**CCT College Dublin**

**Assessment Cover Page**

|  |  |
| --- | --- |
| **Module Title:** | Databases - Approaches & Systems |
| **Assessment Title:** | CA 1 Integrated Databases and OCC (20%) |
| **Lecturer Name:** | **Taufique Ahmed** |
| **Student Full Name:** | Luis Ramirez, Telmuun dunia, Alejo Santos |
| **Student Number:** | 2023169, 2023306, 2023197. |
| **Assessment Due Date:** | Sunday, 15 December 2024, 11:59 PM |
| **Date of Submission:** |  |

**Declaration**

|  |
| --- |
| By submitting this assessment, I confirm that I have read the CCT policy on Academic Misconduct and understand the implications of submitting work that is not my own or does not appropriately reference material taken from a third party or other source. I declare it to be my own work and that all material from third parties has been appropriately referenced. I further confirm that this work has not previously been submitted for assessment by myself or someone else in CCT College Dublin or any other higher education institution. |

Contents

[Introduction. 3](#_Toc185188415)

[**Database Documentation.** 1](#_Toc185188416)

[Updated Components from checkpoint 1: 1](#_Toc185188417)

[Physical design: Produce the SQL and create statements for each relation, including INSERT statements (dummy data). 5](#_Toc185188418)

[**OOC Documentation.** 8](#_Toc185188419)

[**Conclusion.** 13](#_Toc185188420)

# **Introduction.**

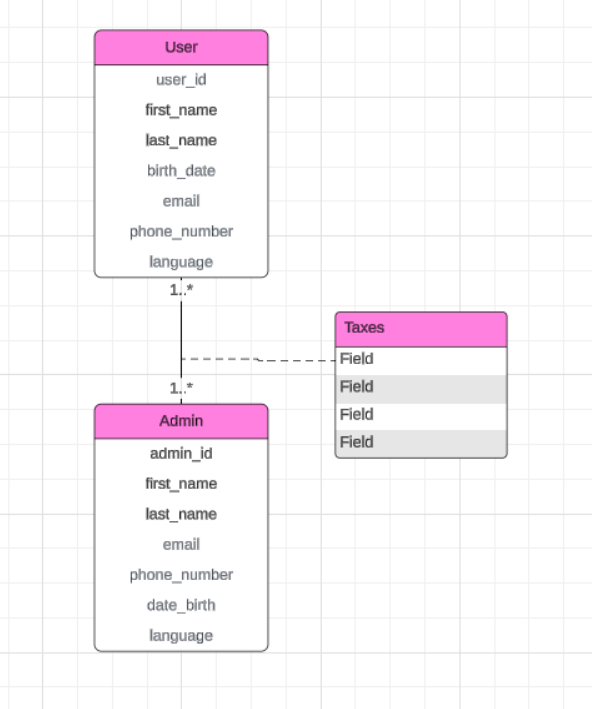
Nowadays, database services are as fundamental as a company's finance department, so the information stored within it must maintain and follow strict characteristics that maintain the integrity of both its security and its information. For this assessment, we will create a project using Java to create and connect to a MySQL database, in order to create roles such as administrator and users that can interact with it.

As we continue to develop this work, we will use different techniques to maintain both the structure and security of the database, such as data normalization, encapsulation, inheritance, and security processes.

# **Database Documentation.**

## Updated Components from checkpoint 1:

For this part what we did was create a new schema since there are 3 of us working on this project, so we decided to create a new one and work on it.



We started creating this from scratch.

After that, the next step was normalization. Starting from the 1NF, eliminating repeating groups in individual tables, creating the tables for each data, and lastly identifying the PK.

A diagram of a computer

Description automatically generated

After applying the 1NF: The tables looked like this:

Eliminates repeating groups in individual tables.

We eliminated the redundant attributes, both user and admin tables had identical fields like first\_name, last\_name, etc. This causes duplication. So, what we did instead we create a person table.

Each row is unique, and a primary key is introduced, such as a user\_id, person\_id, admin\_id, and tax\_id.

The taxes table had no relationship with other tables, no attributes, and no PK. We fixed that with the 1NF.

2NF:

A screenshot of a computer

Description automatically generated

The established relationship between every table:

**User table with person table.**

One-to-One (1:1): Each User has exactly one corresponding Person, and vice versa. This is enforced by the user\_id foreign key in the Person table.

**Person table with admin table.**

One-to-Many (1:N): A Person can be associated with many Admin records, but an Admin can only be associated with one Person. This is enforced by the admin\_id foreign key in the Person table.

**Admin table with taxes tables.**

Many-to-Many (M:N): An Admin can be assigned to many Taxes, and a Tax can be assigned to many Admins. This relationship is implemented using the TaxAssignment junction table.

**Taxes table with taxAssignment table.**

One-to-Many (1:N): A Tax can be assigned to many TaxAssignments, but a TaxAssignment can only be associated with one Tax. This is enforced by the tax\_id foreign key in the TaxAssignment table.

**User table with taxes**

Many-to-Many (M:N): A User can be assigned to many Taxes, and a Tax can be assigned to many Users. This relationship is implemented using the TaxAssignment junction table.

**Tax table with TaxPayment table.**

One-to-Many (1:N): A Tax can have many TaxPayments, but a TaxPayment can only be associated with one Tax. This is enforced by the tax\_id foreign key in the TaxPayment table.

2NF, for this task, we made sure that no partial dependency exists. We have already addressed many normalization issues in 1N. In 2NF, we further refine by ensuring that every non-key attribute depends on the entire primary key. Also, there are no redundant fields on any table.

So, the conclusion is every table has a single-column primary key. All non-key attributes depend on the entire primary key. But we decided to split the taxes tables in 3 since we had a few characteristics with potential dependencies in the taxes table that if you combine them with other data, could generate a duplicate. So, instead, we created 2 tables from taxes, one is “tax payments” and we created attributes as payment\_id as a PK tax\_id as FK and paid\_by as FK. And, the second table was called tax\_assignment with a composite key consisting of the tax\_id in the taxestable and assigned\_by referencing the user\_id in the usertable.

3NF:

In the 3rd formal notation, we had to analyze if there was any violation of 3NF.

We found that in the taxpayment table, the paid\_by attribute is dependent on the assigned\_by attribute, which is not a primary key. This a dependency, and this one was violating the 3NF rule.

Without separating the attribute assigned\_by attribute to another table the same user information could be repeated in many taxpayments records, and this could lead to a data redundancy. So, we figured this out by introducing a new table called “assigned user” by creating this table we eliminated the dependency and we reduced data redundancy. Which lead a more normalized and efficient database design.

Therefore, after all these updates the schema looked like this.

A diagram of a software

Description automatically generated with medium confidence

## Physical design: Produce the SQL and create statements for each relation, including INSERT statements (dummy data).

For this task we put all the data inside the MySQL Workbench, creating the tables and attributes and also putting dummy information as an example.

Dummy data on person table.

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generatedDummy data on tax payment table.

A screenshot of a computer

Description automatically generated

Admin table.

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated User table.

# **OOC Documentation.**

A diagram of a company

Description automatically generated

Figure 1 - Above is a flow chart indicating the program architecture

The user login classes and database setup classes all inherit and extend the DB\_Connection class to utilize the information stored in the father class. We aimed to store as little logic in the main class as possible, and only call methods and properties from the other classes and implement the menu structure in the main class.

A blue background with white dots

Description automatically generated

Figure 2 - User and Admin has been created as a seperate class with their own son Login classes to handle prompting the DB to match the data.

A diagram of a server

Description automatically generatedTo create a connection between the database and the Java file, we had to inherit attributes from the DB\_Connection file as a parent, using the extends keyword.

This java inheritance is because the DB\_Connection file contains all relevant information about the database like the user, password, table names and the database URL (name and port to connect). When a class try to modify, upload or delete information stored in the database, it has to first create the connection with the right credentials, and then continue with the process.

One example is the DatabaseReader file, which is responsible for making sure that the database exists in MySQL application, and doesn’t crash using a while loop.

A screen shot of a computer code

Description automatically generated

Figure 3 - User login Action method to query the MySQL DB and check if the information given matches the information in the database.

A computer screen shot of text

Description automatically generated

Figure 4 - Add user method is also added with INSERT TO syntax that will create and add users to the Database.

A computer screen shot of a program

Description automatically generated

Figure 5 - All interactions with the MySQL DB is being made within code, with syntaxes such as CREATE, INSERT, SELECT AND DELETE

A computer screen shot of text

Description automatically generated

Figure 6 - Example of a method that will change the user data in a table, to be queried as a syntax

A computer code on a blue background

Description automatically generated

Figure 7 - Remove a user from the DB using syntax

A screen shot of a computer code

Description automatically generated

Figure 8 - UserHistory arraylist to store changes made to the Database and list them out when required.

A screen shot of a computer code

Description automatically generated

Figure 9 - Regex validation check for every possible input from the user.

# **Conclusion.**

When doing this evaluation we faced several challenges, for example, creating a connection between Java and the database using the MySQL language and making it possible to read, write and delete information in real time. Those are the basic functionalities for any programming project with a database, and finding all the resources and information was one of the difficult parts.

One of the advantages of using Java and MySQL for this was the amount of documentation that exists online, and that was one of the fundamental principles for solving problems we found along the way.

Using Java classes we managed to separate various functionalities into different files, and thus we were able to improve the security of the information, such as the class used for the Administrator, keeping its attributes private from the outside. We were able to identify when we can create a class to separate functionality from the main code, such as the REGEX validation class, in which we defined the values ​​we needed and what happened in case of an error in the input of a value.

In addition, we managed to work in a fluid and orderly manner by defining our roles based on what we knew best how to do or in which area we had better management of information and by using Github with netBeans we maintained an orderly and efficient work.