2023

Design Manual

GROUP G

EE468 DATABASE SYSTEMS

Table of Contents

Design Overview

System Goals and Required Functionalities

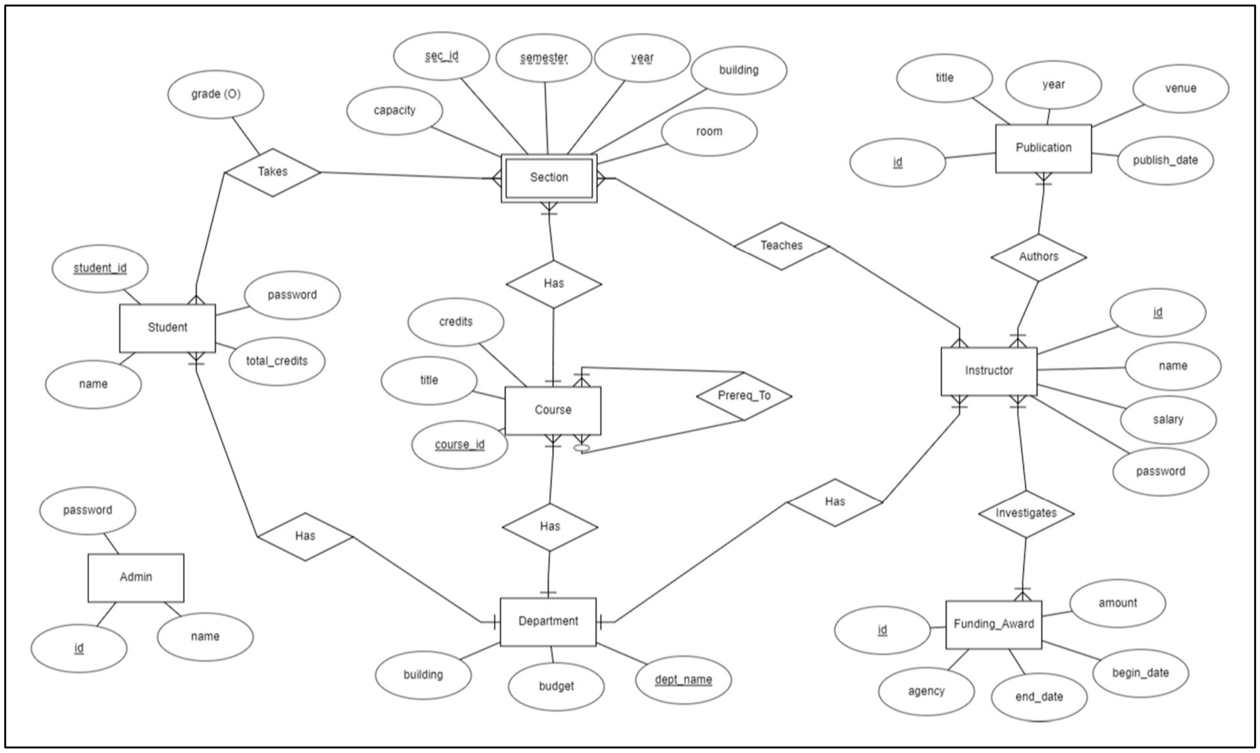
Below are our goals and required functionalities of this systems:

- Login Page for all users to access functionalities.
- All six (FR1 to FR6) feature requests work as expected.
- Fast and optimized database queries
- Presentable user interface and good user experience
- Prone to any security attack

Design Process

- Update the database schema and generate the model classes.
- Design queries are needed for all six features.
- Generate Django URLs needed for all features.
- Design Django templates to take inputs and show query results.
- Write view functions to process input and execute SQL queries.
- Ensure all security measures are in place.

ER Diagram



Database Schema

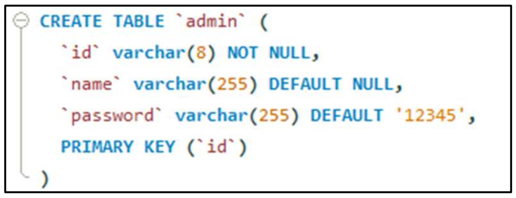
Due to the new requirements given we made the following changes to the existing university database system:

- Added 'admin' table.
- Added 'password' column for student and instructor (professor) tables.
- Added 4 new tables for feature.

Below is the schema for all new tables and the final database schema diagram exported from MySQL Workbench.

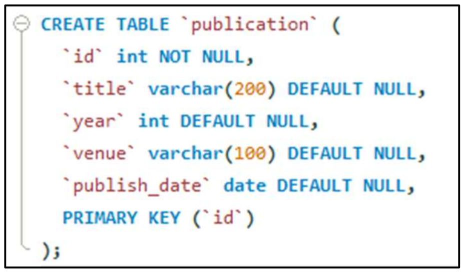
Schema for New Tables

Table "admin"

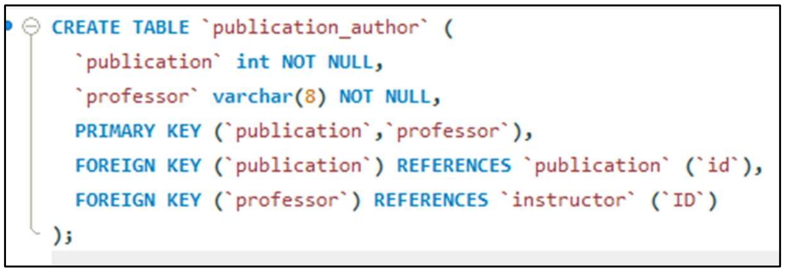


```
CREATE TABLE `admin` (
  `id` varchar(8) NOT NULL,
  `name` varchar(255) DEFAULT NULL,
  `password` varchar(255) DEFAULT '12345',
  PRIMARY KEY (`id`)
)
```

Table "publication" and "publication_author"

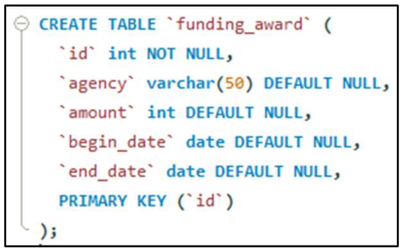


```
CREATE TABLE `publication` (
  `id` int NOT NULL,
  `title` varchar(200) DEFAULT NULL,
  `year` int DEFAULT NULL,
  `venue` varchar(100) DEFAULT NULL,
  `publish_date` date DEFAULT NULL,
  PRIMARY KEY (`id`)
);
```

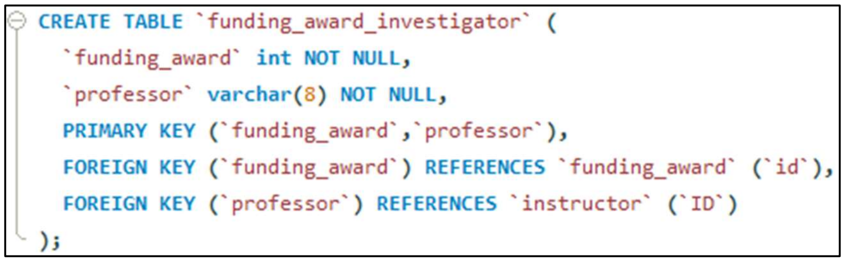


```
CREATE TABLE `publication_author` (
  `publication` int NOT NULL,
  `professor` varchar(8) NOT NULL,
  PRIMARY KEY (`publication`,`professor`),
  FOREIGN KEY (`publication`) REFERENCES `publication` (`id`),
  FOREIGN KEY (`professor`) REFERENCES `instructor` (`ID`)
);
```

Table "funding_award" and "funding_award_investigator"



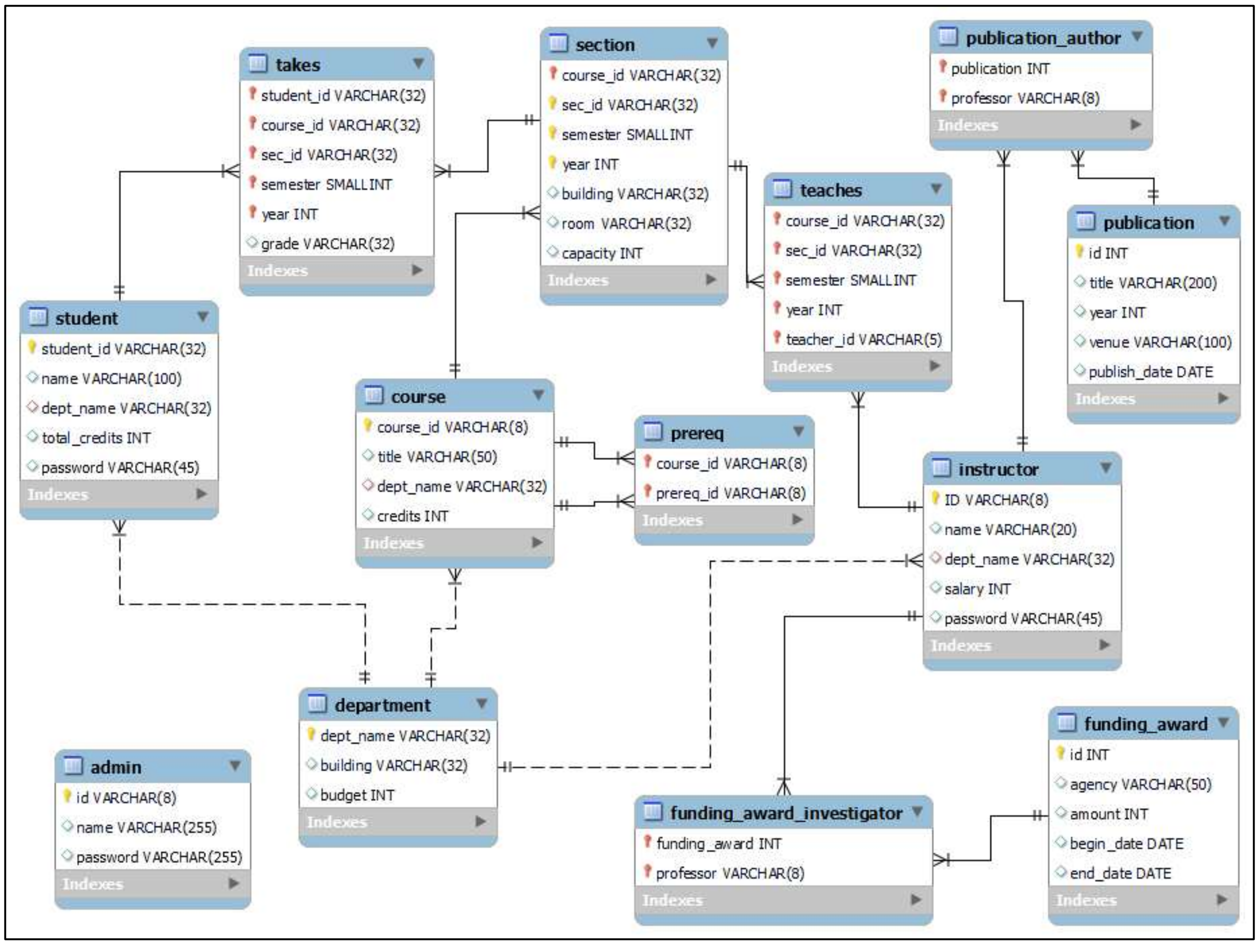
```
CREATE TABLE `funding_award` (
  `id` int NOT NULL,
  `agency` varchar(50) DEFAULT NULL,
  `amount` int DEFAULT NULL,
  `begin_date` date DEFAULT NULL,
  `end_date` date DEFAULT NULL,
  PRIMARY KEY (`id`)
);
```



```
CREATE TABLE `funding_award_investigator` (
  `funding_award` int NOT NULL,
  `professor` varchar(8) NOT NULL,
  PRIMARY KEY (`funding_award`,`professor`),
  FOREIGN KEY (`funding_award`) REFERENCES `funding_award` (`id`),
  FOREIGN KEY (`professor`) REFERENCES `instructor` (`ID`)
);
```

In addition to the new table schema above, Django framework also creates its own internal table schema for several functionalities. We made use of only the table '**django_session**' to store session variables.

Database Schema Diagram



Security Assurance Process

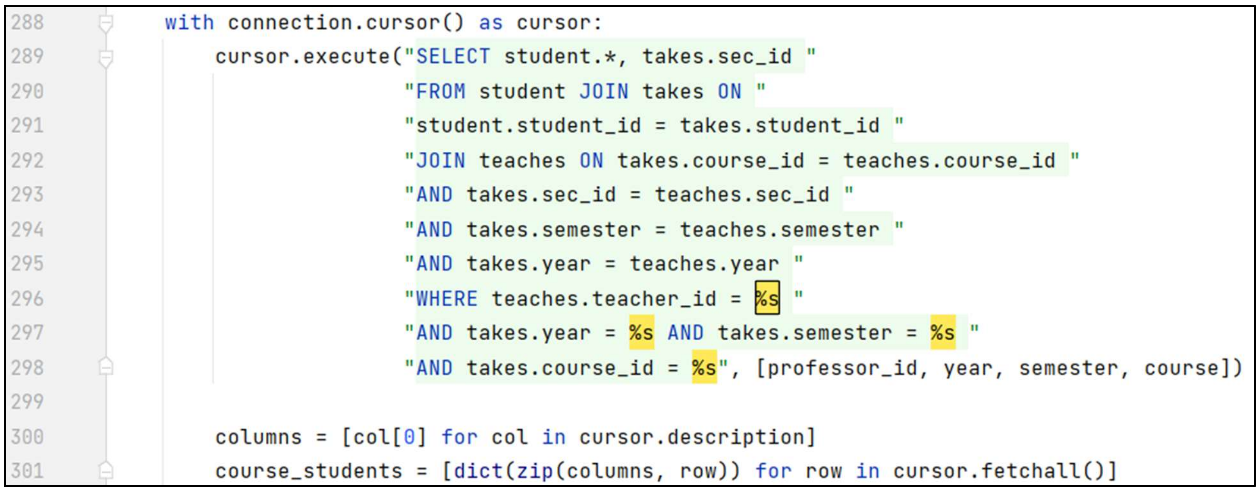
To ensure the university system is not vulnerable from any attack, the following security measures were taken:

SQL Injection Protection

In SQL injection attack, a malicious user can run any SQL code they want on a database. This may lead to the deletion of records or the disclosure of data. Since Django's queries are built using query parameterization, the queries are protected from SQL injection. The parameters of a query are declared separately from the SQL code for that query. The underlying database driver escapes parameters because human input makes them potentially hazardous.

In addition, we exercise extra caution when building raw queries. We made sure all user inputs are parameterized so they can be escaped by Django.

The screenshot below shows an example in Feature 5:



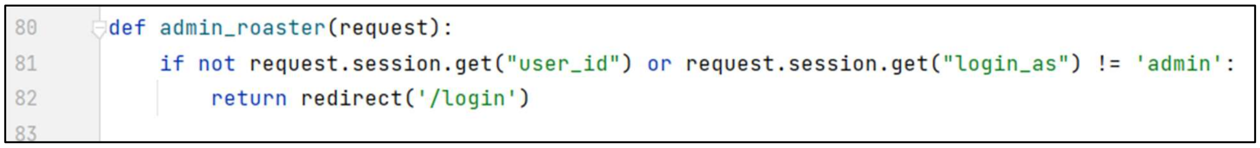
```
    with connection.cursor() as cursor:
        cursor.execute("SELECT student.*, takes.sec_id "
                       "FROM student JOIN takes ON "
                       "student.student_id = takes.student_id "
                       "JOIN teaches ON takes.course_id = teaches.course_id "
                       "AND takes.sec_id = teaches.sec_id "
                       "AND takes.semester = teaches.semester "
                       "AND takes.year = teaches.year "
                       "WHERE teaches.teacher_id = %s "
                       "AND takes.year = %s AND takes.semester = %s "
                       "AND takes.course_id = %s", [professor_id, year, semester, course])

        columns = [col[0] for col in cursor.description]
        course_students = [dict(zip(columns, row)) for row in cursor.fetchall()]
```

Secure Web Pages

The system has three users with distinct features. To protect against unauthorized access, every user must pass through a login page. Upon successful login, session variables are saved and will be verified for every page request. When a user tries to access an unauthorized page, the request will be redirected to the login page.

The code below shows an example of securing Feature 1:



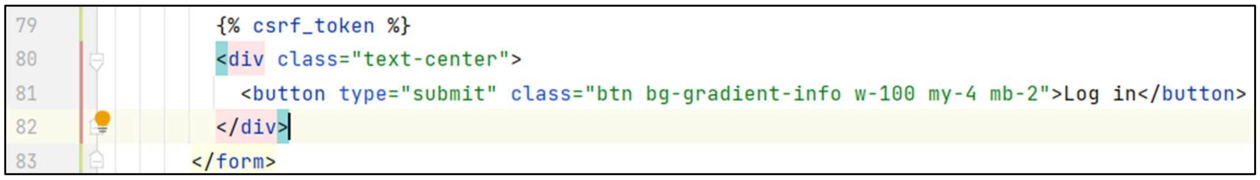
```
def admin_roaster(request):
    if not request.session.get("user_id") or request.session.get("login_as") != 'admin':
        return redirect('/login')
```

Cross Site Request Forgery (CSRF) protection

Through CSRF attacks, a malicious user can carry out operations using another user's login credentials without that user's knowledge or consent. Django features built-in defenses against most CSRF attacks. We enabled them and used them for all login forms that send a ‘POST’ request.

Every POST request is examined for the presence of a secret code to prevent CSRF attacks. Because of this, a malicious user cannot "resend" a form POST to our Django system and unintentionally causes another logged-in user to submit that form. The secret code, which is user-specific (by utilizing a cookie), would need to be known by the malevolent user.

The code below shows usage of CSRF protection in login form:



```
        {% csrf_token %}
        <div class="text-center">
          <button type="submit" class="btn bg-gradient-info w-100 my-4 mb-2">Log in</button>
        </div>
      </form>
```