# A Low-code approach for enforcing Fine-Grained Access Control in Database-centric applications
## Final Project of Formal Model Driven Engineering

Student: Hoàng Nguyễn Phước Bảo

Universidad Autónoma de Madrid, Spain

**Abstract.** In this report, I propose a low-code approach to realize a model-driven proposal to enforce fine-grained access control in database-centric applications. The works involve designing the related metamodels and their domain specific languages, then, applying some model transformation and model manipulation techniques to generate final executable artifacts on the database level.

## 1 Introduction

*Model-Driven Engineering* (MDE) is a software development methodology that focuses on creating *models* of different views of a system, and then automatically generating different system artifacts from these models, such as code and configuration data. *Model-Driven Security* (MDS) is a specialization of model-driven engineering for developing secure systems. In a nutshell, designers specify system models along with their security requirements and use tools to automatically generate security-related system artifacts, such as access control infrastructures.

*SQL Security Injector* (SQLSI) is my first and latest work in the MDS discipline. The project attempts to enforce *fine-grained access control* (FGAC) in database using model-driven methodology. So far, this involves (i) to prospose a formal model to express such access control policy and a rigorous design to enforce it in database-centric application [1], then (ii) to realize the aforementioned work in a notably case study [2], finally, (iii) to prove its correctness. In regards to the second goal, since the inputs in the proposal are indeed models of different kinds, I believe it makes sense to apply a more *low-code* approach, taking advantage of the knowledge I have learnt from the course, to replace the current *ad-hoc* realization represented in [2].

*Organization* The rest of the report is organized as follows. In Section 2, I provide some main remarks about the SQLSI idea and design, including some related works and their drawbacks, from which uplift the approach in this report. Next, in Section 3, I describe the input and output of this new approach. Then, in Section 4, provides the in more detail the realization and technology used. Finally, in Section 5, I provide conclusion and possible future works.

## 2 Background and Motivation

### 2.1 Traditional approach vs SQLSI approach

Figure 1 shows the main difference between traditional and model-driven approach in enforcing FGAC in database-centric application. In particular:

*Traditional approach* On the left-hand side is the traditional approach to enforce FGAC in database-centric applications. Firstly, the policy will be manually implemented and enforced on the application level. Next, given a SQL statement, the application layer sends it to the data layer from which returns the result. Then, the application layer must enforce the FGAC policy to the result and finally return the answer to the client.

*SQLSI approach* On the right-hand side is the SQLSI model-based approach. This approach requires an application *data-model* that corresponds to the underlying database in the data layer and a *security-model* that defines the FGAC policy and respects the given data-model. The SQLSI component have two main functionalities:

(i) Given those data- and security-models, SQLSI component generates a set of SQL *authorization functions* which are executable in the database.
(ii) Then, instead of issuing directly the SQL statements from the application layer, SQLSI will *rewrite* these statements into semantically-equivalent SQL *secure stored-procedures* in which *injects* the given access control policy via means of aforementioned SQL *authorization functions* in (i).

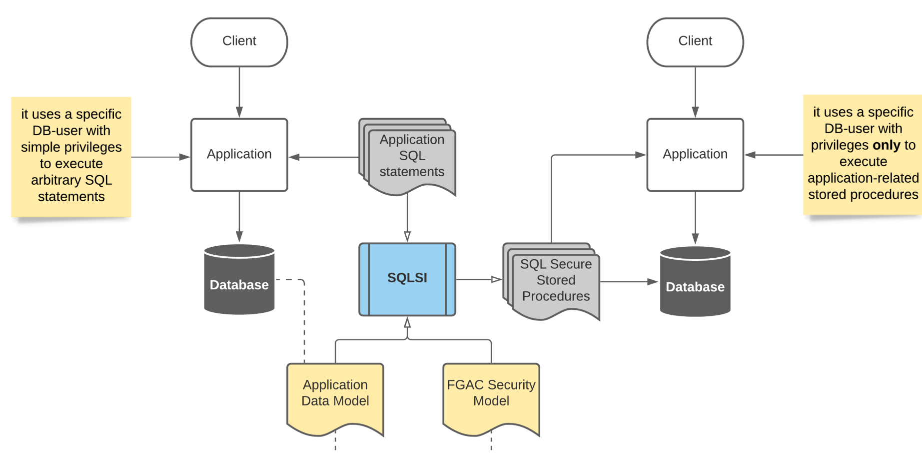



Fig. 1: Traditional approach vs. SQLSI approach

### 2.2 SQLSI realization vs low-code realization

In this paper, we focus on realizing the first functionality of SQLSI, in a more low-code fashion. To distinguish this with the former realization, let us call our new approach SQLSI`emf`. Then, the following work of SQLSI`emf` will be primary:

(i) Define the metamodel for data-model, security-model, and their domain-specific language.
(ii) Define customized-model of relational database schema and the transformation from data-model to this relational database schema model.
(iii) Define the code-generation from my relational database schema model to actual SQL database script.
(iv) Define the transformation from the *user-input* version to the *normalized* version of the security-model.
(v) Define the code-generation from security-model to a set of SQL authentication functions.

Figure 2 shows the workflow in a nutshell, including the technology used, whose description will be explained in Section 3 and Section 4.

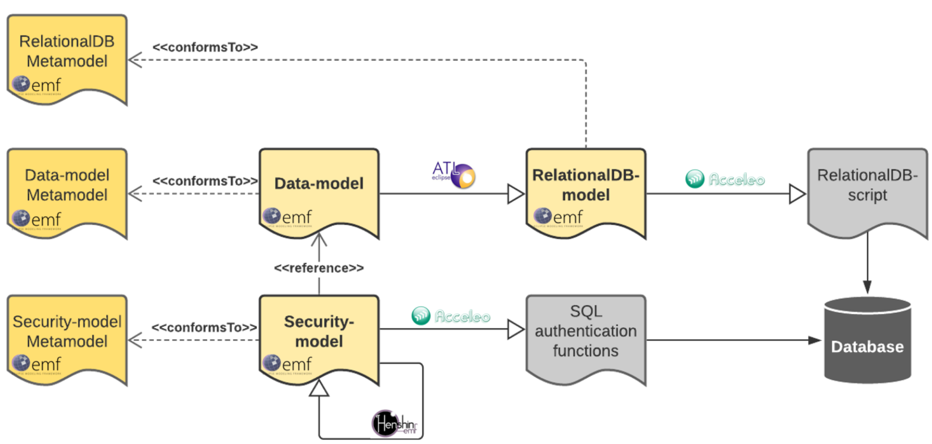



Fig. 2: Proposed approach - SQLSI`emf` in a nutshell

### 2.3 Related work

[2] presents a Java implementation of the whole SQLSI component, including the SQL authorization functions generation. In paritcular, the data-model and security-model are represented as Java object hierarchies and consequently, every model transformation and manipulation was implemented manually through the help of the visitor pattern. Indeed, this is a very ad-hoc approach and less modern, compared to the "one-click-of-a-button" implementation in this report.

In the past, there are some attempts to implement intelligent web-based editors for these data-model and security-model. In particular, these implementations are the final products of bachelor theses. On the one hand, these approaches are, again, very ad-hoc, since any change in the model requires (a handful of) manual changes in the source code. And on the other hand, due to the time allotted for the final thesis, the students do not have enough time to thoroughly understand the idea behind these models and therefore implemented in a way that is very hard to maintain or extend. In my approach, we avoid both drawbacks by using the available modelling technology, for instances, Xtext and ATL.

## 3 The SQLSI Metamodels

### 3.1 Input Metamodels

**Metamodel for data-models** For SQLSI`emf`, a data-model contains entities and associations between them. An entity may have properties which are attributes or associations-ends.

The data-model metamodel for SQLSI`emf` is shown in Figure 3. The `data-model` is the root element: it has a `name` and contains a set of `Entity`s. Every `Entity` represents an entity in the data model: it has a `name` and contains a set of `Property`(-ies). Every `Property` can be either an `Attribute` or an `AssociationEnd`.

– Each `Attribute` represents an attribute of the entity: it unique-ability depends on a boolean value `isUnique` and its `type` is either `String` or `Integer`.
– Each `AssociationEnd` represents an association between its container `Entity` with another `Entity`[1]: its `Multiplicity` is either `MANY` or `ONE`. Finally, each `AssociationEnd` is linked to its opposite `AssociationEnd`, and also with its `targetEntity`.

**Metamodel for security-models** For SQLSI`emf`, a security-model of a data-model contains a list of "fine-grained" rules which represent the authorization constraints for a data-model. Each rule formally states that under which *roles*, under which *actions*, under which *conditions*, the user can access which *resources*.

The security-model for SQLSI`emf` is shown in Figure 4. The `security-model` is the root element, it has a `name` and contains a set of `ProtectedResource`s, a set of `Rule`s and a set of `Role`s. Naturally, every instance of `security-model` is associated with an instance of `data-model`.

– Each `ProtectedResource` represents a resource that the security-model intend to enforce fine-grained access control checks. Every protected resource has a `name` and is linked to a `Property` of the referenced `data-model`.
– Each `Role` represents a role in the application: it has a `name`. Naturally, all the roles must referred to the *user*-`Entity` instance from the referenced `data-model`.

[1] not necessary be a different one

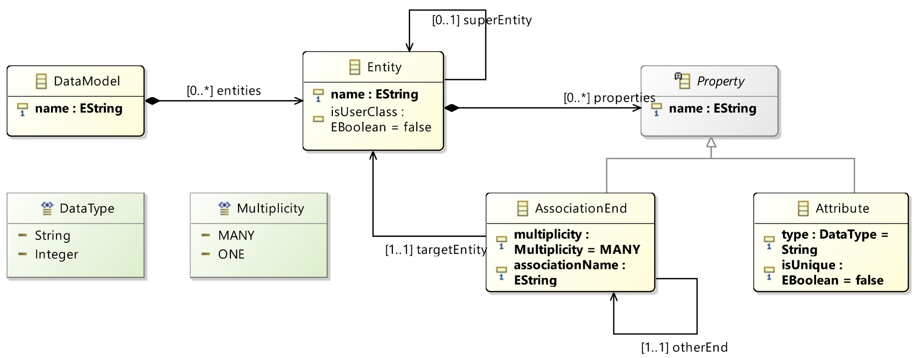



Fig. 3: SQLSI`emf` metamodel for data-models.

– Each `Rule` represents a FGAC-rule in the application: it has a `name` and accomodates with an `action` (chosen from CRUD). Furthermore, every rule also refers to the resources that it intends to protect along a set of `Auth`orization conditions, whose meaning is to state the scenarios when the *resource* is allowed to perform the *action*.[2]

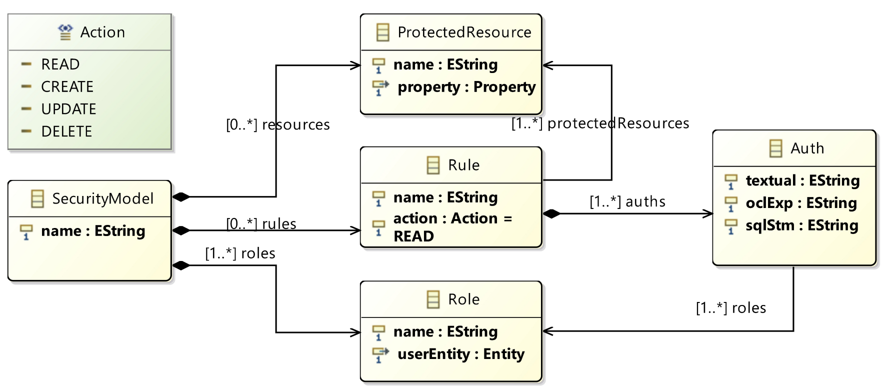



Fig. 4: SQLSI`emf` metamodel for security-models.

[2] In this model, we allow user to write authorization constraints under three different means: either textual, OCL constraint or SQL boolean statement. Furthermore, each authorization constraint can be enforced for a set of `Role`s.

### 3.2 Output Metamodels

**Metamodel for Relational Schema** The relational schema metamodel is shown in Figure 5. It is self-explanatory, therefore its descrption shall be omitted here to save some spaces.

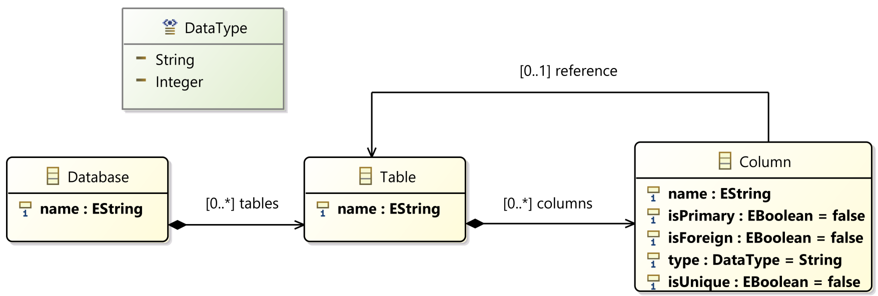



Fig. 5: SQLSI`emf` metamodel for relational database.

## 4 The SQLSI`emf` Language and Design

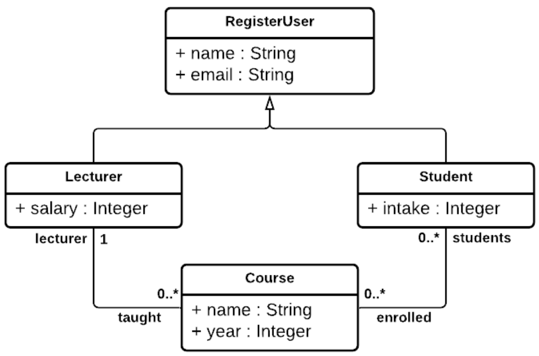
For the sake of clarity, let us continue this section of SQLSI`emf` language and design by associating it with a running example.

### 4.1 A Simple University Management System

Considering a simple UML class diagram `University` in Figure 6a containing four entities: `RegisterUser`, for representing register users; `Lecturer`, for representing lecturers; `Student`, for representing students; and `Course`, for representing the courses in the university. Every `RegisterUser` have a `name` and a unique `email`. A `Lecturer` is a `RegisterUser` with a `salary` attribute, for representing his/her monthly income. A `Student` is also a `RegisterUser` with a year `intake`. A `Course` has a `name` and a `year`. Every `Cource` is taught by exactly one `Lecturer` and can have none or many `Student`s. Additionally, every `Lecturer` can teach none or many `Course`s as well as every `Student` can enroll in none or many `Course`s.

### 4.2 Domain-Specific Language for the Input Models

**DSL for data-model** Figure 6b shows `University` written in our domain specific language. For the interested readers, the specification of this language can be seen in the repository. The additional features (e.g. validating, error-handling, scoping) will be leave for future works.




(a) using UML class diagram

```
DataModel University :
user entity RegUser {
    attribute name String,
    attribute email unique String
},
entity Student extends RegUser {
    attribute intake Integer,
    association Course[*] enrolled
        oppositeTo "Course.students" in Enrollment
},
entity Lecturer extends RegUser {
    attribute salary Integer,
    association Course[*] taught
        oppositeTo "Course.lecturer" in Teaching
},
entity Course {
    attribute name String,
    attribute year Integer,
    association Student[*] students
        oppositeTo "Student.enrolled" in Enrollment,
    association Lecturer[1] lecturer
        oppositeTo "Lecturer.taught" in Teaching
}
```

(b) using data-model DSL

Fig. 6: `University` representation

**DSL for security-model** Consider the following security-model for `UniversityDM`:

- Protected resources: `name` and `email` of the `RegisterUser`, `salary` of the `Lecturer` and `intake` year of the `Student`.[3]
- Role: There are three roles, namely, the role `Administrator`, `Lecturer` and `Student`.
- Permissions: each item represents a rule to enforce, in which is textually explained, follows by the OCL expression.
  - Any user can read any user's basic information. Formally,
    `true`
  - Administrator can read any student intake year. Formally,
    `true`
  - Student can read its classmates' intake year. Formally,
    `caller.enrolledCourses→exists(c|c.students→includes(self))` [4]
  - Lecturer can read his\her students' intake year. Formally,
    `caller.taughtCourses→exists(c|c.students→includes(self))`
  - Lecturer can read its own salary. Formally,
    `caller = self`

[3] Please note that the properties in `data-model` which does not appear hear simply mean that they are unprotected, i.e. they can be accessed by all means.

[4] This is an OCL constraint in which `self` represents the user (object) whose information is accessed and `caller` represents the user (subject) whose are accessing the information.

Figure 7 shows a fragment of the above security-model written in our domain specific language. For the interested readers, the specification of this language can be seen in the repository. The additional features (e.g. validating, error-handling, scoping) will be leave for future works.

```
Rule readStudentSpecificInfo {
    action READ (studentIntake)
    auths {
        roles (Administrator)
        condition: {
            textual "Administrator can read
            any student intake year"
            oclExp "true"
            sqlStm "TRUE"
        },
        roles (Student)
        condition: {
            textual "Student can read its classmates' intake year"
            oclExp "caller.enrolled->exists(c|c.students->includes(self))"
            sqlStm "caller = self"
        },
        roles (Lecturer)
        condition: {
            textual "Lecturer can read its students' intake year"
            oclExp "caller.taught->exists(c|c.students->includes(self))"
            sqlStm "EXISTS (SELECT 1
                FROM (SELECT * FROM teaching WHERE lecturer = caller) as TEMP1
                JOIN (SELECT * FROM enrollment WHERE students = self) as TEMP2
                ON TEMP1.taught = TEMP2.enrolled)"
        }
    }
},
```

Fig. 7: A snipper of `University` security-model

### 4.3 Model Transformation: from Data Model to Relational Schema

**Some transformation remarks** Since transformation from UML class diagram to relational database is widely considered as a "Hello world!" example in the world of model transformation. The detail description of this section have been reduced to save some spaces for other interesting parts.

In short, our model transformation from `data-model` to `RelationalDatabase` is implemented in ATLAS Transformation Language. It is very simple and indeed very similar to the example provided in the course. The main difference is that: during the transformation, it (i) creates a Role-`Table` and (ii) link this with the `Table` that correspond to the user-`Entity` via an association `Table`.

**Code generation: Relational database to executable script** In addtion to the above transformation, we also provide a database schema generation using

Acceleo. In this current version, we only support database schema generation that is executable on MySQL Server[5] but it can be easily extended to other language by create different configuration files.[6] For interested readers, Appendix A shows the generated `University` MySQL database schema script.

### 4.4 Model Manipulation: Security Model

**Rationale** From the user's perspective, it is most often easier to create an FGAC rule of structure: "*role r* is able to perform *action a* on *resource p* under *condition c*". In contrast, from the system's perspective, it is easier to enforce the FGAC policy if all the rules follow this order: *action*-*resource*-*role*-*condition*. For this reason, a model manipulation for security-model is needed. Since refining is not well-supported in ATLAS Transformation Language, Henshin has been chose for this task.

**Remarks** In short, the manipulation procedure is as follows:

1. For every authentication condition that applies for two different roles, we split those roles by create a copy of the original authentication condition. This will be applied exhaustively.
2. Next, for every rule that contains more than one authentication condition, we split those conditions by create a copy of the original rule. This will be applied exhaustively.
3. Then, for every rule that protects more than one resource, we split those protection by create a copy of the original rule. This will be applied exhaustively.
4. Finally, for every rule that protects the same resource and have the condition that applies for the same role, we merge them together.

Due to the space limit, the main unit is displayed in Figure 8 . For interested readers, the full transformation can be visited in the repository.


Fig. 8: The main unit in our transformation

[5] Working version: 8.0.16 MySQL Community Server.

[6] Since the relational database meta-model respects the standard of [3], we should be able to provide generation for any given relational database management systems (i.e. Orable, PostgreSQL, etc.), as long as it follows the same standard.

**Code generation from Security Model to a Secure Authorization SQL-function** In addtion to the above transformation, we also provide a SQL authorization functions code generation using Acceleo. In this current version, we only support SQL code generation that is executable on MySQL Server[7]For interested readers, Appendix B shows the generated `University` SQL authorization functions artifact.

## 5 Conclusions and Future Work

In this report, I have proposed a novel approach to enforcing FGAC policy in database-centric applications.

In regards to the course, this project covers a large materials being taught throughout the whole semester. During this period, in addition to the lectures' notes, the Eclipse community has been a very good source when it comes to some specific, difficult questions. With the total credit points of six, I under-estimated the complexity of the proposed project and hence, spent more time to complete it than I should have done.

In regards to the line of research, this novel approach is more modern. This project opens up many interesting future works, one of which can eventually turn into the master's thesis.

**Future works** The following items describe potential future works, order of importance.

- As one may notice in the sample use-case in Appendix **??**, the project is not complete, in the sense that there are still implementation errors in some of the execution steps. Therefore, the highest priority is to find solutions to these unsolve problems.
- Also, in the sense of usability, there are many things that can be done. For example, one can implement additional features for the DSLs' editors or implement a pop-up menu in Eclipse for some of the execution steps.
- The availability of metamodels certainly opens up more possibility of mapping between these models with other formalism, e.g. to first-order logic. This is very important since it is needed in order to prove the correctness of the application.
- Inspired by the SecureUML metamodel, the security-model in this paper is the simplify version of it. On the one hand, improving to the complete version of the metamodel could be one potential work. On the other hand, the transformation from the security model to its normalized version also seems to be very interesting, and I believe it should fit great (as a contest proposal) for the Transformation Tool Contest.[8]

[7] Working version: 8.0.16 MySQL Community Server.

[8] Transformation Tool Contest: A contest for users and developers of transformation tools. Part of the Software Technologies: Applications and Foundations (STAF) federated conferences.

## References


1. H. N. P. Bao and M. Clavel. Model-based characterization of fine-grained access control authorization for SQL queries. *J. Object Technol.*, 19(3):3:1–13, 2020.
2. H. N. P. Bao and M. Clavel. A model-driven approach for enforcing fine-grained access control for SQL queries. In T. K. Dang, J. Küng, M. Takizawa, and T. M. Chung, editors, *Future Data and Security Engineering - 7th International Conference, FDSE 2020, Quy Nhon, Vietnam, November 25-27, 2020, Proceedings*, volume 12466 of *Lecture Notes in Computer Science*, pages 67–86. Springer, 2020.
3. ISO/IEC 9075-(1–10) Information technology – Database languages – SQL. Technical report, International Organization for Standardization, 2011. http://www.iso.org/iso/.


## A Sample generated Relational Database script

```
DROP DATABASE IF EXISTS UNIVERSITY;
CREATE DATABASE UNIVERSITY;
USE UNIVERSITY;

DROP TABLE IF EXISTS Role;
CREATE TABLE Role(
RoleID INT(11) PRIMARY KEY,
name VARCHAR(200) UNIQUE
);
DROP TABLE IF EXISTS RegUser;
CREATE TABLE RegUser(
RegUserID INT(11) PRIMARY KEY,
name VARCHAR(200),
email VARCHAR(200) UNIQUE
);
DROP TABLE IF EXISTS Student;
CREATE TABLE Student(
StudentID INT(11) PRIMARY KEY,
RegUserID INT(11) UNIQUE,
FOREIGN KEY (RegUserID) REFERENCES RegUser(RegUserID),
intake INT(11)
);
DROP TABLE IF EXISTS Lecturer;
CREATE TABLE Lecturer(
LecturerID INT(11) PRIMARY KEY,
RegUserID INT(11) UNIQUE,
FOREIGN KEY (RegUserID) REFERENCES RegUser(RegUserID),
salary INT(11)
);
DROP TABLE IF EXISTS Course;
CREATE TABLE Course(
CourseID INT(11) PRIMARY KEY,
name VARCHAR(200),
year INT(11)
```

```
);
DROP TABLE IF EXISTS RegUserRole;
CREATE TABLE RegUserRole(
role INT(11) UNIQUE,
FOREIGN KEY (role) REFERENCES Role(RoleID),
RegUserRole INT(11) UNIQUE,
FOREIGN KEY (RegUserRole) REFERENCES RegUser(RegUserID)
);
DROP TABLE IF EXISTS Teaching;
CREATE TABLE Teaching(
taught INT(11) UNIQUE,
FOREIGN KEY (taught) REFERENCES Course(CourseID),
lecturer INT(11) UNIQUE,
FOREIGN KEY (lecturer) REFERENCES Lecturer(LecturerID)
);
DROP TABLE IF EXISTS Enrollment;
CREATE TABLE Enrollment(
students INT(11) UNIQUE,
FOREIGN KEY (students) REFERENCES Student(StudentID),
enrolled INT(11) UNIQUE,
FOREIGN KEY (enrolled) REFERENCES Course(CourseID)
);
```

# B Sample generated authorization functions

```
USE UNIVERSITY;

DROP FUNCTION IF EXISTS auth_READ_regUserEmail;
DELIMITER //
CREATE FUNCTION auth_READ_regUserEmail
(self INT(11), caller INT(11), role VARCHAR(200))
RETURNS INT DETERMINISTIC
BEGIN
DECLARE result INT DEFAULT 0;
IF (role = 'Administrator')
THEN RETURN auth_READ_regUserEmail_Administrator(self, caller);
END IF;
IF (role = 'Lecturer')
THEN RETURN auth_READ_regUserEmail_Lecturer(self, caller);
END IF;
IF (role = 'Student')
THEN RETURN auth_READ_regUserEmail_Student(self, caller);
END IF;
END //
DELIMITER ;

DROP FUNCTION IF EXISTS auth_READ_regUserName;
DELIMITER //
CREATE FUNCTION auth_READ_regUserName
```

```
(self INT(11), caller INT(11), role VARCHAR(200))
RETURNS INT DETERMINISTIC
BEGIN
DECLARE result INT DEFAULT 0;
IF (role = 'Administrator')
THEN RETURN auth_READ_regUserName_Administrator(self, caller);
END IF;
IF (role = 'Lecturer')
THEN RETURN auth_READ_regUserName_Lecturer(self, caller);
END IF;
IF (role = 'Student')
THEN RETURN auth_READ_regUserName_Student(self, caller);
END IF;
END //
DELIMITER ;

DROP FUNCTION IF EXISTS auth_READ_lecturerSalary;
DELIMITER //
CREATE FUNCTION auth_READ_lecturerSalary
(self INT(11), caller INT(11), role VARCHAR(200))
RETURNS INT DETERMINISTIC
BEGIN
DECLARE result INT DEFAULT 0;
IF (role = 'Administrator')
THEN RETURN auth_READ_lecturerSalary_Administrator(self, caller);
END IF;
IF (role = 'Lecturer')
THEN RETURN auth_READ_lecturerSalary_Lecturer(self, caller);
END IF;
IF (role = 'Student')
THEN RETURN auth_READ_lecturerSalary_Student(self, caller);
END IF;
END //
DELIMITER ;

DROP FUNCTION IF EXISTS auth_READ_studentIntake;
DELIMITER //
CREATE FUNCTION auth_READ_studentIntake
(self INT(11), caller INT(11), role VARCHAR(200))
RETURNS INT DETERMINISTIC
BEGIN
DECLARE result INT DEFAULT 0;
IF (role = 'Administrator')
THEN RETURN auth_READ_studentIntake_Administrator(self, caller);
END IF;
IF (role = 'Lecturer')
THEN RETURN auth_READ_studentIntake_Lecturer(self, caller);
END IF;
IF (role = 'Student')
THEN RETURN auth_READ_studentIntake_Student(self, caller);
```

```
END IF;
END //
DELIMITER ;

DROP FUNCTION IF EXISTS auth_READ_regUserEmail_Administrator;
DELIMITER //
CREATE FUNCTION auth_READ_regUserEmail_Administrator
(self INT(11), caller INT(11))
RETURNS INT DETERMINISTIC
BEGIN
DECLARE result INT DEFAULT 0;
SELECT res INTO result FROM (SELECT 0  OR (TRUE)) AS TEMP;
RETURN result;
END //
DELIMITER ;

DROP FUNCTION IF EXISTS auth_READ_regUserEmail_Lecturer;
DELIMITER //
CREATE FUNCTION auth_READ_regUserEmail_Lecturer
(self INT(11), caller INT(11))
RETURNS INT DETERMINISTIC
BEGIN
DECLARE result INT DEFAULT 0;
SELECT res INTO result FROM (SELECT 0  OR (TRUE)) AS TEMP;
RETURN result;
END //
DELIMITER ;

DROP FUNCTION IF EXISTS auth_READ_regUserEmail_Student;
DELIMITER //
CREATE FUNCTION auth_READ_regUserEmail_Student
(self INT(11), caller INT(11))
RETURNS INT DETERMINISTIC
BEGIN
DECLARE result INT DEFAULT 0;
SELECT res INTO result FROM (SELECT 0  OR (TRUE)) AS TEMP;
RETURN result;
END //
DELIMITER ;

DROP FUNCTION IF EXISTS auth_READ_regUserName_Administrator;
DELIMITER //
CREATE FUNCTION auth_READ_regUserName_Administrator
(self INT(11), caller INT(11))
RETURNS INT DETERMINISTIC
BEGIN
DECLARE result INT DEFAULT 0;
SELECT res INTO result FROM (SELECT 0  OR (TRUE)) AS TEMP;
RETURN result;
END //
```

```
DELIMITER ;

DROP FUNCTION IF EXISTS auth_READ_regUserName_Lecturer;
DELIMITER //
CREATE FUNCTION auth_READ_regUserName_Lecturer
(self INT(11), caller INT(11))
RETURNS INT DETERMINISTIC
BEGIN
DECLARE result INT DEFAULT 0;
SELECT res INTO result FROM (SELECT 0  OR (TRUE)) AS TEMP;
RETURN result;
END //
DELIMITER ;

DROP FUNCTION IF EXISTS auth_READ_regUserName_Student;
DELIMITER //
CREATE FUNCTION auth_READ_regUserName_Student
(self INT(11), caller INT(11))
RETURNS INT DETERMINISTIC
BEGIN
DECLARE result INT DEFAULT 0;
SELECT res INTO result FROM (SELECT 0  OR (TRUE)) AS TEMP;
RETURN result;
END //
DELIMITER ;

DROP FUNCTION IF EXISTS auth_READ_lecturerSalary_Administrator;
DELIMITER //
CREATE FUNCTION auth_READ_lecturerSalary_Administrator
(self INT(11), caller INT(11))
RETURNS INT DETERMINISTIC
BEGIN
DECLARE result INT DEFAULT 0;
SELECT res INTO result FROM (SELECT 0) AS TEMP;
RETURN result;
END //
DELIMITER ;

DROP FUNCTION IF EXISTS auth_READ_lecturerSalary_Lecturer;
DELIMITER //
CREATE FUNCTION auth_READ_lecturerSalary_Lecturer
(self INT(11), caller INT(11))
RETURNS INT DETERMINISTIC
BEGIN
DECLARE result INT DEFAULT 0;
SELECT res INTO result FROM (SELECT 0  OR (TRUE)) AS TEMP;
RETURN result;
END //
DELIMITER ;
```

```
DROP FUNCTION IF EXISTS auth_READ_lecturerSalary_Student;
DELIMITER //
CREATE FUNCTION auth_READ_lecturerSalary_Student
(self INT(11), caller INT(11))
RETURNS INT DETERMINISTIC
BEGIN
DECLARE result INT DEFAULT 0;
SELECT res INTO result FROM (SELECT 0) AS TEMP;
RETURN result;
END //
DELIMITER ;

DROP FUNCTION IF EXISTS auth_READ_studentIntake_Administrator;
DELIMITER //
CREATE FUNCTION auth_READ_studentIntake_Administrator
(self INT(11), caller INT(11))
RETURNS INT DETERMINISTIC
BEGIN
DECLARE result INT DEFAULT 0;
SELECT res INTO result FROM (SELECT 0  OR (TRUE)) AS TEMP;
RETURN result;
END //
DELIMITER ;

DROP FUNCTION IF EXISTS auth_READ_studentIntake_Lecturer;
DELIMITER //
CREATE FUNCTION auth_READ_studentIntake_Lecturer
(self INT(11), caller INT(11))
RETURNS INT DETERMINISTIC
BEGIN
DECLARE result INT DEFAULT 0;
SELECT res INTO result FROM
(SELECT 0  OR (EXISTS
(SELECT 1
 FROM (SELECT * FROM teaching WHERE lecturer = caller) as TEMP1
 JOIN (SELECT * FROM enrollment WHERE students = self) as TEMP2
 ON TEMP1.taught = TEMP2.enrolled)
)
) AS TEMP;
RETURN result;
END //
DELIMITER ;

DROP FUNCTION IF EXISTS auth_READ_studentIntake_Student;
DELIMITER //
CREATE FUNCTION auth_READ_studentIntake_Student
(self INT(11), caller INT(11))
RETURNS INT DETERMINISTIC
BEGIN
DECLARE result INT DEFAULT 0;
```

```
SELECT res INTO result FROM (SELECT 0  OR (caller = self)) AS TEMP;
RETURN result;
END //
DELIMITER ;
```

## C Use-case: A simple University Management System