {e}")

    return item

def get_many(self, query: str, values: tuple = ()) -> List[ProjectDAO]:
    items = []
    try:
        cursor = self.connection.cursor()
        cursor.execute(query, values)
        col_names = [desc[0] for desc in cursor.description]
        rows = cursor.fetchall()

        # convert rows to a list of dictionaries
        items = [
            {col: self._convert_value(val) for col, val in zip(col_names, row)}
            for row in rows
        ]

        cursor.close()
    except psycopg2.DatabaseError as e:
        print(f"Postgres Get Data Error. {e}")

    return items
```

**Developing the 'CRUD' Package**

With the database connection in place, we created a 'CRUD' package, which contained the implementation of all Create, Read, Update, and Delete (CRUD) operations for each entity. This package ensured that each entity in the system had dedicated functions for inserting new records, retrieving existing data, modifying entries, and deleting unnecessary records. By structuring the CRUD operations separately, we achieved better organization and scalability, allowing for future modifications without disrupting the entire system.

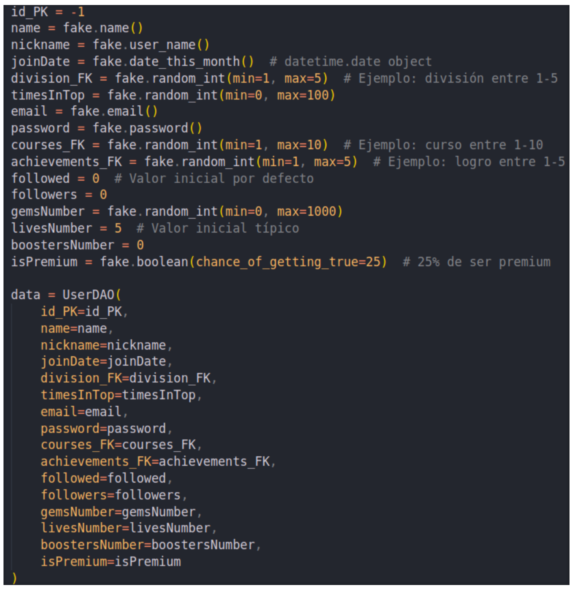
In the next image we are going to show you a part of the code with some operation of the CRUD, the idea is the same for all the operations, only changes the query structure



```python
class User:

    def __init__(self, db_connection: DatabaseConnection):
        self.db_connection = db_connection
        self.db_connection.connect()

    def create(self, data: UserDAO) -> int:

        query = """
            INSERT INTO User(name, nickname, joinDate, division_FK, timesInTop, email, password, courses_FK, achievements_FK, followed,
            VALUES (%s, %s, %s,%s, %s, %s, %s, %s, %s, %s, %s, %s, %s, %s, %s);
        """
        values = (data.name, data.nickname, data.joinDate, data.division_FK, data.timesInTop, data.email, data.password, data.courses_F
        return self.db_connection.create(query, values)

    def update(self, id_PK: int, data: UserDAO):

        query = """
            UPDATE User
            SET name = %s, nickname = %s, joinDate = %s, division_FK = %s, timesInTop = %s, email = %s, password = %s, courses_FK = %s,
            WHERE id_PK = %s;
        """
        values = (data.name, data.progressPercentage, data.goal, id_PK)
        self.db_connection.update(query, values)
```

**Building the 'initialization' Package**

To facilitate testing and ensure that the system had sample data for validation, we developed an 'initialization' package. This package used the Faker library to generate realistic test data for all entities. It included scripts to populate the database with mock users, courses, lessons, and other relevant data, enabling developers to test the functionality of the backend without requiring manual data entry. This approach not only streamlined the testing process but also provided a way to simulate real-world scenarios effectively. By example, in the module Init_Users.py we set the attributes for the table with faker this way:



```python
id_PK = -1
name = fake.name()
nickname = fake.user_name()
joinDate = fake.date_this_month()  # datetime.date object
division_FK = fake.random_int(min=1, max=5)  # Ejemplo: división entre 1-5
timesInTop = fake.random_int(min=0, max=100)
email = fake.email()
password = fake.password()
courses_FK = fake.random_int(min=1, max=10)  # Ejemplo: curso entre 1-10
achievements_FK = fake.random_int(min=1, max=5)  # Ejemplo: logro entre 1-5
followed = 0  # Valor inicial por defecto
followers = 0
gemsNumber = fake.random_int(min=0, max=1000)
livesNumber = 5  # Valor inicial típico
boostersNumber = 0
isPremium = fake.boolean(chance_of_getting_true=25)  # 25% de ser premium

data = UserDAO(
    id_PK=id_PK,
    name=name,
    nickname=nickname,
    joinDate=joinDate,
    division_FK=division_FK,
    timesInTop=timesInTop,
    email=email,
    password=password,
    courses_FK=courses_FK,
    achievements_FK=achievements_FK,
    followed=followed,
    followers=followers,
    gemsNumber=gemsNumber,
    livesNumber=livesNumber,
    boostersNumber=boostersNumber,
    isPremium=isPremium
)
```

Through this structured methodology, the Python backend was designed to be modular, scalable, and efficient, ensuring smooth interaction with the database while maintaining clean and maintainable code.

## IV. Conclusions

The development of this project not only allowed for the establishment of a solid and efficient database model for a language learning platform, but also highlighted the importance of integrating theoretical foundations with modern tools to achieve robust and scalable solutions. First, the design of the Entity-Relationship (ER) Model was carried out through a thorough analysis of the system requirements, which enabled the identification and precise definition of key entities—such as User, Course, Lesson, Achievements, and Progress—and their relationships. The transformation of many-to-many relationships into one-to-many associations, through the creation of intermediate entities, not only guaranteed referential integrity, but also facilitated the scalability and maintenance of the model in the long term.

The technical implementation highlighted the use of Docker and Docker Compose, tools that allowed for the creation of a reproducible and consistent deployment environment for the MySQL and PostgreSQL databases. This strategy eliminated configuration discrepancies between development and production environments, ensuring stable and predictable data management. Likewise, the generation of SQL scripts from the ER model included the definition of primary and foreign keys, uniqueness restrictions and indexes, which reinforces the consistency and integrity of the stored information.

In terms of integration with Python, the adoption of the Data Access Object (DAO) pattern was essential to clearly separate the data access logic from the business logic. This separation improves modularity, facilitates maintenance and allows code reuse. In addition, the use of the Faker library to generate realistic test data was decisive to simulate operational scenarios, validate the correct functioning of CRUD operations and ensure that the system responded optimally to different workloads.

Another relevant point was the theoretical and practical validation of the model. The application of relational algebra operations confirmed that the design is capable of supporting complex queries, such as the identification of premium or featured users, which demonstrates the flexibility of the system to adapt to future needs and advanced queries. The correct execution of CRUD operations, from creating new records to tracking and updating user progress, confirms the effectiveness and comprehensive functionality of the database.

Finally, this project stands as a reference framework not only for replicating the data structure inspired by leading platforms such as Duolingo, but also for future developments and improvements. The combination of theoretical methodologies (such as ER and relational algebra) with modern practices (Docker, Python and design patterns) demonstrates that a multidisciplinary approach is essential to meet the challenges of handling large volumes of data and the constant evolution of web applications. This exercise not only brings value in the educational field, but also lays the foundation for the integration of new functionalities and data analysis systems in other contexts, ensuring a solid and adaptable foundation for the future growth of the platform.

## V. Bibliography

"Home." Docker Documentation. Accessed: Feb. 5, 2025. [Online]. Available: https://docs.docker.com/

"Faker documentation." Welcome to Faker's documentation! — Faker 36.1.1 documentation. Accessed: Feb. 5, 2025. [Online]. Available: https://faker.readthedocs.io/en/master/

"Psycopg2." PyPI. Accessed: Feb. 5, 2025. [Online]. Available: https://pypi.org/project/psycopg2/

"Mysql-connector." PyPI. Accessed: Feb. 5, 2025. [Online]. Available: https://pypi.org/project/mysql-connector/

"Fastapi." PyPI. Accessed: Feb. 5, 2025. [Online]. Available: https://pypi.org/project/fastapi/

"Uvicorn." PyPI. Accessed: Feb. 5, 2025. [Online]. Available: https://pypi.org/project/uvicorn/

freeCodeCamp. "How to set up a virtual environment in python – and why it's useful." freeCodeCamp.org. Accessed: Feb. 5, 2025. [Online]. Available: https://www.freecodecamp.org/news/how-to-setup-virtual-environments-in-python/