Project 1

CPSC 449-01 Fall 2023

Ryan Novoa
Edwin Peraza
Abhinav Singh
Divya Tanwar
Chase Walsh
Gaurav Warad

Task 1: Database Schema and SQL Script

What is to be done:

- Create a database schema to manage classes, users, sections, and registrations.
- Write an SQL script to create tables, insert sample data, and define foreign key relationships.

Tools Used:

- Sqlite3 module from the Python Standard Library

Results:

- Database schema defined in '`classes.sql`' script.
- Tables created for Users, Class, Section, and RegistrationList

```
-- Create the Users table
DROP TABLE IF EXISTS Users;
CREATE TABLE IF NOT EXISTS Users (
CWID INTEGER PRIMARY KEY AUTOINCREMENT,
Name TEXT NOT NULL,
Middle TEXT NULL,
LastName TEXT NOT NULL,
Role TEXT NOT NULL CHECK (role IN ('instructor', 'registrar', 'student'))
);

-- Create the Class table
DROP TABLE IF EXISTS Class;
CREATE TABLE IF NOT EXISTS Class (
CourseCode TEXT PRIMARY KEY,
Name TEXT NOT NULL,
Department TEXT NOT NULL
);

-- Create the Section table
DROP TABLE IF EXISTS Section;
CREATE TABLE IF NOT EXISTS Section (
```

```sql
SectionNumber INTEGER NOT NULL,

CourseCode TEXT NOT NULL,

InstructorID INTEGER NOT NULL,

CurrentEnrollment INTEGER NOT NULL,

MaxEnrollment INTEGER NOT NULL,

Waitlist INTEGER NOT NULL,

SectionStatus TEXT NOT NULL CHECK (SectionStatus IN ('open', 'closed')),

PRIMARY KEY (SectionNumber, CourseCode),

FOREIGN KEY (CourseCode) REFERENCES Class (CourseCode),

FOREIGN KEY (InstructorID) REFERENCES Users (CWID)

);



-- Create the RegistrationList table

DROP TABLE IF EXISTS RegistrationList;

CREATE TABLE IF NOT EXISTS RegistrationList (

RecordID INTEGER PRIMARY KEY AUTOINCREMENT,

StudentID INTEGER NOT NULL,

CourseCode TEXT NOT NULL,

SectionNumber INTEGER NOT NULL,

EnrollmentDate DATETIME DEFAULT (CURRENT_TIMESTAMP),

Status TEXT NOT NULL CHECK (Status IN ('enrolled', 'waitlisted',
'dropped')),

FOREIGN KEY (StudentID) REFERENCES Users (CWID),

FOREIGN KEY (CourseCode, SectionNumber) REFERENCES Section (CourseCode,
SectionNumber)

);
```

- `CWID`: Unique identifier for each user
- `Name, Middle, and LastName`: These columns store the users' name
- `Role`: Indicates whether the user is an instructor, registrar, or student.
- `CourseCode`: Unique identifier for each course, used as primary key
- `Name`: Name of the course

    - `Department:` Indicates the department to which the course belongs
    - `SectionNumber:` Unique identifier for each section within a course
    - `CourseCode:` Foreign key referencing the Class table, indicating which course the section belongs to
    - `InstructorID:` Foreign key referencing the Users table, indicating the instructor for the section
    - `CurrentEnrollment:` Current number of student enrolled in the section
    - `MaxEnrollment:` Maximum number of students allowed to enroll in the section
    - `Waitlist:` Number of students on the waitlist for the section
    - `SectionStatus:` Indicates whether the section is open or closed for enrollment
    - `RecordID:` Unique identifier for each registration record
    - `StudentID:` Foreign key referencing the Users table, indicating the student who registered
    - `CourseCode:` Foreign key referencing the Section table
    - `SectionNumber:` Foreign key referencing the Section table
    - `EnrollmentDate:` Date and time when enrollment occurred
- Foreign key relationships established between tables.
- Sample data inserted into the tables.

```sql
-- pre populate database

-- chatGPT was used to generate some of the data

-- Users Table

INSERT INTO Users (Name, Middle, LastName, Role) VALUES

('Emily', NULL, 'Davis' ,'registrar'),

('John', 'A.', 'Smith', 'instructor'),

('Jane', 'M.', 'Doe' ,'instructor'),

('Robert', 'E.', 'Johnson', 'instructor'),

('Mark', 'B.', 'Johnson', 'instructor'),

('Catherine', 'E.', 'Wilson', 'instructor'),

('Matthew', 'J.', 'Davis', 'instructor'),

('Jennifer', 'R.', 'Harris', 'instructor'),

('Kevin', 'W.', 'Smith', 'instructor'),

('Linda', 'A.', 'Williams', 'instructor'),

('Michael', 'J.', 'Wilson', 'student'),
('Susan', 'K.', 'Brown', 'student'),

('David', 'P.', 'Miller', 'student'),

('Jennifer', NULL, 'Clark', 'student'),

('Richard', 'R.', 'White', 'student'),

('Sarah', 'L.', 'Anderson', 'student'),

('William', 'T.', 'Lee', 'student'),

('Karen', NULL, 'Martinez', 'student'),

('Thomas', 'S.', 'Taylor', 'student'),

('Laura', 'M.', 'Garcia', 'student'),

('Steven', NULL, 'Harris', 'student'),

('Daniel', 'M.', 'Wilson', 'student'),

('Michelle', 'L.', 'Johnson', 'student'),

('Christopher', 'S.', 'Brown', 'student'),

('Jessica', NULL, 'Anderson', 'student'),

('Jason', 'D.', 'Turner', 'student'),

('Melissa', NULL, 'Adams', 'student'),

('Paul', 'R.', 'Robinson', 'student'),

('Jessica', 'A.', 'Miller', 'student'),

('Brian', NULL, 'Thompson', 'student'),

('Sandra', 'N.', 'Davis', 'student'),

('Eric', 'P.', 'Smith', 'student'),

('Rachel', NULL, 'Evans', 'student'),

('George', 'W.', 'Parker', 'student'),

('Lisa', 'K.', 'Hernandez', 'student');


-- Class Table

INSERT INTO Class (CourseCode, Name, Department) VALUES

('CPSC-101', 'Introduction to Programming', 'Computer Science'),

('CPSC-111', 'Data Structures and Algorithms', 'Computer Science'),

('MATH-201', 'Calculus I', 'Mathematics'),

('PHYS-301', 'Physics for Engineers', 'Physics'),
('PYS-101', 'Introduction to Psychology', 'Psychology'),
('ENG-541', 'English Composition', 'English'),
('ART-271', 'Art History', 'Art'),
('CHEM-101', 'Introduction to Chemistry', 'Chemistry'),
('HIST-281', 'World History', 'History'),
('ECON-554', 'Microeconomics', 'Economy'),
('BIOL-211', 'Cell Biology', 'Biology'),
('CHEM-301', 'Organic Chemistry', 'Chemistry'),
('MATH-202', 'Calculus II', 'Mathematics'),
('PHYS-201', 'Classical Mechanics', 'Physics'),
('SOC-101', 'Introduction to Sociology', 'Sociology'),
('BUS-401', 'Marketing Management', 'Business'),
('ENG-321', 'Creative Writing', 'English'),
('PHIL-101', 'Introduction to Philosophy', 'Philosophy'),
('ART-352', 'Modern Art', 'Art'),
('HIST-381', 'European History', 'History'),
('ECON-301', 'Macroeconomics', 'Economy'),
('PSYCH-201', 'Abnormal Psychology', 'Psychology'),
('SOC-201', 'Social Problems', 'Sociology'),
('CHEM-202', 'Analytical Chemistry', 'Chemistry'),
('MATH-301', 'Linear Algebra', 'Mathematics'),
('PHYS-401', 'Quantum Mechanics', 'Physics'),
('BUS-201', 'Financial Accounting', 'Business'),
('ENG-431', 'American Literature', 'English'),
('PHIL-202', 'Ethics', 'Philosophy'),
('ART-413', 'Renaissance Art', 'Art');


-- Section Table
INSERT INTO Section (sectionNumber, CourseCode, InstructorID,
CurrentEnrollment, MaxEnrollment, Waitlist, SectionStatus) VALUES
(1, 'CPSC-101', 2, 5, 30, 1, 'open'),
(2, 'CPSC-101', 2, 4, 30, 1, 'open'),

(1, 'CPSC-111', 2, 2, 35, 1, 'open'),

(2, 'CPSC-111', 7, 3, 25, 0, 'open'),

(5, 'MATH-201', 3, 0, 25, 0, 'open'),

(1, 'MATH-201', 8, 3, 30, 0, 'open'),

(2, 'MATH-201', 9, 2, 20, 1, 'open'),

(1, 'PHYS-301', 3, 4, 20, 0, 'open'),

(2, 'PHYS-301', 10, 4, 35, 0, 'open'),

(1, 'PYS-101', 2, 6, 35, 0, 'open'),

(2, 'PYS-101', 6, 4, 25, 0, 'open'),

(1, 'ENG-541', 7, 2, 25, 1, 'open'),

(2, 'ENG-541', 8, 1, 30, 0, 'open'),

(1, 'ART-271', 9, 1, 20, 0, 'open'),

(2, 'ART-271', 10, 0, 35, 0, 'open'),

(1, 'CHEM-101', 4, 1, 30, 0, 'open'),

(2, 'CHEM-101', 5, 0, 25, 1, 'open'),

(1, 'HIST-281', 6, 0, 35, 0, 'open'),

(2, 'HIST-281', 7, 0, 20, 0, 'open'),

(1, 'ECON-554', 8, 2, 30, 0, 'open'),

(2, 'ECON-554', 9, 0, 25, 0, 'open'),

(1, 'BIOL-211', 10, 2, 20, 0, 'open'),

(2, 'BIOL-211', 4, 0, 35, 0, 'open'),

(1, 'CHEM-301', 5, 0, 30, 0, 'open'),

(2, 'CHEM-301', 6, 0, 25, 0, 'open'),

(1, 'MATH-202', 7, 3, 35, 0, 'open'),

(2, 'MATH-202', 8, 2, 20, 0, 'open'),

(1, 'PHYS-201', 9, 2, 30, 0, 'open'),

(2, 'PHYS-201', 10, 0, 25, 0, 'open'),

(1, 'SOC-101', 4, 2, 25, 0, 'open'),

(2, 'SOC-101', 5, 2, 30, 0, 'open'),

(1, 'BUS-401', 6, 0, 20, 0, 'open'),
(2, 'BUS-401', 7, 0, 35, 0, 'open'),

(1, 'ENG-321', 8, 0, 30, 0, 'open'),

(2, 'ENG-321', 9, 0, 25, 0, 'open'),

(1, 'PHIL-101', 10, 0, 35, 0, 'open'),

(2, 'PHIL-101', 4, 0, 20, 0, 'open'),

(1, 'ART-352', 5, 0, 30, 0, 'open'),

(2, 'ART-352', 6, 0, 25, 0, 'open'),

(1, 'HIST-381', 7, 0, 25, 0, 'open'),

(2, 'HIST-381', 8, 0, 30, 0, 'open'),

(1, 'ECON-301', 9, 0, 20, 0, 'open'),

(2, 'ECON-301', 10, 0, 35, 0, 'open'),

(1, 'PSYCH-201', 4, 0, 30, 0, 'open'),

(2, 'PSYCH-201', 5, 0, 25, 0, 'open'),

(1, 'SOC-201', 6, 0, 35, 0, 'open'),

(2, 'SOC-201', 7, 0, 20, 0, 'open'),

(1, 'CHEM-202', 8, 0, 30, 0, 'open'),

(2, 'CHEM-202', 9, 0, 25, 0, 'open'),

(1, 'MATH-301', 10, 1, 20, 0, 'open'),

(2, 'MATH-301', 4, 2, 35, 0, 'open'),

(1, 'PHYS-401', 5, 2, 30, 0, 'open'),

(2, 'PHYS-401', 6, 2, 25, 0, 'open'),

(1, 'BUS-201', 7, 0, 35, 0, 'open'),

(2, 'BUS-201', 8, 0, 20, 0, 'open'),

(1, 'ENG-431', 9, 1, 30, 0, 'open'),

(2, 'ENG-431', 10, 0, 25, 0, 'open'),

(1, 'PHIL-202', 4, 2, 25, 0, 'open'),

(2, 'PHIL-202', 5, 1, 30, 0, 'open'),

(1, 'ART-413', 6, 0, 20, 1, 'open');


-- RegistrationList Table
```

```sql
INSERT INTO RegistrationList (StudentID, CourseCode, SectionNumber,
Status) VALUES
-- Student 12
(12, 'CPSC-101', 1, 'enrolled'),
(12, 'MATH-201', 1, 'enrolled'),
(12, 'PHYS-301', 1, 'enrolled'),
(12, 'PYS-101', 1, 'enrolled'),
(12, 'CPSC-111', 1, 'dropped'),
-- Student 13
(13, 'CPSC-101', 2, 'enrolled'),
(13, 'MATH-201', 2, 'enrolled'),
(13, 'PHYS-301', 2, 'enrolled'),
(13, 'CPSC-111', 1, 'waitlisted'),
-- Student 14
(14, 'CPSC-101', 1, 'enrolled'),
(14, 'MATH-201', 1, 'enrolled'),
(14, 'PHYS-301', 1, 'enrolled'),
(14, 'CPSC-111', 2, 'dropped'),
-- Student 15
(15, 'CPSC-111', 1, 'enrolled'),
(15, 'PHYS-301', 1, 'enrolled'),
(15, 'PYS-101', 1, 'enrolled'),
(15, 'CPSC-101', 2, 'waitlisted'),
-- Student 16
(16, 'CPSC-111', 2, 'enrolled'),
(16, 'MATH-201', 2, 'enrolled'),
(16, 'PHYS-301', 2, 'enrolled'),
(16, 'PYS-101', 2, 'enrolled'),
(16, 'CPSC-101', 1, 'waitlisted'),
-- Student 17
(17, 'MATH-201', 1, 'enrolled'),
(17, 'PYS-101', 1, 'enrolled'),
(17, 'CPSC-101', 2, 'dropped'),
-- Student 18
(18, 'CPSC-101', 1, 'enrolled'),
(18, 'PHYS-301', 1, 'enrolled'),
(18, 'PYS-101', 1, 'enrolled'),
(18, 'MATH-201', 2, 'waitlisted'),

-- Student 19
(19, 'CPSC-101', 1, 'enrolled'),
(19, 'MATH-202', 1, 'enrolled'),
(19, 'PHYS-201', 1, 'enrolled'),
(19, 'PYS-101', 1, 'enrolled'),
(19, 'CHEM-101', 2, 'waitlisted'),
-- Student 20
(20, 'CPSC-101', 2, 'enrolled'),
(20, 'MATH-301', 2, 'enrolled'),
(20, 'PHYS-401', 2, 'enrolled'),
(20, 'PYS-101', 2, 'enrolled'),
(20, 'ENG-541', 1, 'waitlisted'),
-- Student 21
(21, 'MATH-202', 1, 'enrolled'),
(21, 'PHYS-201', 1, 'enrolled'),
(21, 'CHEM-101', 1, 'enrolled'),
(21, 'ENG-541', 1, 'enrolled'),
-- Student 22
(22, 'CPSC-101', 2, 'enrolled'),
(22, 'PHIL-101', 2, 'enrolled'),
(22, 'SOC-101', 2, 'enrolled'),
(22, 'MATH-301', 2, 'enrolled'),
(22, 'PHYS-401', 2, 'enrolled'),
-- Student 23
(23, 'CPSC-111', 1, 'enrolled'),
(23, 'ECON-554', 1, 'enrolled'),
(23, 'BIOL-211', 1, 'enrolled'),
(23, 'ENG-431', 1, 'enrolled'),
-- Student 24
(24, 'CPSC-111', 2, 'enrolled'),
(24, 'PHYS-301', 2, 'enrolled'),
(24, 'PYS-101', 2, 'enrolled'),
(24, 'MATH-202', 2, 'enrolled'),
-- Student 25
(25, 'MATH-301', 1, 'enrolled'),
(25, 'PHYS-401', 1, 'enrolled'),
(25, 'SOC-101', 1, 'enrolled'),
(25, 'PHIL-202', 1, 'enrolled'),
(25, 'ART-413', 1, 'waitlisted'),
-- Student 26
(26, 'CPSC-101', 1, 'dropped'),
(26, 'PHYS-301', 1, 'dropped'),
(26, 'PYS-101', 1, 'dropped'),
(26, 'MATH-202', 2, 'enrolled'),
-- Student 27
(27, 'MATH-301', 2, 'dropped'),
(27, 'PHYS-401', 2, 'dropped'),
(27, 'SOC-101', 2, 'enrolled'),
(27, 'PHIL-202', 2, 'enrolled'),
-- Student 28
(28, 'CPSC-111', 1, 'dropped'),
(28, 'ECON-554', 1, 'enrolled'),
(28, 'BIOL-211', 1, 'enrolled'),
(28, 'ENG-431', 1, 'dropped'),
-- Student 29

(29, 'CPSC-111', 2, 'enrolled'),

(29, 'PHYS-301', 2, 'dropped'),

(29, 'PYS-101', 2, 'enrolled'),

(29, 'MATH-202', 2, 'dropped'),

-- Student 30

(30, 'MATH-301', 1, 'dropped'),

(30, 'PHYS-401', 1, 'enrolled'),

(30, 'SOC-101', 1, 'enrolled'),

(30, 'PHIL-202', 1, 'enrolled'),

-- Student 31

(31, 'CPSC-101', 2, 'enrolled'),

(31, 'PHYS-301', 2, 'enrolled'),

(31, 'PYS-101', 2, 'dropped'),

(31, 'ENG-541', 2, 'enrolled'),

-- Student 32

(32, 'CPSC-101', 1, 'enrolled'),

(32, 'MATH-202', 1, 'enrolled'),

(32, 'PHYS-201', 1, 'dropped'),

(32, 'PYS-101', 1, 'enrolled'),

-- Student 33

(33, 'CHEM-101', 2, 'dropped'),

(33, 'ENG-541', 1, 'enrolled'),

(33, 'ART-271', 1, 'enrolled'),

(33, 'CHEM-301', 1, 'dropped');


COMMIT;
```

Comments:

- ChatGPT was used to generate some of the data to populate the tables

Task 2: FastAPI Implementation and Database Query Functions

What is to be done:

- Implement a FastAPI application to serve endpoints for managing class registrations.
- Define endpoints for students, registrar, waitlist, and instructors.
- Implement CRUD operations for adding classes, enrolling students, dropping courses, waitlist, etc..

Tools Used:

- FastAPI
- SQLite

Results:

- Main FastAPI application implemented in ‘`_main_.py`’.
- Endpoints defined for students, registrar, waitlist, and instructors.

```
########## STUDENTS ENDPOINTS ######################

@app.get(path="/classes", operation_id="available_classes",

response_model = AvailableClassResponse)

async def available_classes(department_name: str):

"""API to fetch list of available classes for a given department name.


Args:

department_name (str): Department name


Returns:

AvailableClassResponse: AvailableClassResponse model

"""

result = get_available_classes(db_connection=db_connection,

department_name=department_name)

logger.info('Succesffuly exexuted available')

return AvailableClassResponse(available_classes = result)


@app.post(path ="/enrollment", operation_id="course_enrollment",

response_model= EnrollmentResponse)

async def course_enrollment(enrollment_request: EnrollmentRequest):
```

```
"""Allow enrollment of a course under given section for a student


Args:
enrollment_request (EnrollmentRequest): EnrollmentRequest model


Raises:
HTTPException: Raise HTTP exception when role is not authrorized
HTTPException: Raise HTTP exception when query fail to execute in
database


Returns:
EnrollmentResponse: EnrollmentResponse model
"""


role = check_user_role(db_connection, enrollment_request.student_id)
if role == UserRole.NOT_FOUND or role != UserRole.STUDENT:
raise HTTPException(status_code = status.HTTP_401_UNAUTHORIZED, detail=
f'Enrollment not authorized for role:{role}')
check_if_already_enrolled = check_status_query(db_connection,
enrollment_request)
if check_if_already_enrolled :
return check_if_already_enrolled
eligibility_status = check_enrollment_eligibility(db_connection,
enrollment_request.section_number, enrollment_request.course_code)
if eligibility_status == RegistrationStatus.NOT_ELIGIBLE:
return EnrollmentResponse(enrollment_status = 'not eligible')


try:
registration = Registration(student_id = enrollment_request.student_id,
enrollment_status = eligibility_status,
section_number = enrollment_request.section_number, course_code =
enrollment_request.course_code)
```

```
insert_status = complete_registration(db_connection,registration)

if insert_status == QueryStatus.SUCCESS:

return EnrollmentResponse(enrollment_date = datetime.utcnow(),
enrollment_status = eligibility_status)


except DBException as err:

raise HTTPException(status_code=status.HTTP_500_INTERNAL_SERVER_ERROR,
detail= err.error_detail)



@app.put(path = "/dropcourse", operation_id=
"update_registration_status",response_model= DropCourseResponse)

async def
update_registration_status(enrollment_request:EnrollmentRequest):

"""API for students to drop a course


Args:

enrollment_request (EnrollmentRequest): Enrollment request


Raises:

HTTPException: Raise Exception if database fail to execute query


Returns:

DropCourseResponse : drop course response

"""

try:

registration = Registration(section_number=
enrollment_request.section_number,

student_id=enrollment_request.student_id,

course_code=enrollment_request.course_code,

enrollment_status='enrolled')

result = update_student_registration_status(db_connection,registration)
```

```
if result == RegistrationStatus.DROPPED:
return DropCourseResponse(course_code=enrollment_request.course_code,
section_number=enrollment_request.section_number,
status='already dropped')
return DropCourseResponse(course_code=enrollment_request.course_code,
section_number=enrollment_request.section_number,
status='drop successfull')
except DBException as err:
raise
HTTPException(status_code=status.HTTP_500_INTERNAL_SERVER_ERROR,detail=er
r.error_detail)


########## REGISTRAR ENDPOINTS ######################
@app.post(path="/classes", operation_id="add_class",
response_model=AddClassResponse)
async def add_class(addClass_request: AddClassRequest):
classExists = check_class_exists(db_connection,
addClass_request.course_code)
if classExists:
try:
response = addSection(db_connection, addClass_request.section_number,
addClass_request.course_code, addClass_request.instructor_id,
addClass_request.max_enrollment)
if response == QueryStatus.SUCCESS:
return AddClassResponse(addClass_status = 'Successfully added new
section')
else:
return AddClassResponse(addClass_status = 'Failed to add Section')
except DBException as err:
raise HTTPException(status_code=status.HTTP_500_INTERNAL_SERVER_ERROR,
detail= err.error_detail)
else:
try:
```

```
addClassResponse = addClass(db_connection, addClass_request.course_code,
addClass_request.class_name, addClass_request.department)
if addClassResponse == QueryStatus.SUCCESS:
addSectionResponse = addSection(db_connection,
addClass_request.section_number, addClass_request.course_code,
addClass_request.instructor_id, addClass_request.max_enrollment)
if addSectionResponse == QueryStatus.SUCCESS:
return AddClassResponse(addClass_status = 'Successfully added Class &
Section')
else:
return AddClassResponse(addClass_status = 'Failed to add Class &
Section')
except DBException as err:
raise HTTPException(status_code=status.HTTP_500_INTERNAL_SERVER_ERROR,
detail= err.error_detail)


@app.delete(path="/sections", operation_id="delete_section",
response_model=DeleteSectionResponse)
async def delete_section(deleteSection_Request: DeleteSectionRequest):
sectionExists = check_section_exists(db_connection,
deleteSection_Request.course_code, deleteSection_Request.section_number)
if not sectionExists:
raise HTTPException(status_code = status.HTTP_401_UNAUTHORIZED, detail=
f'This section does not exist')
response = deleteSection(db_connection,
deleteSection_Request.course_code, deleteSection_Request.section_number)
if response == QueryStatus.SUCCESS:
return DeleteSectionResponse(deleteSection_status = 'Successfully deleted
section ' + str(deleteSection_Request.section_number) + ' of course ' +
deleteSection_Request.course_code)
else:
return DeleteSectionResponse(deleteSection_status = 'Failed to delete
section')
```

```
@app.post(path="/changeSectionInstructor",
operation_id="change_section_instructor",
response_model=ChangeInstructorResponse)
async def change_section_instructor(changeInstructor_Request:
ChangeInstructorRequest):
sectionExists = check_section_exists(db_connection,
changeInstructor_Request.course_code,
changeInstructor_Request.section_number)
if sectionExists == 0:
raise HTTPException(status_code = status.HTTP_401_UNAUTHORIZED, detail=
f'This section does not exist')
response = changeSectionInstructor(db_connection,
changeInstructor_Request.course_code,
changeInstructor_Request.section_number,
changeInstructor_Request.instructor_id)
if response == QueryStatus.SUCCESS:
return ChangeInstructorResponse(changeInstructor_status = 'Successfully
changed instructor of section ' +
str(changeInstructor_Request.section_number))
else:
return ChangeInstructorResponse(changeInstructor_status = 'Failed to
change instructor')
@app.post(path="/freezeEnrollment", operation_id='freeze_enrollment',
response_model=FreezeEnrollmentResponse)
async def freeze_enrollment(freezeEnrollment_Request:
FreezeEnrollmentRequest):
sectionExists = check_section_exists(db_connection,
freezeEnrollment_Request.course_code,
freezeEnrollment_Request.section_number)
if sectionExists == 0:
raise HTTPException(status_code = status.HTTP_401_UNAUTHORIZED, detail=
f'This section does not exist')
response = freezeEnrollment(db_connection,
freezeEnrollment_Request.course_code,
freezeEnrollment_Request.section_number)
if response == QueryStatus.SUCCESS:
```

```
return FreezeEnrollmentResponse(freezeEnrollment_status = 'Successfully
freezed enrollment for section ' +
str(freezeEnrollment_Request.section_number))
else:
return FreezeEnrollmentResponse(freezeEnrollment_status = 'Failed to
freeze enrollment')

########## REGISTRAR ENDPOINTS ENDS ######################


########## WAITLIST ENDPOINTS ######################
# student viewing their position in the waitlist
@app.get(path="/waitlist_position", operation_id="waitlist_position",
response_model = WaitlistPositionRes)
async def waitlist_position(waitlist_request: WaitlistPositionReq):
"""API to fetch the current position of a student in a waitlist.
Args:
student_id: int
Returns:
WaitlistPositionRes: WaitlistPositionRes model
"""
result = get_waitlist_status(db_connection=db_connection,
student_id=waitlist_request.student_id)
logger.info('Succesffuly executed the query')
return WaitlistPositionRes(waitlist_positions = result)

# instructors viewing the current waitlist for a course and section
@app.get(path="/view_waitlist", operation_id="view_waitlist",
response_model = ViewWaitlistRes)
async def view_waitlist(view_waitlist_req: ViewWaitlistReq):
"""API to fetch the students in a waitlist.
Args:
section_number: int
```

```
course_code: str

Returns:

ViewWaitlistRes: ViewWaitlistRes model

"""

result = get_waitlist(db_connection=db_connection,

course_code=view_waitlist_req.course_code,

section_number=view_waitlist_req.section_number)

logger.info('Succesffuly executed the query')

return ViewWaitlistRes(waitlisted_students = result)


########## WAITLIST ENDPOINTS ENDS ######################




########## INSTRUCTOR ENDPOINTS ######################

@app.get(path="/classEnrollment", operation_id="list_enrollment",
response_model=RecordsEnrollmentResponse)

async def list_enrollment(instructor_id: int, section_number:
Optional[int] = None, course_code: Optional[str] = None):

"""API to fetch list of enrolled students for a given instructor.


Args:

instructor_id (int): Instructor id

section_number (Optional[int]): Section number (optional)

course_code (Optional[str]): Course code (optional)


Returns:

RecordsEnrollmentResponse: RecordsEnrollmentResponse model

"""

role = check_is_instructor(db_connection, instructor_id)

if role == UserRole.NOT_FOUND or role != UserRole.INSTRUCTOR:

logger.info('List Class Enrollment not authorized for role')
```

```
raise HTTPException(status_code = status.HTTP_401_UNAUTHORIZED, detail=
f'List Class Enrollment not authorized for role: {role}')
result = get_enrolled_students(db_connection, instructor_id, course_code,
section_number)
logger.info('Successfully executed list_enrollment')
return RecordsEnrollmentResponse(enrolled_students = result)


# TODO: test this endpoint
@app.get(path="/classWaitlist", operation_id="list_waitlist",
response_model=RecordsWaitlistResponse)
async def list_waitlist(instructor_id: int, section_number: Optional[int]
= None, course_code: Optional[str] = None):
"""API to fetch list of enrolled students for a given instructor.


Args:
instructor_id (int): Instructor id
section_number (Optional[int]): Section number (optional)
course_code (Optional[str]): Course code (optional)


Returns:
RecordsWaitlistResponse: RecordsWaitlistResponse model
"""
role = check_is_instructor(db_connection, instructor_id)
if role == UserRole.NOT_FOUND or role != UserRole.INSTRUCTOR:
logger.info('List Class Waitlist not authorized for role')
raise HTTPException(status_code = status.HTTP_401_UNAUTHORIZED, detail=
f'List Class Waitlist not authorized for role: {role}')
result = get_waitlisted_students(db_connection, instructor_id,
course_code, section_number)
logger.info('Successfully executed list_waitlist')
return RecordsWaitlistResponse(waitlisted_students = result)
```

```
@app.get(path="/classDropped", operation_id="list_dropped",
response_model=RecordsDroppedResponse)
async def list_dropped(instructor_id: int, section_number: Optional[int]
= None, course_code: Optional[str] = None):
"""API to fetch list of dropped students for a given section.

Args:
instructor_id (int): Instructor id
section_number (Optional[int]): Section number (optional)
course_code (Optional[str]): Course code (optional)

Returns:
RecordsDroppedResponse: RecordsDroppedResponse model
"""
role = check_is_instructor(db_connection, instructor_id)
if role == UserRole.NOT_FOUND or role != UserRole.INSTRUCTOR:
logger.info('List Class Dropped not authorized for role')
raise HTTPException(status_code = status.HTTP_401_UNAUTHORIZED, detail=
f'List Class Dropped not authorized for role: {role}')
result = get_dropped_students(db_connection, instructor_id, course_code,
section_number)
logger.info('Successfully executed list_dropped')
return RecordsDroppedResponse(dropped_students = result)

@app.post(path="/dropStudent", operation_id="instructor_drop_student",
response_model=DroppedResponse)
async def instructor_drop_student(DropRequest: DropStudentRequest):
"""API to drop a student from a section.

Args:
instructor_id (int): Instructor id
student_id (int): Student id
```

```
section_number (int): Section number
course_code (str): Course code

Returns:
droppedResponse: droppedResponse model
"""
role = check_is_instructor(db_connection, DropRequest.instructor_id)
# # check if action is being perform by instructor
if role == UserRole.NOT_FOUND or role != UserRole.INSTRUCTOR:
logger.info('Drop Student not authorized for role')
raise HTTPException(status_code = status.HTTP_401_UNAUTHORIZED, detail=
f'Drop Student not authorized for role: {role}')
# # check if instructor teaches the section
check_instructor = check_is_instructor_of_section(db_connection,
DropRequest)
if check_instructor == False:
logger.info('Instructor does not teach the section')
raise HTTPException(status_code = status.HTTP_401_UNAUTHORIZED, detail=
f'Instructor does not teach the section')
# # check if student is enrolled in the section or waitlisted
check_status = check_is_enrolled(db_connection, DropRequest)
if check_status == False:
logger.info('Student is not enrolled in the section')
raise HTTPException(status_code = status.HTTP_401_UNAUTHORIZED, detail=
f'Student is not enrolled in the section')
try:
result = drop_student(db_connection, DropRequest)
logger.info('Successfully executed drop_student')
if result == QueryStatus.SUCCESS:
return DroppedResponse(drop_status = "Student was dropped")
except DBException as err:
```

```
raise
HTTPException(status_code=status.HTTP_500_INTERNAL_SERVER_ERROR,detail=er
r.error_detail)
########## INSTRUCTOR ENDPOINTS ENDS #####################
```

- CRUD operations implemented for adding classes, enrolling students, dropping courses, waitlist, etc in ‘`database_query.py`’.

```
def get_available_classes(db_connection: Connection, department_name:
str) -> List[AvailableClass]:
"""Query database to get available classes for a given department name


Args:
db_connection (Connection): SQLite Connection
department_name (str): Department name


Returns:
List[AvailableClass]: List of available classes
"""
result = []
query = LIST_AVAILABLE_SQL_QUERY.format(department_name=department_name)
cursor = db_connection.cursor()
rows = cursor.execute(query)
if rows.arraysize == 0:
raise HTTPException(status_code= status.HTTP_400_BAD_REQUEST, detail=
f'Record not found for given department_name:{department_name}')
for row in rows:
available_class = AvailableClass(course_name=row[0],
course_code=row[1],
department=row[2],
current_enrollment=row[3],
waitlist=row[4],
max_enrollment=row[5],

section_number=row[6],

instructor_first_name=row[7],

instructor_last_name=row[8])

result.append(available_class)

cursor.close()

return result


def check_user_role(db_connection: Connection, student_id: int)->
Union[str, None]:

logger.info('Checking user role')

query = f"""

SELECT role FROM Users where CWID = {student_id}

"""

cursor = db_connection.cursor()

rows = cursor.execute(query)

if rows.arraysize == 0:

raise HTTPException(status_code= status.HTTP_400_BAD_REQUEST, detail=
f'Record not found for given student_id:{student_id}')

result = UserRole.NOT_FOUND

for row in rows:

result = row[0]

return result



def count_waitlist_registration(db_connection: Connection, section_id:
int)->int:

logger.info('Checking waitlist registration')

query = f"""SELECT COUNT(*) FROM RegistrationList WHERE ClassID =
{section_id} and Status = 'waitlisted'

"""

cursor = db_connection.cursor()
```

```
rows = cursor.execute(query)

if rows.arraysize == 0:

raise HTTPException(status_code= status.HTTP_400_BAD_REQUEST, detail=
f'Record not found for given section_id:{section_id}')

result = 0

for row in rows:

result = row[0]

return result


def check_enrollment_eligibility(db_connection: Connection,
section_number: int, course_code: str)->str:

logger.info('Checking enrollment eligibility')

query = f"""SELECT CurrentEnrollment as 'current_enrollment',
MaxEnrollment as 'max_enrollment', Waitlist as 'waitlist' FROM "Section"
WHERE CourseCode = '{course_code}' and SectionNumber = {section_number}
"""


cursor = db_connection.cursor()

rows = cursor.execute(query)

if rows.arraysize == 0:

raise HTTPException(status_code= status.HTTP_400_BAD_REQUEST, detail=
f'Record not found for given section_number:{section_number} and
course_code:{course_code}')

query_result = {}

for row in rows:

query_result['current_enrollment'] = row[0]

query_result['max_enrollment'] = row[1]

query_result['waitlist'] = row[2]

# First check whether there is capacity to enroll in a section

if query_result['max_enrollment'] - query_result['current_enrollment'] >=
1:

return RegistrationStatus.ENROLLED

if query_result['waitlist'] <= WAITLIST_ALLOWED:
```

```
return RegistrationStatus.WAITLISTED

return RegistrationStatus.NOT_ELIGIBLE


def check_status_query(db_connection: Connection, enrollment_request:
EnrollmentRequest) -> Union[EnrollmentResponse, None]:

check_status_query = f""" SELECT Status, EnrollmentDate FROM
RegistrationList where StudentID = {enrollment_request.student_id} and
SectionNumber = {enrollment_request.section_number} and CourseCode =
'{enrollment_request.course_code}'"""

cursor = db_connection.cursor()

try:

rows = cursor.execute(check_status_query)

if rows.arraysize == 0:

raise HTTPException(status_code= status.HTTP_400_BAD_REQUEST, detail=
f'Record not found')

row = rows.fetchone()

if row[0] == RegistrationStatus.ENROLLED:

return EnrollmentResponse(enrollment_status="already enrolled",
enrollment_date=row[1])

except Exception as err:

logger.error(err)

raise DBException(error_detail = 'Fail to register')

return None



def complete_registration(db_connection: Connection, registration:
Registration) -> str:

logger.info('Starting registration')

insert_query = f"""

INSERT INTO RegistrationList (StudentID,CourseCode, SectionNumber,
Status) VALUES ({registration.student_id},'{registration.course_code}',
{registration.section_number}, '{registration.enrollment_status}')

"""
```

```
update_current_enrollment_query = f"""
UPDATE "Section" SET CurrentEnrollment = CurrentEnrollment + 1 WHERE
SectionNumber = {registration.section_number} and CourseCode =
'{registration.course_code}'
"""
update_waitlist_query = f"""
UPDATE "Section" SET Waitlist = Waitlist + 1 WHERE SectionNumber =
{registration.section_number} and CourseCode =
'{registration.course_code}'
"""


cursor = db_connection.cursor()
cursor.execute("BEGIN")
try:
cursor.execute(insert_query)
if registration.enrollment_status == RegistrationStatus.ENROLLED:
cursor.execute(update_current_enrollment_query)
elif registration.enrollment_status == RegistrationStatus.WAITLISTED:
cursor.execute(update_waitlist_query)
cursor.execute("COMMIT")
except Exception as err:
logger.error(err)
cursor.execute("ROLLBACK")
logger.info('Rolling back transaction')
raise DBException(error_detail = 'Fail to register')
finally:
cursor.close()


return QueryStatus.SUCCESS


def update_student_registration_status(db_connection:Connection,
registration: Registration)-> str:
```

```
logger.info('Upadting the registration status')

check_status_query = f""" SELECT Status FROM RegistrationList where
StudentID = {registration.student_id} and SectionNumber =
{registration.section_number} and CourseCode =
'{registration.course_code}'"""

update_status_query = f""" UPDATE RegistrationList SET Status = 'dropped'
where StudentID = {registration.student_id} and
SectionNumber = {registration.section_number} and CourseCode =
'{registration.course_code}' and status = 'enrolled' """

update_current_enrollment_query = f"""UPDATE SECTION set
CurrentEnrollment = CurrentEnrollment -1 where SectionNumber =
{registration.section_number} and CourseCode =
'{registration.course_code}'"""

update_waitlist_count_query = f"""UPDATE "Section" SET Waitlist =
Waitlist - 1 WHERE SectionNumber = {registration.section_number} and
CourseCode = '{registration.course_code}'"""

cursor = db_connection.cursor()

cursor.execute("BEGIN")

try:

rows = cursor.execute(check_status_query)

if rows.arraysize == 0:

raise HTTPException(status_code= status.HTTP_400_BAD_REQUEST, detail=
f'Record not found')

row = rows.fetchone()

if row[0] == RegistrationStatus.DROPPED:

return RegistrationStatus.DROPPED

else:

cursor.execute(update_status_query)

if row[0] == RegistrationStatus.ENROLLED:

cursor.execute(update_current_enrollment_query)

elif row[0] == RegistrationStatus.WAITLISTED:

cursor.execute(update_waitlist_count_query)

cursor.execute("COMMIT")

except Exception as err:
```

```
logger.error(err)

cursor.execute("ROLLBACK")

logger.info('Rolling back transaction')

raise DBException(error_detail = 'Fail to drop the class')

finally:

cursor.close()

return QueryStatus.SUCCESS

def check_class_exists(db_connection: Connection, course_code: str)->
bool:

logger.info('Checking if class exists')

result = False

query = f"""

SELECT CourseCode FROM Class where CourseCode = '{course_code}'

"""

cursor = db_connection.cursor()

cursor.execute(query)

rows = cursor.fetchall()

if len(rows) > 0:

result = True

return result


def check_section_exists(db_connection: Connection, course_code: str,
section_number: int)-> bool:

logger.info('Checking if section exists')

result = False

query = f"""

SELECT SectionNumber FROM Section where CourseCode = '{course_code}' and
SectionNumber = {section_number}

"""

cursor = db_connection.cursor()

cursor.execute(query)

rows = cursor.fetchall()
```

```
if len(rows) > 0:

result = True

return result


def addClass(db_connection: Connection, course_code, class_name,
department) -> str:

logger.info('Starting to add class')

insert_query = f"""

INSERT INTO Class (CourseCode, Name, Department) VALUES ('{course_code}',
'{class_name}', '{department}')

"""


cursor = db_connection.cursor()


cursor.execute("BEGIN")

try:

cursor.execute(insert_query)

cursor.execute("COMMIT")

except Exception as err:

logger.error(err)

cursor.execute("ROLLBACK")

logger.info('Rolling back transaction')

raise DBException(error_detail = 'Fail to add class')

finally:

cursor.close()


return QueryStatus.SUCCESS


def addSection(db_connection: Connection, section_number, course_code,
instructor_id, max_enrollment) -> str:

logger.info('Starting to add section')

insert_query = f"""
```

```
INSERT INTO Section (SectionNumber, CourseCode, InstructorID,
MaxEnrollment, CurrentEnrollment, Waitlist, SectionStatus) VALUES
({section_number}, '{course_code}', {instructor_id}, {max_enrollment}, 0,
0, 'open')
"""
cursor = db_connection.cursor()
cursor.execute("BEGIN")
try:
cursor.execute(insert_query)
cursor.execute("COMMIT")
except Exception as err:
logger.error(err)
cursor.execute("ROLLBACK")
logger.info('Rolling back transaction')
raise DBException(error_detail = 'Fail to add section')
finally:
cursor.close()

return QueryStatus.SUCCESS

def deleteSection(db_connection: Connection, course_code: str,
section_number: int) -> str:
logger.info('Starting to delete section')
delete_query = f"""
DELETE FROM Section WHERE CourseCode = '{course_code}' and SectionNumber
= {section_number}
"""

cursor = db_connection.cursor()
cursor.execute("BEGIN")
try:
cursor.execute(delete_query)
cursor.execute("COMMIT")

except Exception as err:

logger.error(err)

cursor.execute("ROLLBACK")

logger.info('Rolling back transaction')

raise DBException(error_detail = 'Fail to delete section')

finally:

cursor.close()


return QueryStatus.SUCCESS


def changeSectionInstructor(db_connection: Connection, course_code: str,
section_number: int, instructor_id: int) -> str:

logger.info('Starting to change instructor for section ',
str(section_number))

update_query = f"""

UPDATE Section SET InstructorID = {instructor_id} WHERE SectionNumber =
{section_number} and CourseCode = '{course_code}'

"""


cursor = db_connection.cursor()

cursor.execute("BEGIN")

try:

cursor.execute(update_query)

cursor.execute("COMMIT")

except Exception as err:

logger.error(err)

cursor.execute("ROLLBACK")

logger.info('Rolling back transaction')

raise DBException(error_detail = 'Fail to change instructor')

finally:

cursor.close()
```

```
return QueryStatus.SUCCESS


def freezeEnrollment(db_connection: Connection, course_code: str,
section_number: int) -> str:
logger.info('Starting to freeze enrollment for section ',
str(section_number))
update_query = f"""
UPDATE Section SET SectionStatus = 'closed' WHERE SectionNumber =
{section_number} and CourseCode = '{course_code}'
"""


cursor = db_connection.cursor()
cursor.execute("BEGIN")
try:
cursor.execute(update_query)
cursor.execute("COMMIT")
except Exception as err:
logger.error(err)
cursor.execute("ROLLBACK")
logger.info('Rolling back transaction')
raise DBException(error_detail = 'Fail to freeze enrollment')
finally:
cursor.close()


return QueryStatus.SUCCESS

def get_enrolled_students(db_connection: Connection, instructor_id: int,
course_code: Optional[str] = None, section_number: Optional[int] = None)
-> List[EnrollmentListResponse]:
logger.info('Getting enrolled students for instructor with CWID:')
query = """
SELECT
```

```
Users.CWID AS StudentCWID,

Users.Name AS StudentFirstName,

Users.LastName AS StudentLastName,

Class.CourseCode AS CourseCode,

Section.SectionNumber AS SectionNumber,

Class.Name AS ClassName,

RegistrationList.Status AS Status

FROM

RegistrationList

JOIN Users ON RegistrationList.StudentID = Users.CWID

JOIN Section ON RegistrationList.CourseCode = Section.CourseCode AND
RegistrationList.SectionNumber = Section.SectionNumber

JOIN Class ON Section.CourseCode = Class.CourseCode

WHERE

Section.InstructorID = ?

AND RegistrationList.Status = 'enrolled'

"""

params = [instructor_id]

if course_code is not None:

query += " AND Section.CourseCode = ?"

params.append(course_code)

if section_number is not None:

query += " AND Section.SectionNumber = ?"

params.append(section_number)

else:

query += " ORDER BY Class.CourseCode, Section.SectionNumber,
Users.LastName, Users.Name"

cur = db_connection.execute(query, tuple(params))

enrollment = cur.fetchall()

if not enrollment:

raise HTTPException(
status_code=status.HTTP_404_NOT_FOUND, detail="Enrollment for instructor
not found"
)
results = [{"student_cwid": row[0],
"student_first_name": row[1],
"student_last_name": row[2],
"course_code": row[3],
"section_number": row[4],
"class_name": row[5],
"status": row[6]} for row in enrollment]
return results


# dropped students
def get_dropped_students(db_connection: Connection, instructor_id:
int,course_code: Optional[str] = None, section_number: Optional[int] =
None) -> List[EnrollmentListResponse]:
logger.info('Getting dropped students for instructor')
query = """
SELECT
Users.CWID AS StudentCWID,
Users.Name AS StudentFirstName,
Users.LastName AS StudentLastName,
Class.CourseCode AS CourseCode,
Section.SectionNumber AS SectionNumber,
Class.Name AS ClassName,
RegistrationList.Status AS Status
FROM
RegistrationList
JOIN Users ON RegistrationList.StudentID = Users.CWID
JOIN Section ON RegistrationList.CourseCode = Section.CourseCode AND
RegistrationList.SectionNumber = Section.SectionNumber
JOIN Class ON Section.CourseCode = Class.CourseCode
```

```
WHERE

Section.InstructorID = ?

AND RegistrationList.Status = 'dropped'

"""

params = [instructor_id]

if course_code is not None:

query += " AND Section.CourseCode = ?"

params.append(course_code)

if section_number is not None:

query += " AND Section.SectionNumber = ?"

params.append(section_number)

else:

query += " ORDER BY Class.CourseCode, Section.SectionNumber,
Users.LastName, Users.Name"

cur = db_connection.execute(query, tuple(params))

enrollment = cur.fetchall()

if not enrollment:

raise HTTPException(

status_code=status.HTTP_404_NOT_FOUND, detail="No students that dropped
found for instructor"

)

results = [{"student_cwid": row[0],

"student_first_name": row[1],

"student_last_name": row[2],

"course_code": row[3],

"section_number": row[4],

"class_name": row[5],

"status": row[6]} for row in enrollment]

return results


def get_waitlist_status(db_connection: Connection, student_id: int) ->
str:
```

```
logger.info('Checking waitlist position for student ', str(student_id))

query = f"""

WITH WaitlistPosition AS (

SELECT

rl.StudentID,

rl.CourseCode,

rl.SectionNumber,

rl.Status,

ROW_NUMBER() OVER (PARTITION BY rl.CourseCode, rl.SectionNumber ORDER BY
rl.EnrollmentDate) AS Position

FROM

RegistrationList rl

WHERE

rl.Status = 'waitlisted'

)

SELECT

wlp.Position,

wlp.CourseCode,

wlp.SectionNumber

FROM

WaitlistPosition wlp

WHERE

wlp.StudentID = {student_id}

"""

cursor = db_connection.cursor()

rows = cursor.execute(query)

if rows.arraysize == 0:

raise HTTPException(status_code= status.HTTP_400_BAD_REQUEST, detail=
f'Record not found.')

result = []

for row in rows:
```

```
logger.info(str(row))

waitlist = WaitlistPositionList(

waitlist_position = row[0],

section_number = row[2],

course_code = row[1]

)

result.append(waitlist)

print(result)

return result


def get_waitlist(db_connection: Connection, course_code: str,
section_number: int) -> list:

logger.info(f'fetching the students on the waitlist with coursecode and
section no {course_code}, {section_number}')

query = f"""

SELECT

r.StudentID,

u.Name AS StudentName,

r.EnrollmentDate,

r.Status

FROM

RegistrationList r

JOIN

Users u ON r.StudentID = u.CWID

WHERE

r.CourseCode = "{course_code}"

AND r.SectionNumber = {section_number}

AND r.Status = 'waitlisted'

ORDER BY

r.EnrollmentDate;

"""
```

```
cursor = db_connection.cursor()

rows = cursor.execute(query)

if rows.arraysize == 0:

raise HTTPException(status_code= status.HTTP_400_BAD_REQUEST, detail=
f'Records not found.')

# todo: throw appropriate error messages

result = []

for row in rows:

logger.info(str(row))

student = WaitlistStudents(

student_id = row[0],

student_name = row[1],

enrollment_date = row[2]

)

result.append(student)

logger.info(result)

return result




#Remove from Waitlist

def remove_student_from_waitlist(db_connection: Connection, student_id:
int, course_code: str, section_number: int) -> str:

try:

# Check if the student is on the waitlist for the specified course and
section

is_on_waitlist = check_student_on_waitlist(db_connection, student_id,
course_code, section_number)

if not is_on_waitlist:

return RemoveFromWaitlistRes(status="Student is not on the waitlist")
```

```
# Remove the student from the waitlist
remove_query = f"""
DELETE FROM RegistrationList WHERE StudentID = {student_id} AND
CourseCode = '{course_code}' AND SectionNumber = {section_number} AND
Status = 'waitlisted'
"""
cursor = db_connection.cursor()
cursor.execute("BEGIN")
cursor.execute(remove_query)
cursor.execute("COMMIT")
return RemoveFromWaitlistRes(status="Successfully removed student from
the waitlist")

except Exception as err:
raise HTTPException(status_code=status.HTTP_500_INTERNAL_SERVER_ERROR,
detail=str(err))

def check_student_on_waitlist(db_connection: Connection, student_id: int,
course_code: str, section_number: int) -> bool:
query = f"""
SELECT 1 FROM RegistrationList WHERE StudentID = {student_id} AND
CourseCode = '{course_code}' AND SectionNumber = {section_number} AND
Status = 'waitlisted'
"""
cursor = db_connection.cursor()
cursor.execute(query)
return cursor.fetchone() is not None
```

- Database connection and shutdown managed.

```
app = FastAPI()
DATABASE_URL = "./api/share/classes.db"
db_connection = sqlite3.connect(DATABASE_URL)
db_connection.isolation_level = None
```

```
@app.on_event("shutdown")
async def shutdown():
db_connection.close()
```

- Error handling for various scenarios.

Task 3: Models and Data Validation

What is to be done:

- Define Pydantic models for input and output data.
- Ensure data validation using Pydantic models.

Tools Used:

- Pydantic

Results:

- Pydantic models defined in’`models.py`’.

```
from pydantic import BaseModel
from enum import Enum


class AvailableClass(BaseModel):
course_code: str
course_name: str
department: str
instructor_first_name: str
instructor_last_name: str
current_enrollment: int
max_enrollment: int
waitlist: int
section_number: int
```

```
class AvailableClassResponse(BaseModel):

available_classes: List[AvailableClass]



class EnrollmentResponse(BaseModel):

enrollment_status: str

enrollment_date: Optional[datetime] = None


class EnrollmentRequest(BaseModel):

section_number: int

course_code: str

student_id: int


class RegistrationStatus(str, Enum):

ENROLLED = 'enrolled'

WAITLISTED = 'waitlisted'

NOT_ELIGIBLE = 'not_eligible'

DROPPED = "dropped"

ALREADY_ENROLLED = "already_enrolled"


class Registration(BaseModel):

section_number: int #Section Number

student_id: int

enrollment_status: str

course_code: str


class QueryStatus(str, Enum):

SUCCESS = "success"

FAILED = "failed"


class UserRole(str, Enum):
```

```
STUDENT = "student"

INSTRUCTOR = "instructor"

REGISTRAR = "registrar"

NOT_FOUND = "not_found"


class DropCourseResponse(BaseModel):

course_code: str

section_number: int

status: str

class AddClassRequest(BaseModel):

course_code: str

class_name: str

department: str

section_number: int

instructor_id: int

max_enrollment: int


class AddClassResponse(BaseModel):

addClass_status: str


class DeleteSectionResponse(BaseModel):

deleteSection_status: str


class DeleteSectionRequest(BaseModel):

course_code: str

section_number: int


class ChangeInstructorResponse(BaseModel):

changeInstructor_status: str


class ChangeInstructorRequest(BaseModel):
```

```
course_code: str

section_number: int

instructor_id: int


class FreezeEnrollmentResponse(BaseModel):

freezeEnrollment_status: str


class FreezeEnrollmentRequest(BaseModel):

course_code: str

section_number: int

# instructor models

class EnrollmentListResponse(BaseModel):

student_cwid: int

student_first_name: str

student_last_name: str

course_code: str

section_number: int

class_name: str

status: str


class RecordsEnrollmentResponse(BaseModel):

enrolled_students: List[EnrollmentListResponse]


class RecordsDroppedResponse(BaseModel):

dropped_students: List[EnrollmentListResponse]


class WaitlistPositionReq(BaseModel):

# section_number: int

# course_code: str

student_id: int
```

```
class WaitlistPositionList(BaseModel):

section_number: int

course_code: str

waitlist_position: int


class WaitlistPositionRes(BaseModel):

waitlist_positions: List[WaitlistPositionList]


class ViewWaitlistReq(BaseModel):

section_number: int

course_code: str



class WaitlistStudents(BaseModel):

student_id: int

student_name: str

enrollment_date: datetime


class ViewWaitlistRes(BaseModel):

waitlisted_students: List[WaitlistStudents]



#Waitlist

class WaitlistPositionList(BaseModel):

section_number: int

course_code: str

waitlist_position: int


class WaitlistPositionRes(BaseModel):

waitlist_positions: List[WaitlistPositionList]

class RemoveWaitlistReq(BaseModel):
```

```
student_id: int

section_number: int

course_code: str



class ViewWaitlistReq(BaseModel):

section_number: int

course_code: str


class WaitlistStudents(BaseModel):

student_id: int

student_name: str

enrollment_date: datetime


class ViewWaitlistRes(BaseModel):

waitlisted_students: List[WaitlistStudents]


class RemoveFromWaitlistRes(BaseModel):

status: str
```

- Models used for input validation in FastAPI endpoints.

```
@app.get(path="/classes", operation_id="available_classes",

response_model = AvailableClassResponse)

async def available_classes(department_name: str):
```