Fall 2023 CPSC 449-01

Project 1

Juan Uriarte

Project Tasks

Define the API

Create a RESTful service that mimics enrollment in college courses. The service should allow students to list available classes, attempt to enroll in a class, and drop a class. Instructors should be able to view current enrollment for their classes, view students who have dropped the class, and drop students administratively. The registrar should be able to add new classes and sections, remove existing sections, change the instructor for a section, and freeze automatic enrollment from waiting lists.

Create the database

The relational SQL database consists of 5 tables: instructors, classes, students, enrollments, and waiting lists. Each class is assigned a department, course code, section number, name and instructor ID. Each student will have a student ID, first name, last name, and email. Each instructor will have an ID, first name, last name, and email. The enrollment table will have an enrollment ID, student ID, class ID, enrollment date, and a dropped field. The waiting list table will have a waiting list ID, student ID, class ID, waiting list position, and date added.

Implement the service

FastAPI is used to define endpoints and representations of the actions that can be performed on the database. Endpoints make appropriate use of HTTP methods, status codes, headers, and have their inputs/outputs set to JSON format.

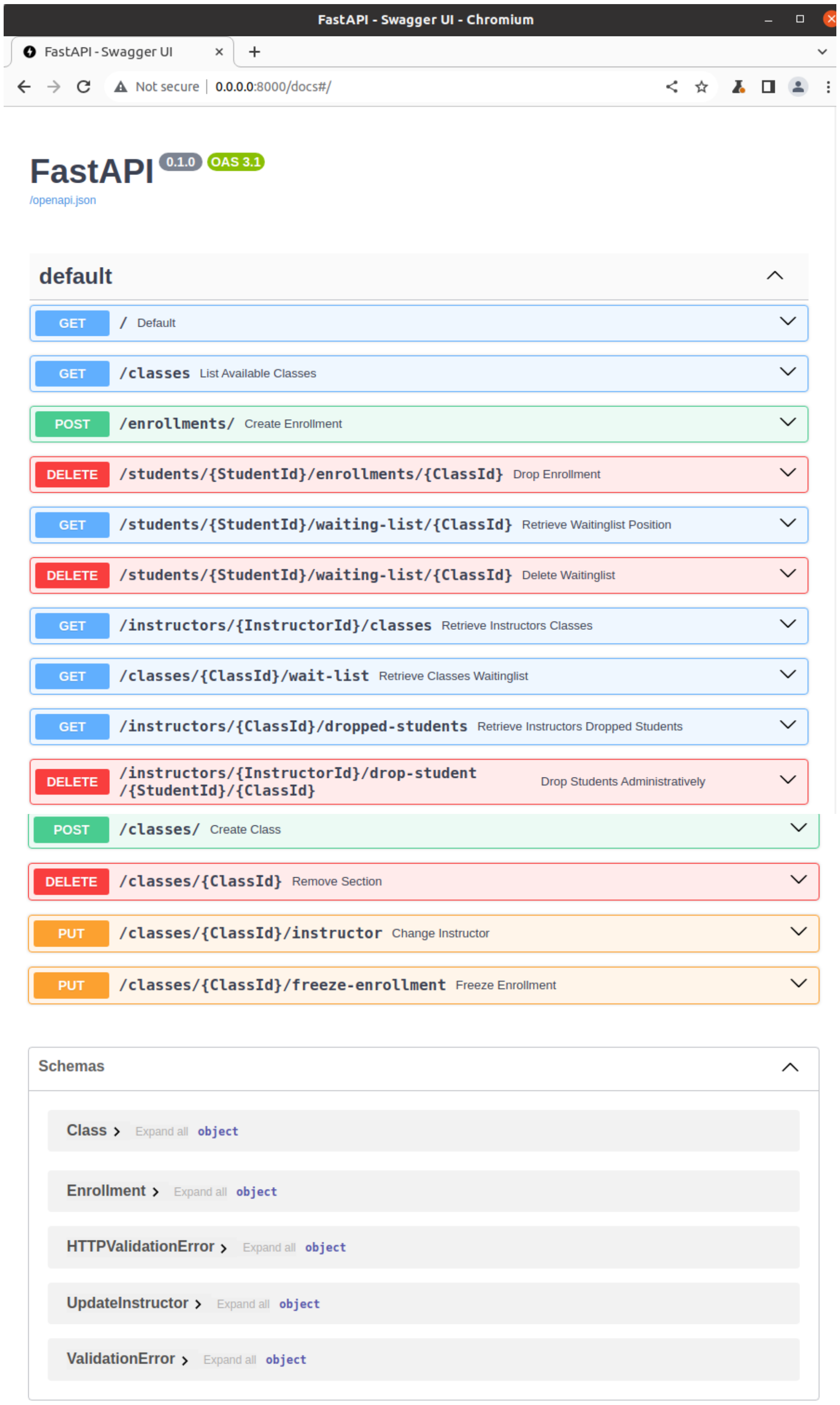
Tools used

We used the FastAPI framework, Swagger documentation framework, and Foreman to develop our project in Python. Additionally, our database code makes use of the Python Standard Library's sqlite3 module.

Results

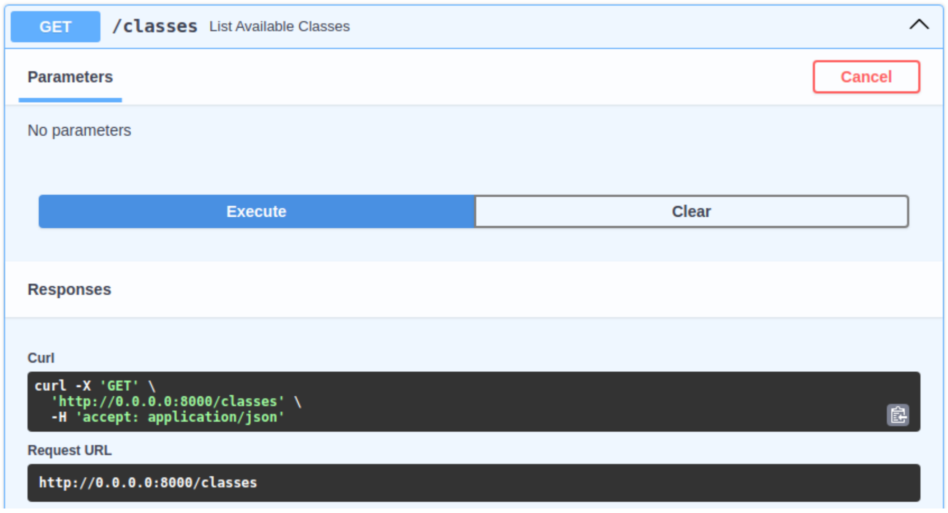
Running `foreman start`, our Restful API allows us to access the following page provided by the Swagger API, where the documentation for the database's PUT, POST, GET, and DELETE operations are displayed. The list of available classes, enrollment options, and drop/add options are all available to students here. Professors have access to information about student enrollment, drop-outs, and they can drop students administratively. The registrar can also alter instructors, create new sections, delete existing ones, and freeze automatic enrollment.

The following screenshot demonstrates a successful response to the GET classes command. Here we see that we receive all information pertaining to the available classes. Such as the class id, instructor id, department, section number , class name, max enrollment, etc.


FastAPI - Swagger UI - Chromium
FastAPI - Swagger UI
Not secure | 0.0.0.0:8000/docs#/
FastAPI 0.1.0 OAS 3.1
/openapi.json
default

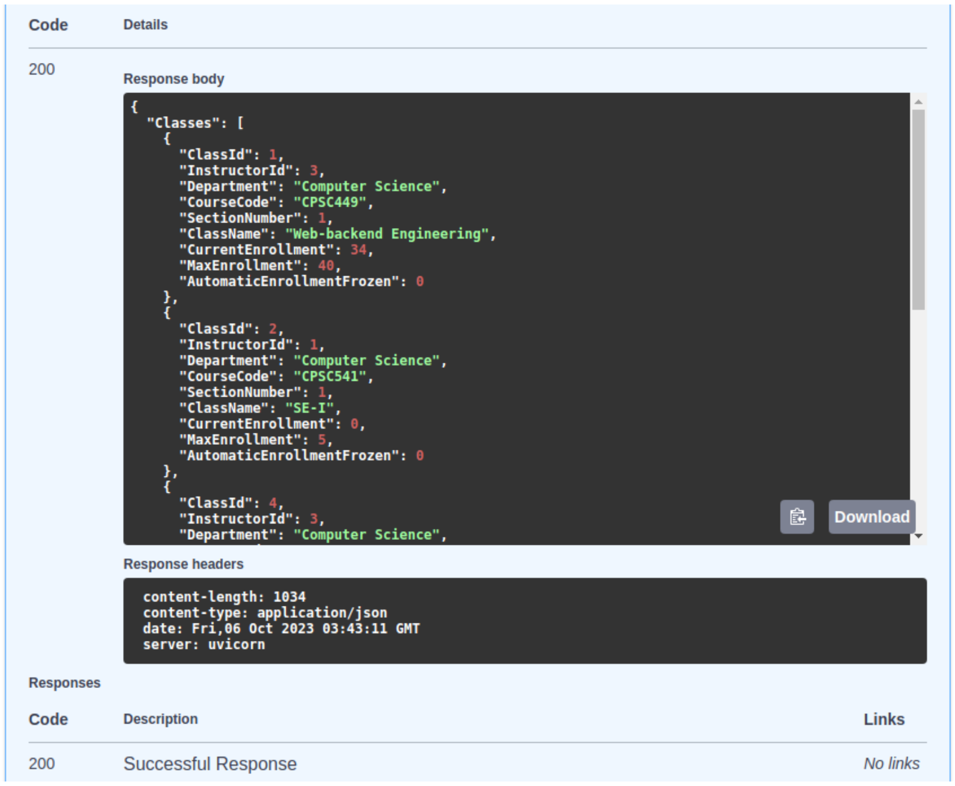
```
GET     /  Default
GET     /classes  List Available Classes
POST    /enrollments/  Create Enrollment
DELETE  /students/{StudentId}/enrollments/{ClassId}  Drop Enrollment
GET     /students/{StudentId}/waiting-list/{ClassId}  Retrieve Waitinglist Position
DELETE  /students/{StudentId}/waiting-list/{ClassId}  Delete Waitinglist
GET     /instructors/{InstructorId}/classes  Retrieve Instructors Classes
GET     /classes/{ClassId}/wait-list  Retrieve Classes Waitinglist
GET     /instructors/{ClassId}/dropped-students  Retrieve Instructors Dropped Students
DELETE  /instructors/{InstructorId}/drop-student/{StudentId}/{ClassId}  Drop Students Administratively
POST    /classes/  Create Class
DELETE  /classes/{ClassId}  Remove Section
PUT     /classes/{ClassId}/instructor  Change Instructor
PUT     /classes/{ClassId}/freeze-enrollment  Freeze Enrollment
```

Schemas
Class Expand all object
Enrollment Expand all object
HTTPValidationError Expand all object
UpdateInstructor Expand all object
ValidationError Expand all object



```
curl -X 'GET' \
  'http://0.0.0.0:8000/classes' \
  -H 'accept: application/json'
```

```
http://0.0.0.0:8000/classes
```



```
{
  "Classes": [
    {
      "ClassId": 1,
      "InstructorId": 3,
      "Department": "Computer Science",
      "CourseCode": "CPSC449",
      "SectionNumber": 1,
      "ClassName": "Web-backend Engineering",
      "CurrentEnrollment": 34,
      "MaxEnrollment": 40,
      "AutomaticEnrollmentFrozen": 0
    },
    {
      "ClassId": 2,
      "InstructorId": 1,
      "Department": "Computer Science",
      "CourseCode": "CPSC541",
      "SectionNumber": 1,
      "ClassName": "SE-I",
      "CurrentEnrollment": 0,
      "MaxEnrollment": 5,
      "AutomaticEnrollmentFrozen": 0
    },
    {
      "ClassId": 4,
      "InstructorId": 3,
      "Department": "Computer Science",
```

```
content-length: 1034
content-type: application/json
date: Fri,06 Oct 2023 03:43:11 GMT
server: uvicorn
```

Code	Description	Links
200	Successful Response	No links

Comments

An issue that we encountered in our project was getting the POST and GET to receive and create new data within the database. This was due to the error stating that our file was not a database which was then fixed by editing our pathing.

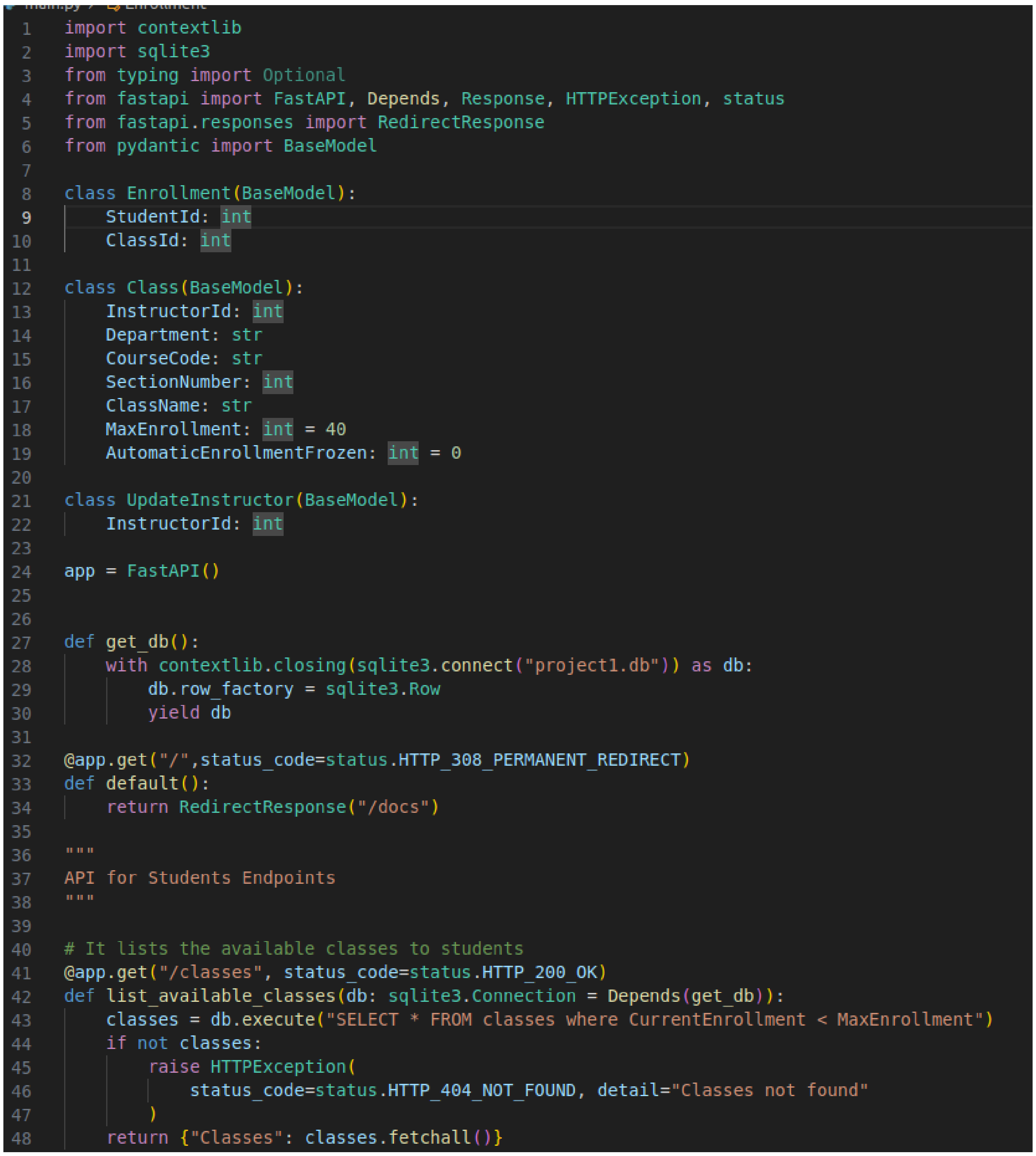
Code Review

Our project consists of three files which are main.py, Procfile, and project1_database.py.

main.py

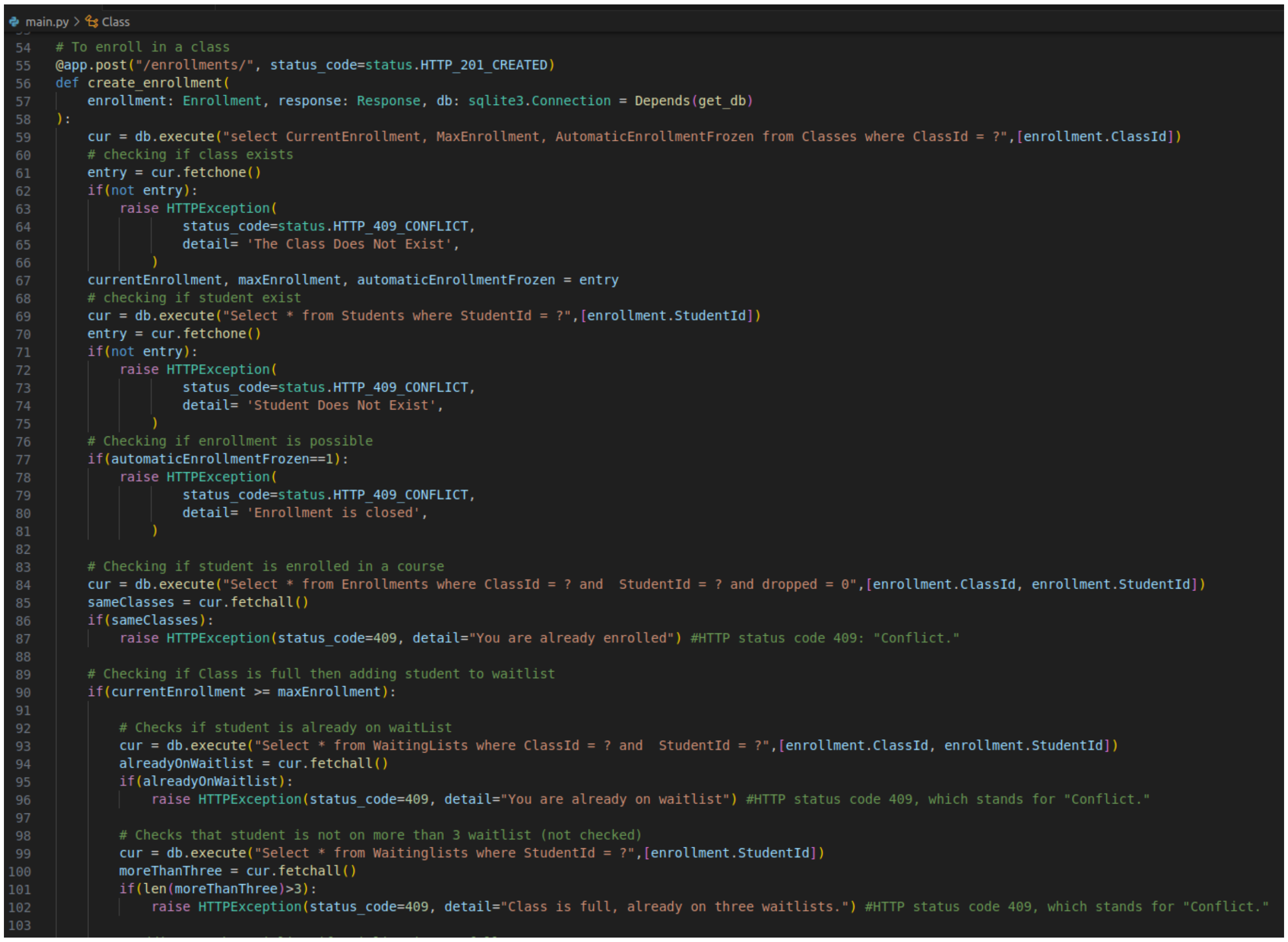
We used import statements such as contextlib to help connect with the database, sqlite3 to interact with the database, fastapi, and pydantic. We then created class “Enrollment” that represents the status of enrollment of a student in a class and class “Class” that represents a class that has instructor id, department, course code, section number, and other details. The max enrollment of our class is 40. We then have another class called “update instructor” that would represent the id of an instructor.

We then implemented app = fast api to start a new Fast API process. We then connect to the database named project1.db using the def get_db() statement and using sqlite3 row so that we can access the results. We then access the root endpoint of “/” that will take us to /docs endpoint. Our next endpoint is Students that will list us the available classes for students to take. The query of classes in the database will give us all the classes that are available and that does not exceed 40 which is the maximum enrollment per class. If there are no classes that have less than 40 students, it would return a 404 error. Otherwise, it would give us a list of classes.



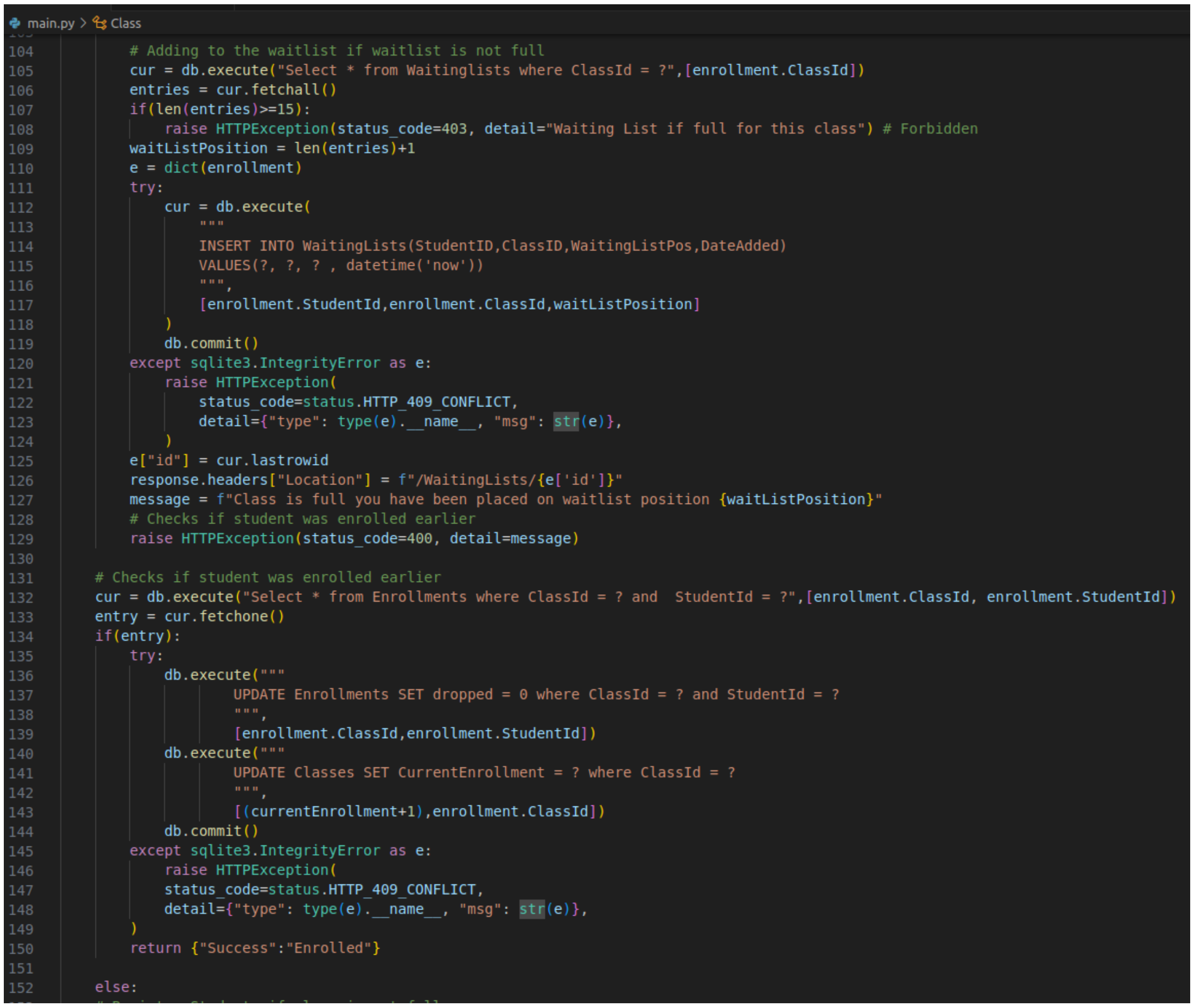
```python
import contextlib
import sqlite3
from typing import Optional
from fastapi import FastAPI, Depends, Response, HTTPException, status
from fastapi.responses import RedirectResponse
from pydantic import BaseModel

class Enrollment(BaseModel):
    StudentId: int
    ClassId: int

class Class(BaseModel):
    InstructorId: int
    Department: str
    CourseCode: str
    SectionNumber: int
    ClassName: str
    MaxEnrollment: int = 40
    AutomaticEnrollmentFrozen: int = 0

class UpdateInstructor(BaseModel):
    InstructorId: int

app = FastAPI()


def get_db():
    with contextlib.closing(sqlite3.connect("project1.db")) as db:
        db.row_factory = sqlite3.Row
        yield db

@app.get("/",status_code=status.HTTP_308_PERMANENT_REDIRECT)
def default():
    return RedirectResponse("/docs")

"""
API for Students Endpoints
"""

# It lists the available classes to students
@app.get("/classes", status_code=status.HTTP_200_OK)
def list_available_classes(db: sqlite3.Connection = Depends(get_db)):
    classes = db.execute("SELECT * FROM classes where CurrentEnrollment < MaxEnrollment")
    if not classes:
        raise HTTPException(
            status_code=status.HTTP_404_NOT_FOUND, detail="Classes not found"
        )
    return {"Classes": classes.fetchall()}
```

We create a POST endpoint for /enrollments/. Create enrollment has parameters enrollment, response, and db for connection to the database. The following request details from current enrollment and max enrollment of the class the student wants to enroll in. If the class does not exist status code 409 will be shown. It also checks if the student exists by implementing the student id number, if the student does not exist a status code 409 will be shown. It also checks if the student is already enrolled in the class. It then checks the given capacity of the class.



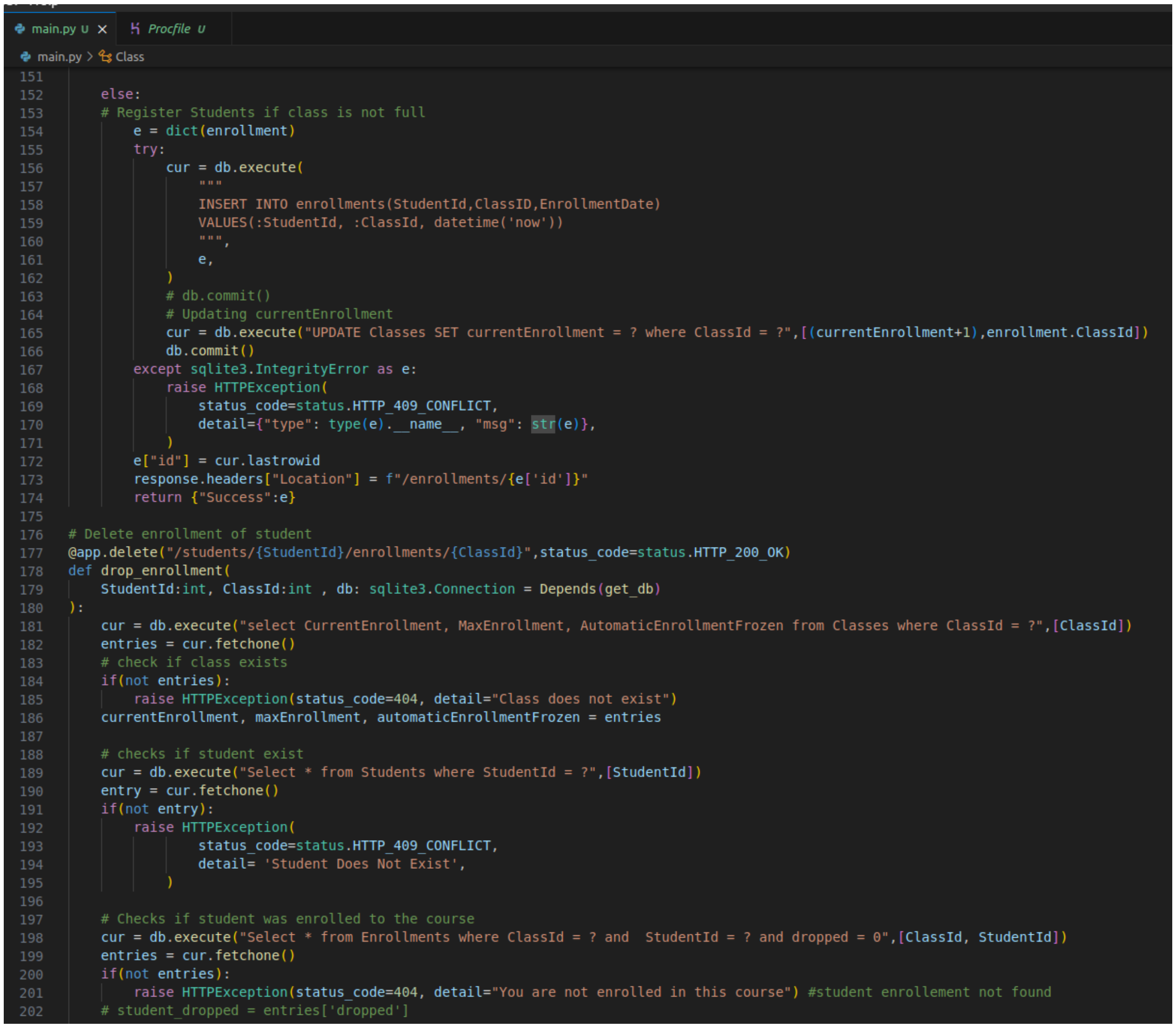
```python
# To enroll in a class
@app.post("/enrollments/", status_code=status.HTTP_201_CREATED)
def create_enrollment(
    enrollment: Enrollment, response: Response, db: sqlite3.Connection = Depends(get_db)
):
    cur = db.execute("select CurrentEnrollment, MaxEnrollment, AutomaticEnrollmentFrozen from Classes where ClassId = ?",[enrollment.ClassId])
    # checking if class exists
    entry = cur.fetchone()
    if(not entry):
        raise HTTPException(
                status_code=status.HTTP_409_CONFLICT,
                detail= 'The Class Does Not Exist',
            )
    currentEnrollment, maxEnrollment, automaticEnrollmentFrozen = entry
    # checking if student exist
    cur = db.execute("Select * from Students where StudentId = ?",[enrollment.StudentId])
    entry = cur.fetchone()
    if(not entry):
        raise HTTPException(
                status_code=status.HTTP_409_CONFLICT,
                detail= 'Student Does Not Exist',
            )
    # Checking if enrollment is possible
    if(automaticEnrollmentFrozen==1):
        raise HTTPException(
                status_code=status.HTTP_409_CONFLICT,
                detail= 'Enrollment is closed',
            )

    # Checking if student is enrolled in a course
    cur = db.execute("Select * from Enrollments where ClassId = ? and  StudentId = ? and dropped = 0",[enrollment.ClassId, enrollment.StudentId])
    sameClasses = cur.fetchall()
    if(sameClasses):
        raise HTTPException(status_code=409, detail="You are already enrolled") #HTTP status code 409: "Conflict."

    # Checking if Class is full then adding student to waitlist
    if(currentEnrollment >= maxEnrollment):

        # Checks if student is already on waitList
        cur = db.execute("Select * from WaitingLists where ClassId = ? and  StudentId = ?",[enrollment.ClassId, enrollment.StudentId])
        alreadyOnWaitlist = cur.fetchall()
        if(alreadyOnWaitlist):
            raise HTTPException(status_code=409, detail="You are already on waitlist") #HTTP status code 409, which stands for "Conflict."

        # Checks that student is not on more than 3 waitlist (not checked)
        cur = db.execute("Select * from Waitinglists where StudentId = ?",[enrollment.StudentId])
        moreThanThree = cur.fetchall()
        if(len(moreThanThree)>3):
            raise HTTPException(status_code=409, detail="Class is full, already on three waitlists.") #HTTP status code 409, which stands for "Conflict."

```

The waitlist is checked by checking the current waitlist number of a class. If the waitlist is more than 15, status code 403 will be shown. If the waitlist is less than 15, the student is put on the waitlist and then inserted into the waitlist table with a time and date. It then shows the student the classroom is full and was put on the waitlist. Previous enrollment is checked if the student was already enrolled in the class. If the student was already enrolled in the class, the table is updated and the current enrollment is updated by 1. If there is an error a status code 409 will be shown otherwise a success message will be shown.



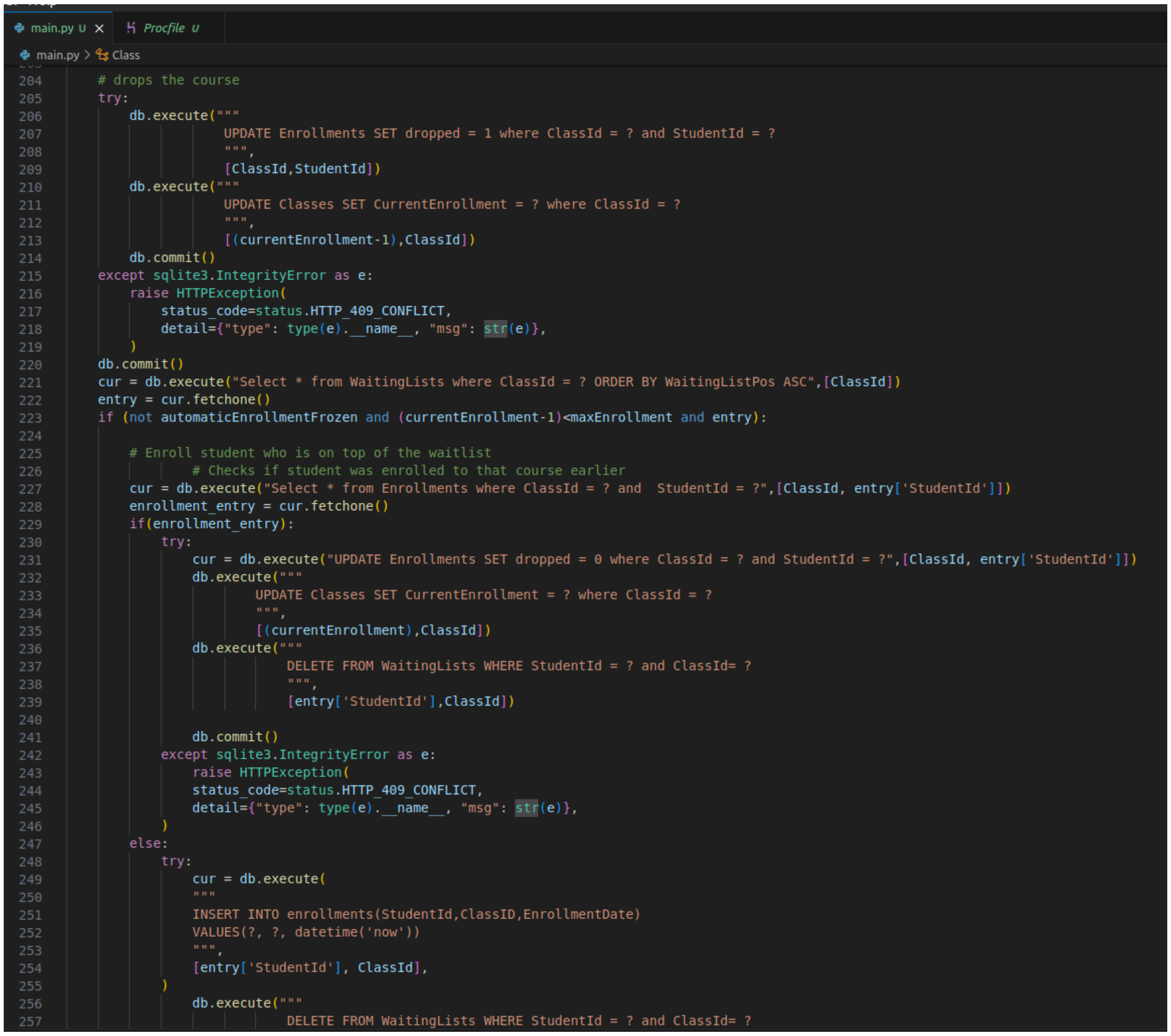
```python
        # Adding to the waitlist if waitlist is not full
        cur = db.execute("Select * from Waitinglists where ClassId = ?",[enrollment.ClassId])
        entries = cur.fetchall()
        if(len(entries)>=15):
            raise HTTPException(status_code=403, detail="Waiting List if full for this class") # Forbidden
        waitListPosition = len(entries)+1
        e = dict(enrollment)
        try:
            cur = db.execute(
                """
                INSERT INTO WaitingLists(StudentID,ClassID,WaitingListPos,DateAdded)
                VALUES(?, ?, ? , datetime('now'))
                """,
                [enrollment.StudentId,enrollment.ClassId,waitListPosition]
            )
            db.commit()
        except sqlite3.IntegrityError as e:
            raise HTTPException(
                status_code=status.HTTP_409_CONFLICT,
                detail={"type": type(e).__name__, "msg": str(e)},
            )
        e["id"] = cur.lastrowid
        response.headers["Location"] = f"/WaitingLists/{e['id']}"
        message = f"Class is full you have been placed on waitlist position {waitListPosition}"
        # Checks if student was enrolled earlier
        raise HTTPException(status_code=400, detail=message)

    # Checks if student was enrolled earlier
    cur = db.execute("Select * from Enrollments where ClassId = ? and  StudentId = ?",[enrollment.ClassId, enrollment.StudentId])
    entry = cur.fetchone()
    if(entry):
        try:
            db.execute("""
                    UPDATE Enrollments SET dropped = 0 where ClassId = ? and StudentId = ?
                    """,
                    [enrollment.ClassId,enrollment.StudentId])
            db.execute("""
                    UPDATE Classes SET CurrentEnrollment = ? where ClassId = ?
                    """,
                    [(currentEnrollment+1),enrollment.ClassId])
            db.commit()
        except sqlite3.IntegrityError as e:
            raise HTTPException(
            status_code=status.HTTP_409_CONFLICT,
            detail={"type": type(e).__name__, "msg": str(e)},
        )
        return {"Success":"Enrolled"}

    else:
```

If the student was not enrolled in the class, the student will then be inserted in the enrollment table with the student id, class id and the date. It then updates the current enrollment number by one. The endpoint of deleting enrollment has the drop enrollment with parameters student id and class id including the database. It then checks details of the classes with the given class id, if the class does not exist, status code 404 will be shown. It then checks if the student exists in the table of students by checking the student id, if the student does not exist, status code 409 will be shown. Lastly, it checks if the student is currently enrolled, if not the status code 404 will be shown.



```python

    else:
    # Register Students if class is not full
        e = dict(enrollment)
        try:
            cur = db.execute(
                """
                INSERT INTO enrollments(StudentId,ClassID,EnrollmentDate)
                VALUES(:StudentId, :ClassId, datetime('now'))
                """,
                e,
            )
            # db.commit()
            # Updating currentEnrollment
            cur = db.execute("UPDATE Classes SET currentEnrollment = ? where ClassId = ?",[(currentEnrollment+1),enrollment.ClassId])
            db.commit()
        except sqlite3.IntegrityError as e:
            raise HTTPException(
                status_code=status.HTTP_409_CONFLICT,
                detail={"type": type(e).__name__, "msg": str(e)},
            )
        e["id"] = cur.lastrowid
        response.headers["Location"] = f"/enrollments/{e['id']}"
        return {"Success":e}

# Delete enrollment of student
@app.delete("/students/{StudentId}/enrollments/{ClassId}",status_code=status.HTTP_200_OK)
def drop_enrollment(
    StudentId:int, ClassId:int , db: sqlite3.Connection = Depends(get_db)
):
    cur = db.execute("select CurrentEnrollment, MaxEnrollment, AutomaticEnrollmentFrozen from Classes where ClassId = ?",[ClassId])
    entries = cur.fetchone()
    # check if class exists
    if(not entries):
        raise HTTPException(status_code=404, detail="Class does not exist")
    currentEnrollment, maxEnrollment, automaticEnrollmentFrozen = entries

    # checks if student exist
    cur = db.execute("Select * from Students where StudentId = ?",[StudentId])
    entry = cur.fetchone()
    if(not entry):
        raise HTTPException(
                status_code=status.HTTP_409_CONFLICT,
                detail= 'Student Does Not Exist',
            )

    # Checks if student was enrolled to the course
    cur = db.execute("Select * from Enrollments where ClassId = ? and  StudentId = ? and dropped = 0",[ClassId, StudentId])
    entries = cur.fetchone()
    if(not entries):
        raise HTTPException(status_code=404, detail="You are not enrolled in this course") #student enrollement not found
    # student_dropped = entries['dropped']
```

The database is then updated if the course is dropped. Then the student from the top of the waitlist will get automatically enrolled to the class if the conditions are met.If the student dropped the course previously, the dropped status in the enrollment table is set to 0. Then the current enrollment count for classes remains the same.

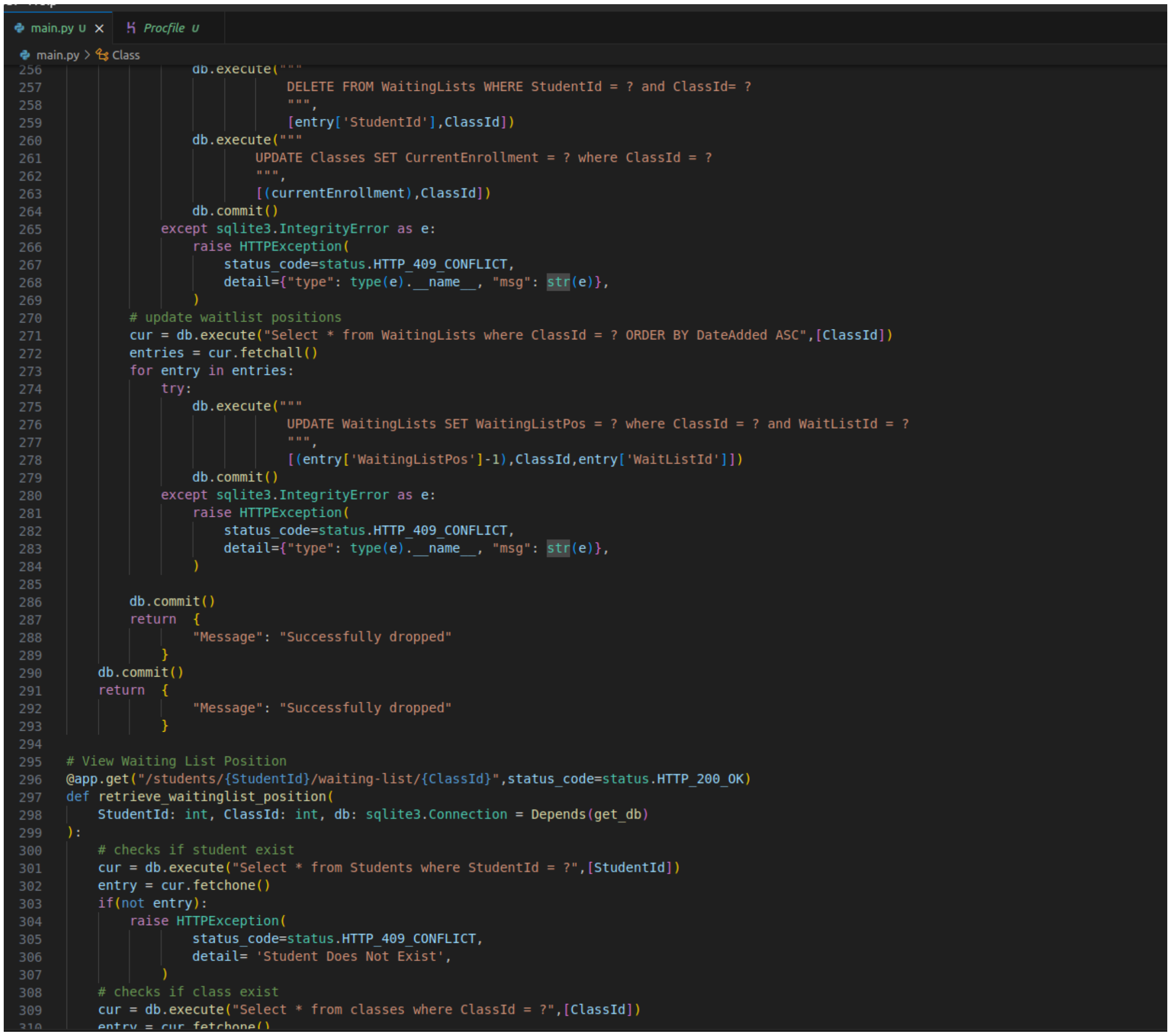


main.py U × Procfile U

main.py > Class

```
    # drops the course
    try:
        db.execute("""
                    UPDATE Enrollments SET dropped = 1 where ClassId = ? and StudentId = ?
                    """,
                    [ClassId,StudentId])
        db.execute("""
                    UPDATE Classes SET CurrentEnrollment = ? where ClassId = ?
                    """,
                    [(currentEnrollment-1),ClassId])
        db.commit()
    except sqlite3.IntegrityError as e:
        raise HTTPException(
            status_code=status.HTTP_409_CONFLICT,
            detail={"type": type(e).__name__, "msg": str(e)},
        )
    db.commit()
    cur = db.execute("Select * from WaitingLists where ClassId = ? ORDER BY WaitingListPos ASC",[ClassId])
    entry = cur.fetchone()
    if (not automaticEnrollmentFrozen and (currentEnrollment-1)<maxEnrollment and entry):

        # Enroll student who is on top of the waitlist
                # Checks if student was enrolled to that course earlier
        cur = db.execute("Select * from Enrollments where ClassId = ? and  StudentId = ?",[ClassId, entry['StudentId']])
        enrollment_entry = cur.fetchone()
        if(enrollment_entry):
            try:
                cur = db.execute("UPDATE Enrollments SET dropped = 0 where ClassId = ? and StudentId = ?",[ClassId, entry['StudentId']])
                db.execute("""
                        UPDATE Classes SET CurrentEnrollment = ? where ClassId = ?
                        """,
                        [(currentEnrollment),ClassId])
                db.execute("""
                            DELETE FROM WaitingLists WHERE StudentId = ? and ClassId= ?
                            """,
                            [entry['StudentId'],ClassId])

                db.commit()
            except sqlite3.IntegrityError as e:
                raise HTTPException(
                status_code=status.HTTP_409_CONFLICT,
                detail={"type": type(e).__name__, "msg": str(e)},
            )
        else:
            try:
                cur = db.execute(
                """
                INSERT INTO enrollments(StudentId,ClassID,EnrollmentDate)
                VALUES(?, ?, datetime('now'))
                """,
                [entry['StudentId'], ClassId],
            )
                db.execute("""
                            DELETE FROM WaitingLists WHERE StudentId = ? and ClassId= ?
```

If a student drops a course, the next student on the waitlist is next to fill the spot. If the student was never enrolled, a new entry is added to the enrollment table with the current date. Then the current enrollment number is incremented by 1. After a student from the waitlist was enrolled, the slots of the other students must change. The positions of all students are decremented by 1 so that the waitlist has no gaps. The endpoint of waitlist position allows a student to get from the waitlist, if checks if the student exists by checking the student id number in the students table. If the student does not exist, status code 409 will be shown.

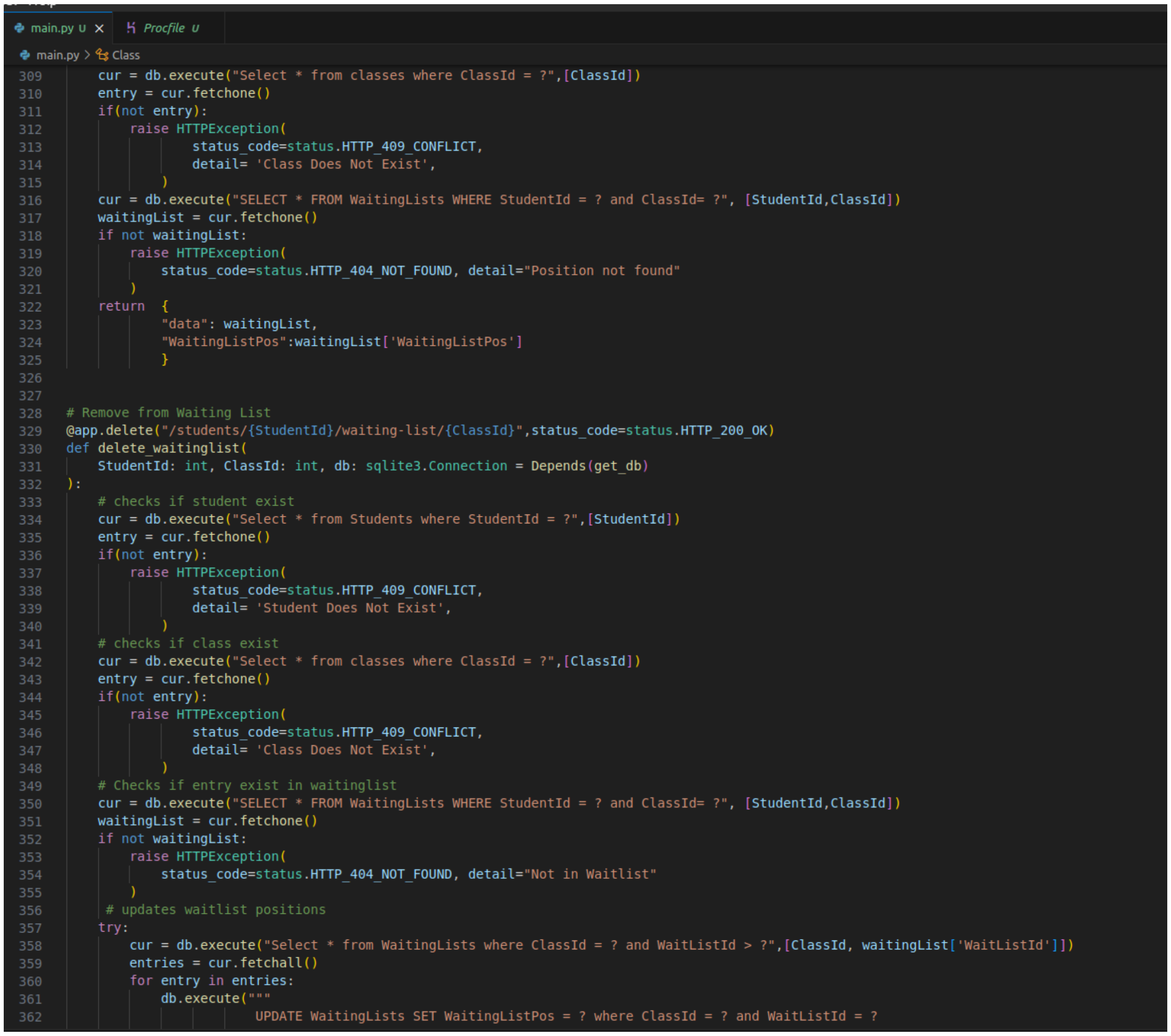


main.py U × Procfile U

main.py > Class

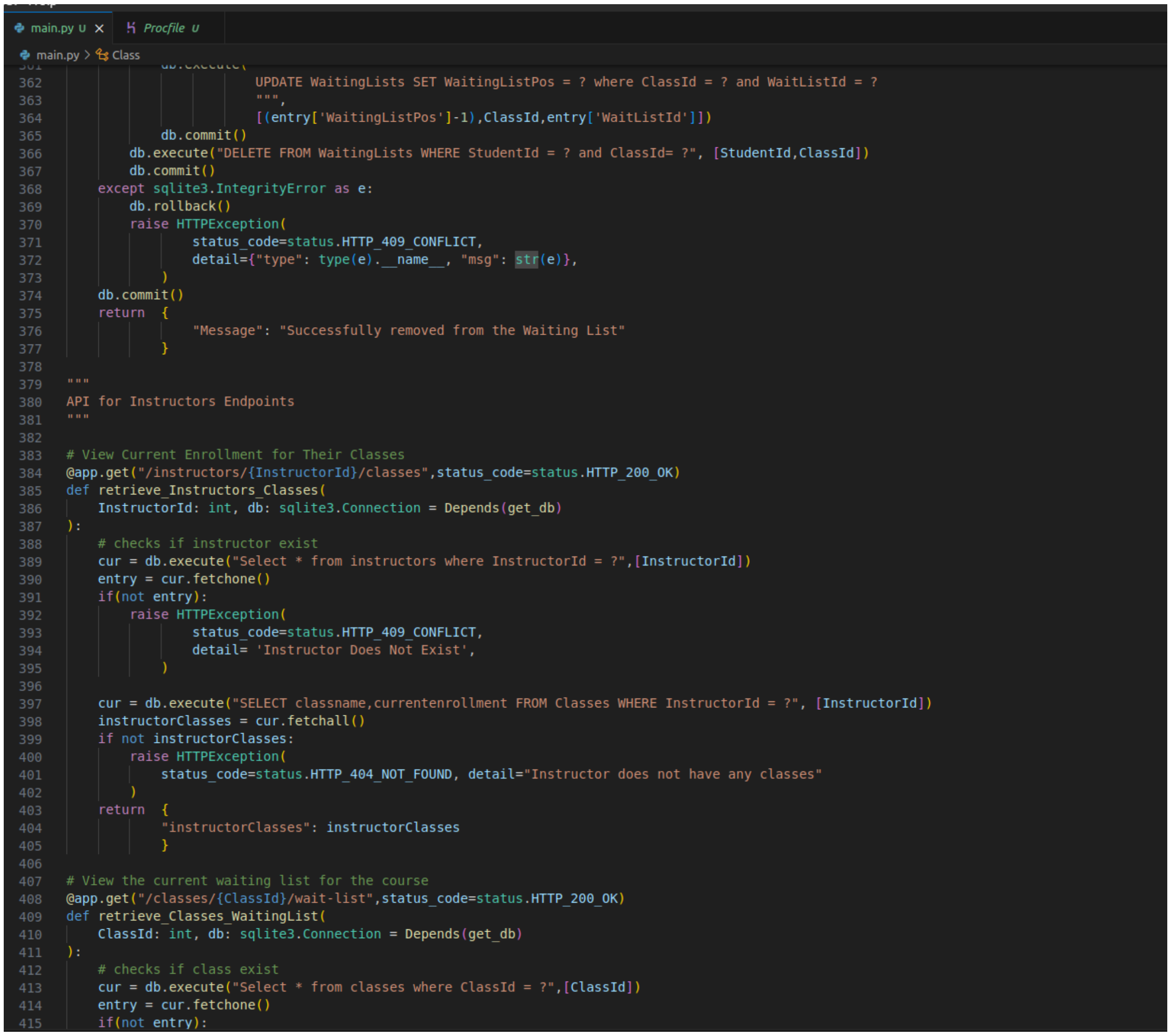
```python
                db.execute("""
                            DELETE FROM WaitingLists WHERE StudentId = ? and ClassId= ?
                            """,
                            [entry['StudentId'],ClassId])
                db.execute("""
                        UPDATE Classes SET CurrentEnrollment = ? where ClassId = ?
                        """,
                        [(currentEnrollment),ClassId])
                db.commit()
            except sqlite3.IntegrityError as e:
                raise HTTPException(
                    status_code=status.HTTP_409_CONFLICT,
                    detail={"type": type(e).__name__, "msg": str(e)},
                )
        # update waitlist positions
        cur = db.execute("Select * from WaitingLists where ClassId = ? ORDER BY DateAdded ASC",[ClassId])
        entries = cur.fetchall()
        for entry in entries:
            try:
                db.execute("""
                            UPDATE WaitingLists SET WaitingListPos = ? where ClassId = ? and WaitListId = ?
                            """,
                            [(entry['WaitingListPos']-1),ClassId,entry['WaitListId']])
                db.commit()
            except sqlite3.IntegrityError as e:
                raise HTTPException(
                    status_code=status.HTTP_409_CONFLICT,
                    detail={"type": type(e).__name__, "msg": str(e)},
                )

        db.commit()
        return  {
                "Message": "Successfully dropped"
            }
    db.commit()
    return  {
                "Message": "Successfully dropped"
            }

# View Waiting List Position
@app.get("/students/{StudentId}/waiting-list/{ClassId}",status_code=status.HTTP_200_OK)
def retrieve_waitinglist_position(
    StudentId: int, ClassId: int, db: sqlite3.Connection = Depends(get_db)
):
    # checks if student exist
    cur = db.execute("Select * from Students where StudentId = ?",[StudentId])
    entry = cur.fetchone()
    if(not entry):
        raise HTTPException(
                status_code=status.HTTP_409_CONFLICT,
                detail= 'Student Does Not Exist',
            )
    # checks if class exist
    cur = db.execute("Select * from classes where ClassId = ?",[ClassId])
    entry = cur.fetchone()
```

If the class exists and the student id is on the waitlist, it returns the waitlist. Removing a student from the waitlist list of a class checks if the student id exists in the table, checks if the class exists in the class table, verifies if the student is on the waitlist table for that specific class. If the student is found, the positions of the waitlist will be updated and will delete the current student in the table.



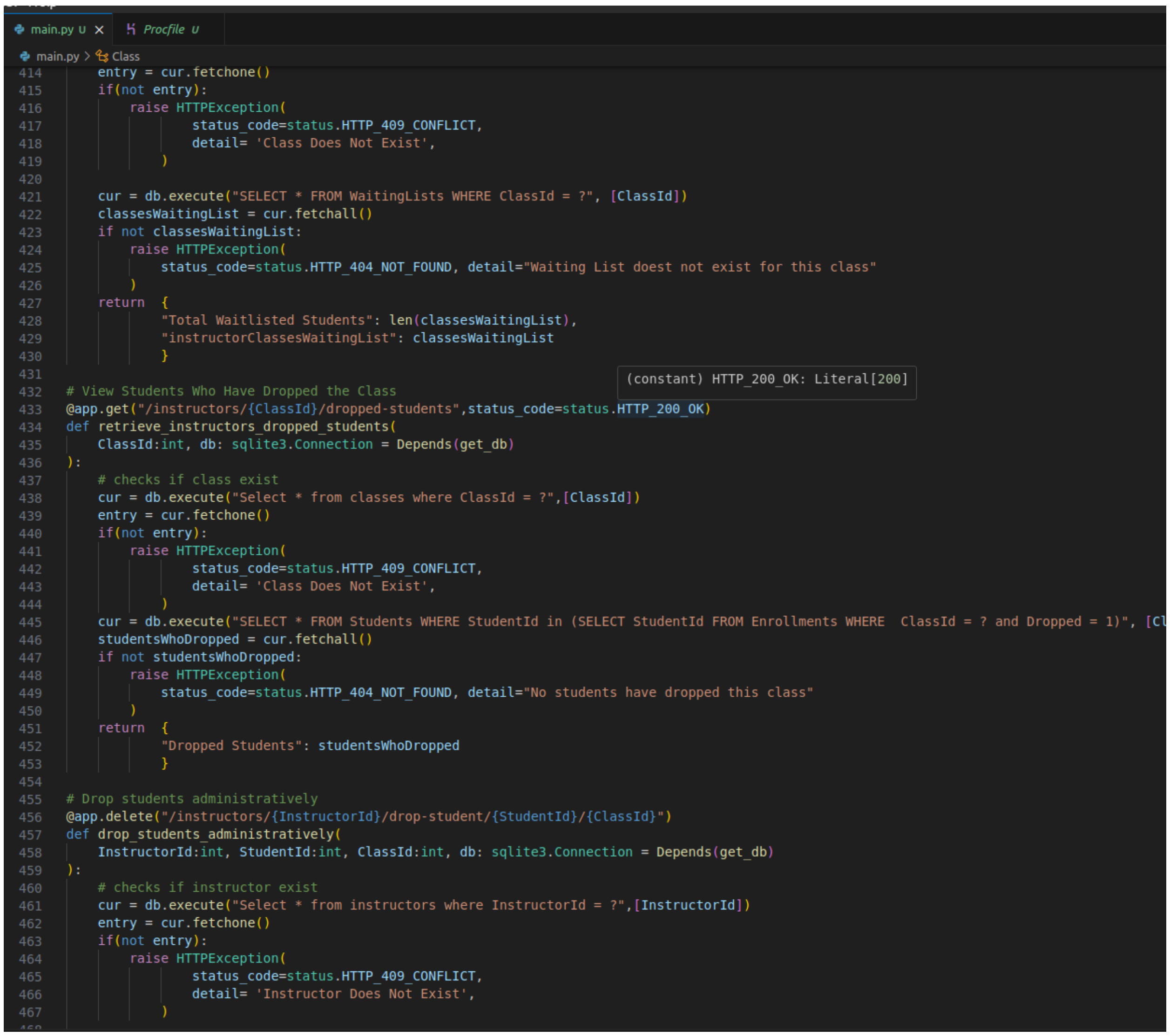
```python
    cur = db.execute("Select * from classes where ClassId = ?",[ClassId])
    entry = cur.fetchone()
    if(not entry):
        raise HTTPException(
                status_code=status.HTTP_409_CONFLICT,
                detail= 'Class Does Not Exist',
            )
    cur = db.execute("SELECT * FROM WaitingLists WHERE StudentId = ? and ClassId= ?", [StudentId,ClassId])
    waitingList = cur.fetchone()
    if not waitingList:
        raise HTTPException(
            status_code=status.HTTP_404_NOT_FOUND, detail="Position not found"
        )
    return  {
            "data": waitingList,
            "WaitingListPos":waitingList['WaitingListPos']
            }


# Remove from Waiting List
@app.delete("/students/{StudentId}/waiting-list/{ClassId}",status_code=status.HTTP_200_OK)
def delete_waitinglist(
    StudentId: int, ClassId: int, db: sqlite3.Connection = Depends(get_db)
):
    # checks if student exist
    cur = db.execute("Select * from Students where StudentId = ?",[StudentId])
    entry = cur.fetchone()
    if(not entry):
        raise HTTPException(
                status_code=status.HTTP_409_CONFLICT,
                detail= 'Student Does Not Exist',
            )
    # checks if class exist
    cur = db.execute("Select * from classes where ClassId = ?",[ClassId])
    entry = cur.fetchone()
    if(not entry):
        raise HTTPException(
                status_code=status.HTTP_409_CONFLICT,
                detail= 'Class Does Not Exist',
            )
    # Checks if entry exist in waitinglist
    cur = db.execute("SELECT * FROM WaitingLists WHERE StudentId = ? and ClassId= ?", [StudentId,ClassId])
    waitingList = cur.fetchone()
    if not waitingList:
        raise HTTPException(
            status_code=status.HTTP_404_NOT_FOUND, detail="Not in Waitlist"
        )
     # updates waitlist positions
    try:
        cur = db.execute("Select * from WaitingLists where ClassId = ? and WaitListId > ?",[ClassId, waitingList['WaitListId']])
        entries = cur.fetchall()
        for entry in entries:
            db.execute("""
                        UPDATE WaitingLists SET WaitingListPos = ? where ClassId = ? and WaitListId = ?
```

The SQL command update waitlists decrements the student in the waitlist by 1. Each student will then be moved by 1 position in the waitlist. The SQL command Delete removes the student from the waitlist for that specific class. If there is an error, status code 409 will be shown, otherwise a return message will be shown. Then there is an API for Instructors where they are able to view the current enrollment and the waitlist of their classes. The endpoint gets all the classes being taught by the professor, it then checks if the professor exists in the database. If the instructor exists, it grabs the names and the current enrollment number of all the classes associated with the professor. Otherwise it will print that the instructor does not have any classes.



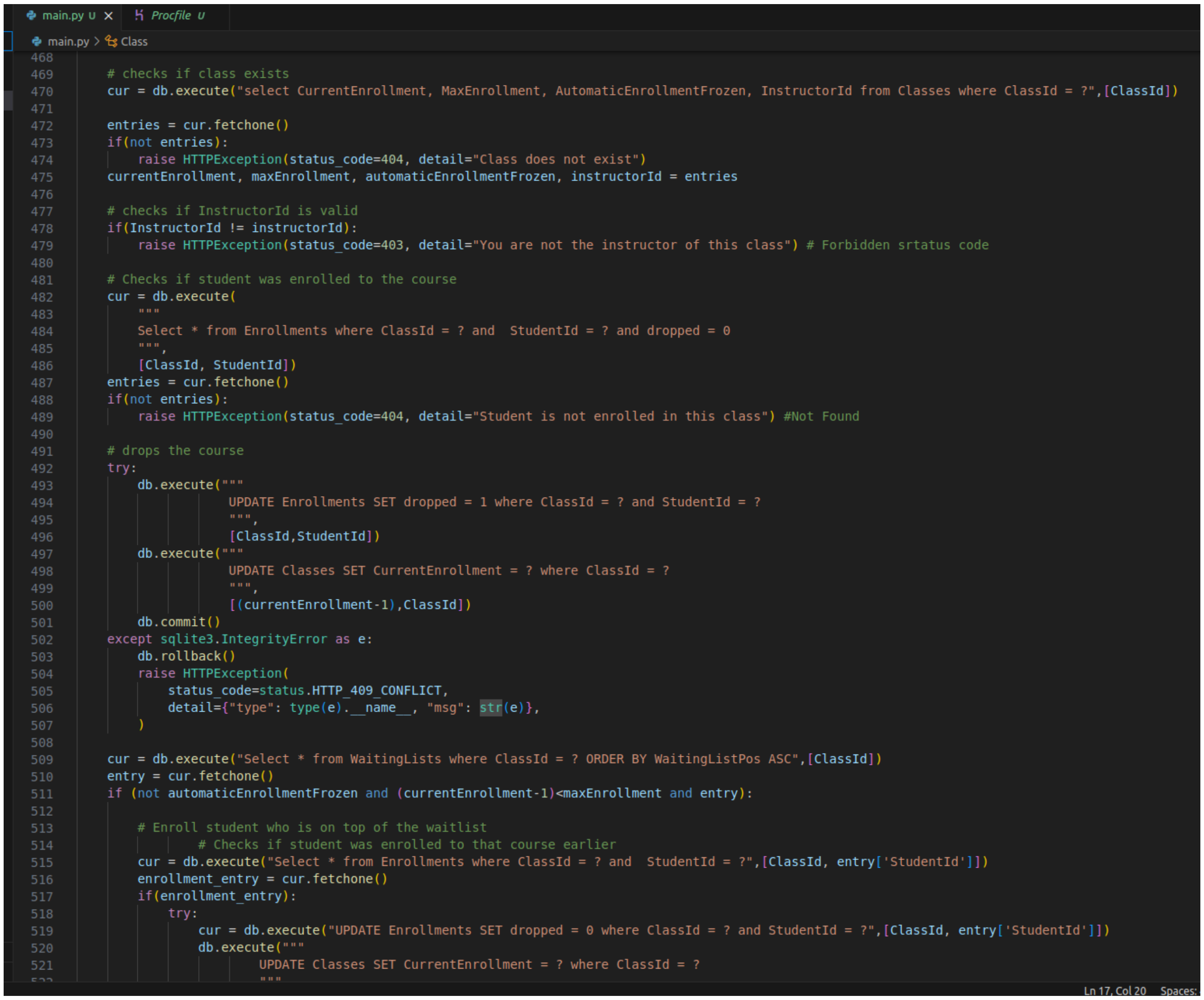
```
                        UPDATE WaitingLists SET WaitingListPos = ? where ClassId = ? and WaitListId = ?
                        """,
                        [(entry['WaitingListPos']-1),ClassId,entry['WaitListId']])
            db.commit()
        db.execute("DELETE FROM WaitingLists WHERE StudentId = ? and ClassId= ?", [StudentId,ClassId])
        db.commit()
    except sqlite3.IntegrityError as e:
        db.rollback()
        raise HTTPException(
                status_code=status.HTTP_409_CONFLICT,
                detail={"type": type(e).__name__, "msg": str(e)},
            )
    db.commit()
    return  {
                "Message": "Successfully removed from the Waiting List"
            }

"""
API for Instructors Endpoints
"""

# View Current Enrollment for Their Classes
@app.get("/instructors/{InstructorId}/classes",status_code=status.HTTP_200_OK)
def retrieve_Instructors_Classes(
    InstructorId: int, db: sqlite3.Connection = Depends(get_db)
):
    # checks if instructor exist
    cur = db.execute("Select * from instructors where InstructorId = ?",[InstructorId])
    entry = cur.fetchone()
    if(not entry):
        raise HTTPException(
                status_code=status.HTTP_409_CONFLICT,
                detail= 'Instructor Does Not Exist',
            )

    cur = db.execute("SELECT classname,currentenrollment FROM Classes WHERE InstructorId = ?", [InstructorId])
    instructorClasses = cur.fetchall()
    if not instructorClasses:
        raise HTTPException(
            status_code=status.HTTP_404_NOT_FOUND, detail="Instructor does not have any classes"
        )
    return  {
            "instructorClasses": instructorClasses
            }

# View the current waiting list for the course
@app.get("/classes/{ClassId}/wait-list",status_code=status.HTTP_200_OK)
def retrieve_Classes_WaitingList(
    ClassId: int, db: sqlite3.Connection = Depends(get_db)
):
    # checks if class exist
    cur = db.execute("Select * from classes where ClassId = ?",[ClassId])
    entry = cur.fetchone()
    if(not entry):
```

If the class does not exist, status code 409 will be shown. If there is no waitlist for that class status code 404 will be shown. Then we are able to view students who have dropped a class by creating an endpoint that enables us to have a list of students. If no students dropped the class a status code 404 will be shown. Instructors are also able to drop students from the waitlist by creating an endpoint delete, if the instructor does not exist, status code 409 is shown.



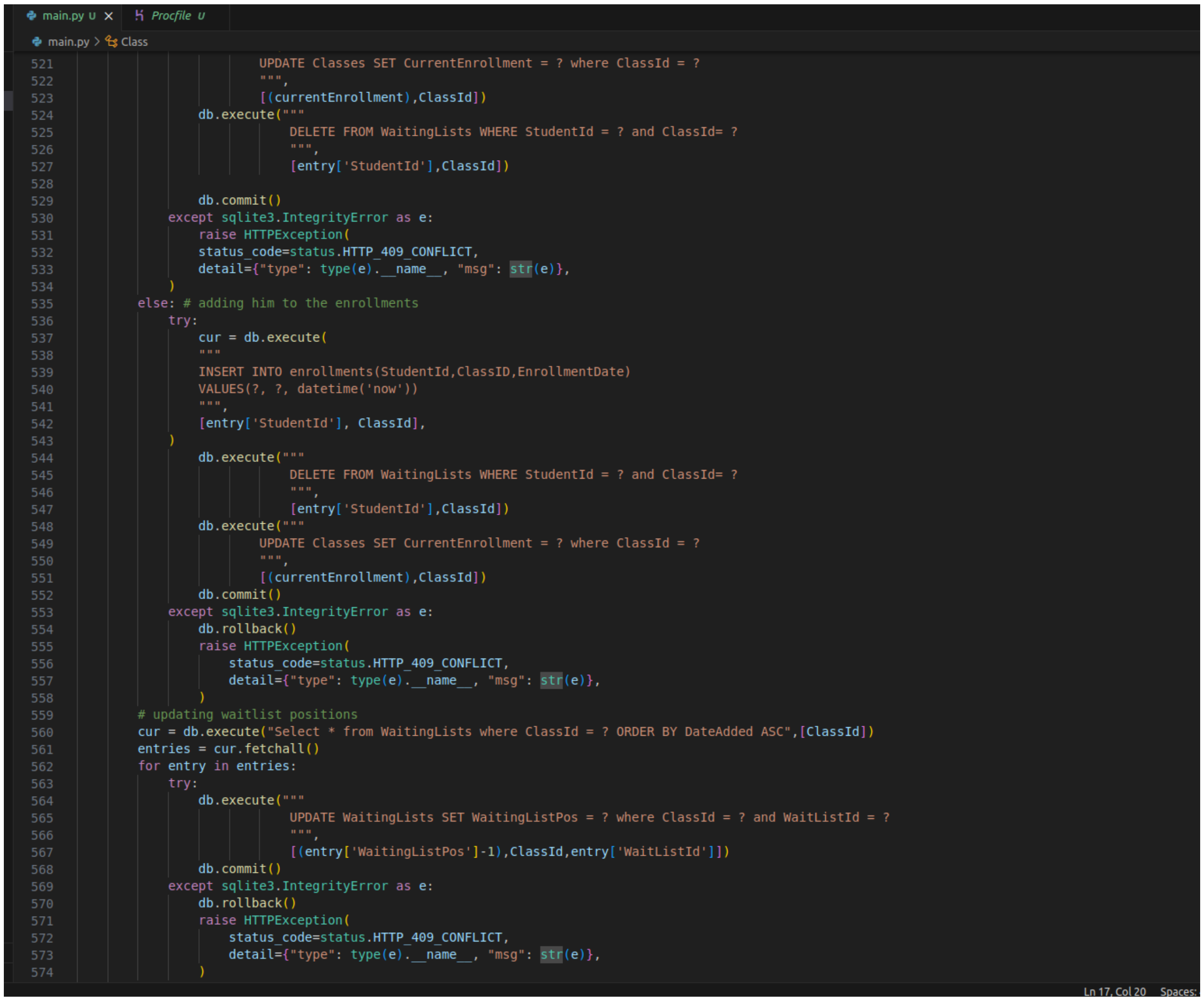
```python
    entry = cur.fetchone()
    if(not entry):
        raise HTTPException(
                status_code=status.HTTP_409_CONFLICT,
                detail= 'Class Does Not Exist',
            )

    cur = db.execute("SELECT * FROM WaitingLists WHERE ClassId = ?", [ClassId])
    classesWaitingList = cur.fetchall()
    if not classesWaitingList:
        raise HTTPException(
            status_code=status.HTTP_404_NOT_FOUND, detail="Waiting List doest not exist for this class"
        )
    return  {
            "Total Waitlisted Students": len(classesWaitingList),
            "instructorClassesWaitingList": classesWaitingList
            }

# View Students Who Have Dropped the Class
@app.get("/instructors/{ClassId}/dropped-students",status_code=status.HTTP_200_OK)
def retrieve_instructors_dropped_students(
    ClassId:int, db: sqlite3.Connection = Depends(get_db)
):
    # checks if class exist
    cur = db.execute("Select * from classes where ClassId = ?",[ClassId])
    entry = cur.fetchone()
    if(not entry):
        raise HTTPException(
                status_code=status.HTTP_409_CONFLICT,
                detail= 'Class Does Not Exist',
            )
    cur = db.execute("SELECT * FROM Students WHERE StudentId in (SELECT StudentId FROM Enrollments WHERE  ClassId = ? and Dropped = 1)", [Cl
    studentsWhoDropped = cur.fetchall()
    if not studentsWhoDropped:
        raise HTTPException(
            status_code=status.HTTP_404_NOT_FOUND, detail="No students have dropped this class"
        )
    return  {
            "Dropped Students": studentsWhoDropped
            }

# Drop students administratively
@app.delete("/instructors/{InstructorId}/drop-student/{StudentId}/{ClassId}")
def drop_students_administratively(
    InstructorId:int, StudentId:int, ClassId:int, db: sqlite3.Connection = Depends(get_db)
):
    # checks if instructor exist
    cur = db.execute("Select * from instructors where InstructorId = ?",[InstructorId])
    entry = cur.fetchone()
    if(not entry):
        raise HTTPException(
                status_code=status.HTTP_409_CONFLICT,
                detail= 'Instructor Does Not Exist',
            )
```

If class exists, it grabs details from the class id, otherwise status code 404 will be shown. If the instructor id does not match to the specific class, status code 403 will be shown. It also checks if the student dropped the class and handles automatic enrollment from waitlists.



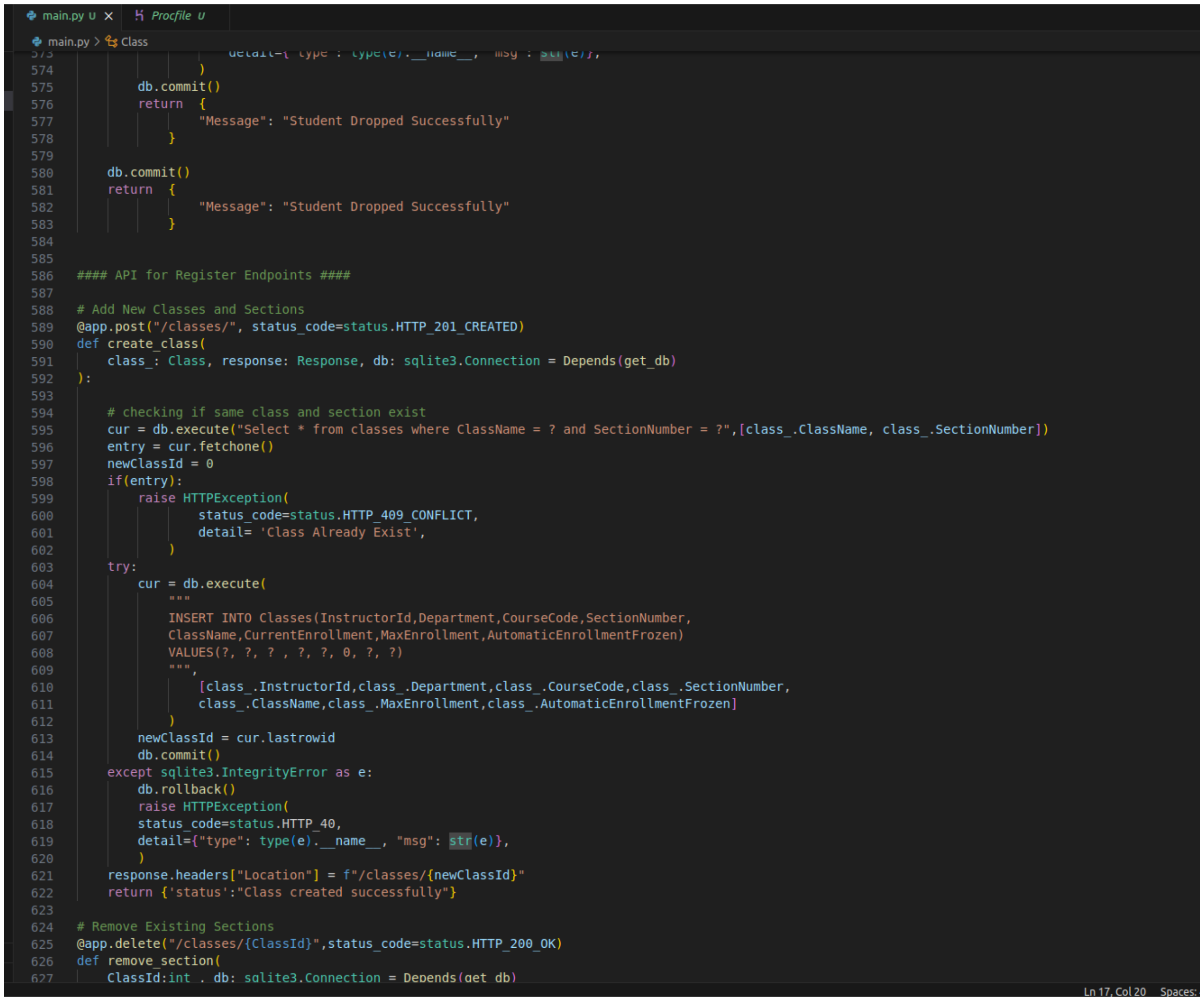
```python
    # checks if class exists
    cur = db.execute("select CurrentEnrollment, MaxEnrollment, AutomaticEnrollmentFrozen, InstructorId from Classes where ClassId = ?",[ClassId])

    entries = cur.fetchone()
    if(not entries):
        raise HTTPException(status_code=404, detail="Class does not exist")
    currentEnrollment, maxEnrollment, automaticEnrollmentFrozen, instructorId = entries

    # checks if InstructorId is valid
    if(InstructorId != instructorId):
        raise HTTPException(status_code=403, detail="You are not the instructor of this class") # Forbidden srtatus code

    # Checks if student was enrolled to the course
    cur = db.execute(
        """
        Select * from Enrollments where ClassId = ? and  StudentId = ? and dropped = 0
        """,
        [ClassId, StudentId])
    entries = cur.fetchone()
    if(not entries):
        raise HTTPException(status_code=404, detail="Student is not enrolled in this class") #Not Found

    # drops the course
    try:
        db.execute("""
                    UPDATE Enrollments SET dropped = 1 where ClassId = ? and StudentId = ?
                    """,
                    [ClassId,StudentId])
        db.execute("""
                    UPDATE Classes SET CurrentEnrollment = ? where ClassId = ?
                    """,
                    [(currentEnrollment-1),ClassId])
        db.commit()
    except sqlite3.IntegrityError as e:
        db.rollback()
        raise HTTPException(
            status_code=status.HTTP_409_CONFLICT,
            detail={"type": type(e).__name__, "msg": str(e)},
        )

    cur = db.execute("Select * from WaitingLists where ClassId = ? ORDER BY WaitingListPos ASC",[ClassId])
    entry = cur.fetchone()
    if (not automaticEnrollmentFrozen and (currentEnrollment-1)<maxEnrollment and entry):

        # Enroll student who is on top of the waitlist
                # Checks if student was enrolled to that course earlier
        cur = db.execute("Select * from Enrollments where ClassId = ? and  StudentId = ?",[ClassId, entry['StudentId']])
        enrollment_entry = cur.fetchone()
        if(enrollment_entry):
            try:
                cur = db.execute("UPDATE Enrollments SET dropped = 0 where ClassId = ? and StudentId = ?",[ClassId, entry['StudentId']])
                db.execute("""
                    UPDATE Classes SET CurrentEnrollment = ? where ClassId = ?
                    """
```

It then updates the current enrollment of the class by 1 and deletes the student from the waitlist. If there is an error status code 409 will be shown. If the student was never enrolled, a new entry is created for the student in the enrollments table with the date and time. It then deletes the student from the waitlist and updates the current enrollment of the class. If there is an error, status code 409 will be shown. After enrolling a student from the waitlist, the positions of the students in the waitlist are changed by the date added. If error status code 409 will be shown.



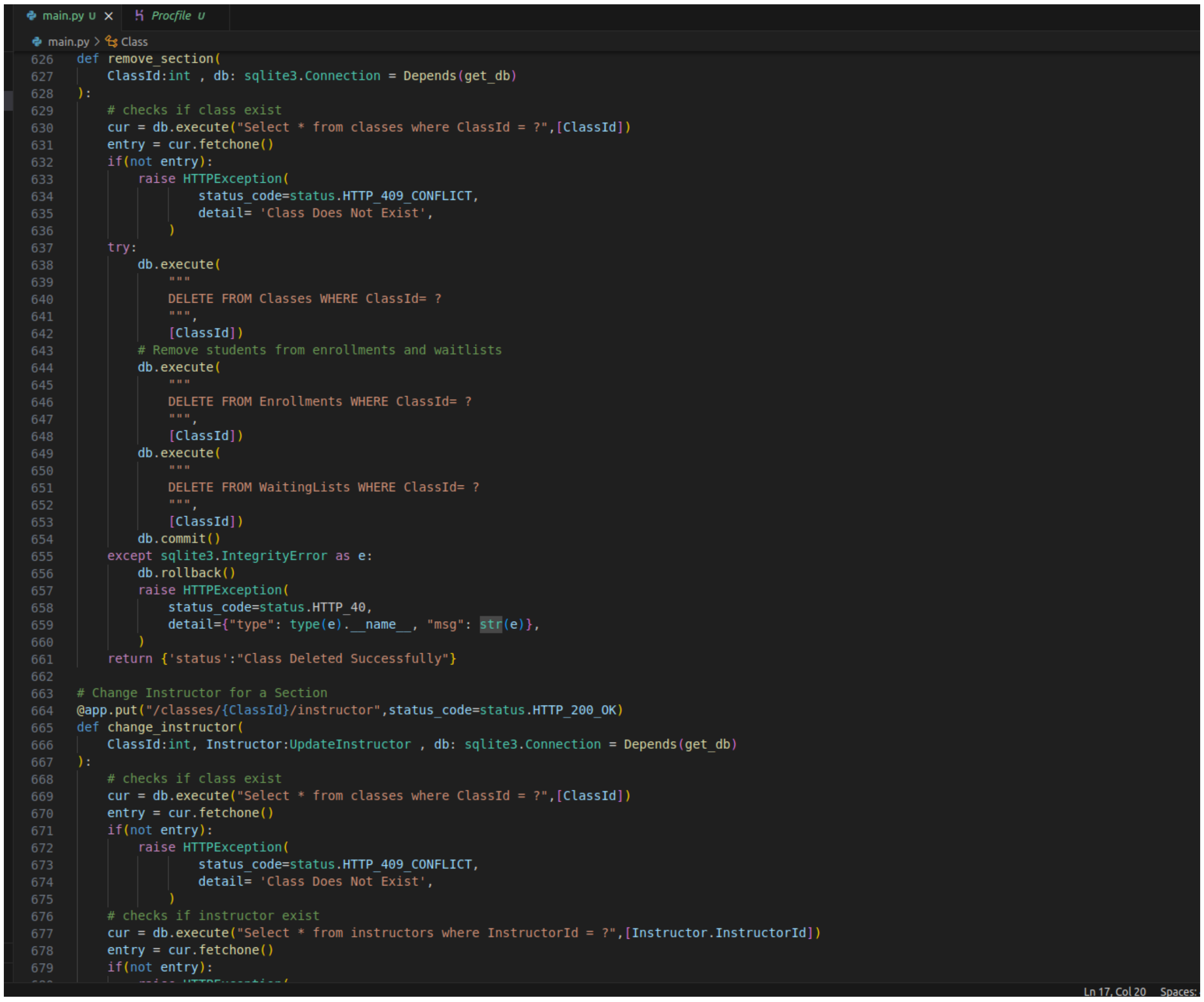
```python
                        UPDATE Classes SET CurrentEnrollment = ? where ClassId = ?
                        """,
                        [(currentEnrollment),ClassId])
                db.execute("""
                            DELETE FROM WaitingLists WHERE StudentId = ? and ClassId= ?
                            """,
                            [entry['StudentId'],ClassId])

                db.commit()
            except sqlite3.IntegrityError as e:
                raise HTTPException(
                status_code=status.HTTP_409_CONFLICT,
                detail={"type": type(e).__name__, "msg": str(e)},
            )
        else: # adding him to the enrollments
            try:
                cur = db.execute(
                """
                INSERT INTO enrollments(StudentId,ClassID,EnrollmentDate)
                VALUES(?, ?, datetime('now'))
                """,
                [entry['StudentId'], ClassId],
            )
                db.execute("""
                            DELETE FROM WaitingLists WHERE StudentId = ? and ClassId= ?
                            """,
                            [entry['StudentId'],ClassId])
                db.execute("""
                        UPDATE Classes SET CurrentEnrollment = ? where ClassId = ?
                        """,
                        [(currentEnrollment),ClassId])
                db.commit()
            except sqlite3.IntegrityError as e:
                db.rollback()
                raise HTTPException(
                    status_code=status.HTTP_409_CONFLICT,
                    detail={"type": type(e).__name__, "msg": str(e)},
                )
        # updating waitlist positions
        cur = db.execute("Select * from WaitingLists where ClassId = ? ORDER BY DateAdded ASC",[ClassId])
        entries = cur.fetchall()
        for entry in entries:
            try:
                db.execute("""
                            UPDATE WaitingLists SET WaitingListPos = ? where ClassId = ? and WaitListId = ?
                            """,
                            [(entry['WaitingListPos']-1),ClassId,entry['WaitListId']])
                db.commit()
            except sqlite3.IntegrityError as e:
                db.rollback()
                raise HTTPException(
                    status_code=status.HTTP_409_CONFLICT,
                    detail={"type": type(e).__name__, "msg": str(e)},
                )
```

We create a POST endpoint for /classes/. Create classes has parameters Class, response, and db connection into the database. It first checks to see if the same class and section number exists. If it does exist, return error code 409 to the user. Otherwise, it will insert the class into the database and the class is created successfully.



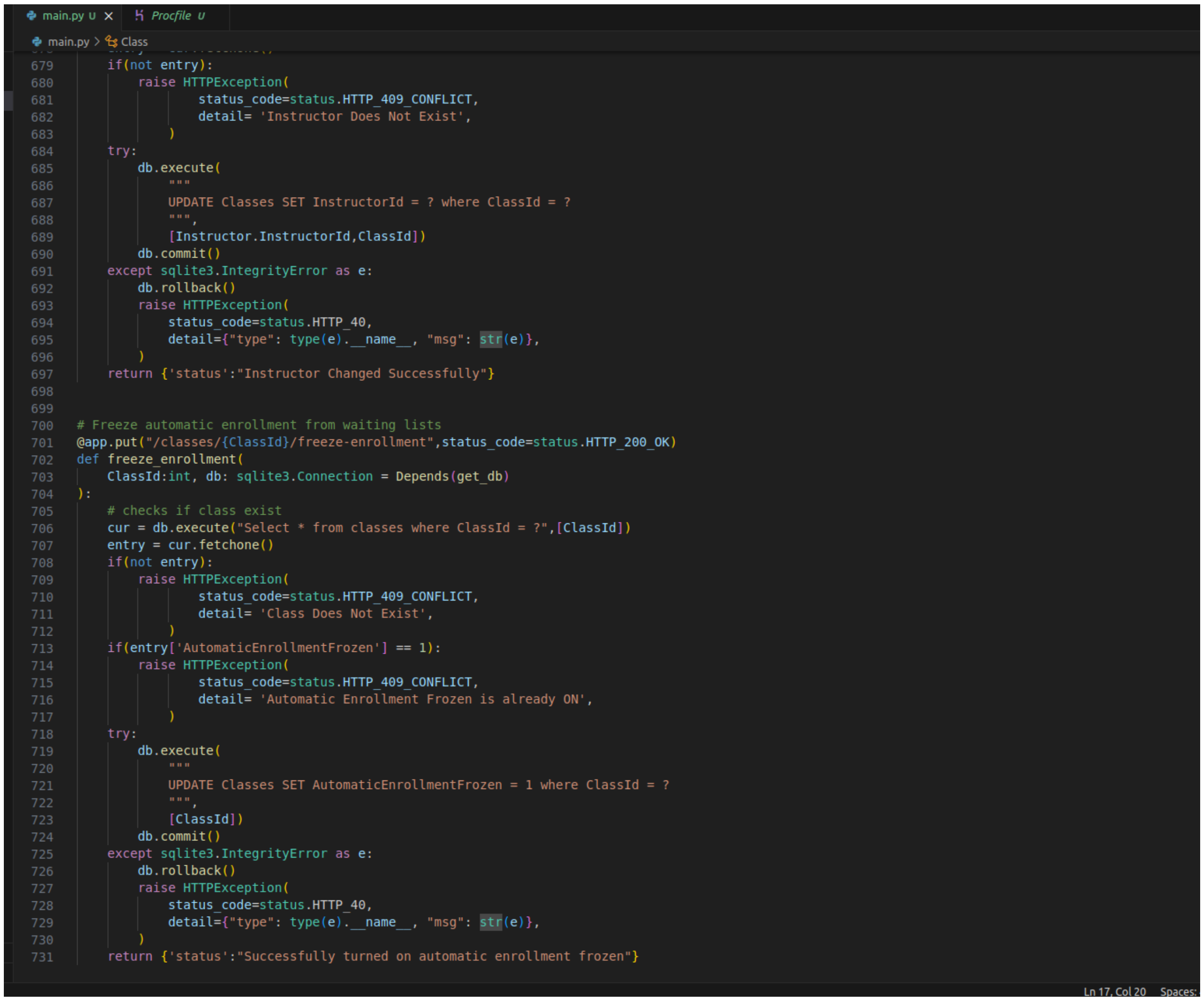
```
                    detail={"type": type(e).__name__, "msg": str(e)},
                )
        db.commit()
        return  {
                "Message": "Student Dropped Successfully"
            }

    db.commit()
    return  {
                "Message": "Student Dropped Successfully"
            }


#### API for Register Endpoints ####

# Add New Classes and Sections
@app.post("/classes/", status_code=status.HTTP_201_CREATED)
def create_class(
    class_: Class, response: Response, db: sqlite3.Connection = Depends(get_db)
):

    # checking if same class and section exist
    cur = db.execute("Select * from classes where ClassName = ? and SectionNumber = ?",[class_.ClassName, class_.SectionNumber])
    entry = cur.fetchone()
    newClassId = 0
    if(entry):
        raise HTTPException(
                status_code=status.HTTP_409_CONFLICT,
                detail= 'Class Already Exist',
            )
    try:
        cur = db.execute(
            """
            INSERT INTO Classes(InstructorId,Department,CourseCode,SectionNumber,
            ClassName,CurrentEnrollment,MaxEnrollment,AutomaticEnrollmentFrozen)
            VALUES(?, ?, ? , ?, ?, 0, ?, ?)
            """,
                [class_.InstructorId,class_.Department,class_.CourseCode,class_.SectionNumber,
                class_.ClassName,class_.MaxEnrollment,class_.AutomaticEnrollmentFrozen]
            )
        newClassId = cur.lastrowid
        db.commit()
    except sqlite3.IntegrityError as e:
        db.rollback()
        raise HTTPException(
        status_code=status.HTTP_40,
        detail={"type": type(e).__name__, "msg": str(e)},
        )
    response.headers["Location"] = f"/classes/{newClassId}"
    return {'status':"Class created successfully"}

# Remove Existing Sections
@app.delete("/classes/{ClassId}",status_code=status.HTTP_200_OK)
def remove_section(
    ClassId:int , db: sqlite3.Connection = Depends(get_db)
```

It checks to make sure that the class exists, otherwise it returns error code 409. It will then delete data based on the ClassId given, removing students from enrollment and waiting lists if they exist. The endpoint of changing instructors for a section checks the class id and the instructor id. It verifies to see if the class and instructor do exist in the database. If they do not exist, they return error code 409.



```python
def remove_section(
    ClassId:int , db: sqlite3.Connection = Depends(get_db)
):
    # checks if class exist
    cur = db.execute("Select * from classes where ClassId = ?",[ClassId])
    entry = cur.fetchone()
    if(not entry):
        raise HTTPException(
                status_code=status.HTTP_409_CONFLICT,
                detail= 'Class Does Not Exist',
            )
    try:
        db.execute(
            """
            DELETE FROM Classes WHERE ClassId= ?
            """,
            [ClassId])
        # Remove students from enrollments and waitlists
        db.execute(
            """
            DELETE FROM Enrollments WHERE ClassId= ?
            """,
            [ClassId])
        db.execute(
            """
            DELETE FROM WaitingLists WHERE ClassId= ?
            """,
            [ClassId])
        db.commit()
    except sqlite3.IntegrityError as e:
        db.rollback()
        raise HTTPException(
            status_code=status.HTTP_40,
            detail={"type": type(e).__name__, "msg": str(e)},
        )
    return {'status':"Class Deleted Successfully"}

# Change Instructor for a Section
@app.put("/classes/{ClassId}/instructor",status_code=status.HTTP_200_OK)
def change_instructor(
    ClassId:int, Instructor:UpdateInstructor , db: sqlite3.Connection = Depends(get_db)
):
    # checks if class exist
    cur = db.execute("Select * from classes where ClassId = ?",[ClassId])
    entry = cur.fetchone()
    if(not entry):
        raise HTTPException(
                status_code=status.HTTP_409_CONFLICT,
                detail= 'Class Does Not Exist',
            )
    # checks if instructor exist
    cur = db.execute("Select * from instructors where InstructorId = ?",[Instructor.InstructorId])
    entry = cur.fetchone()
    if(not entry):
```

The change instructors endpoint will update the class' id with the currently input instructor id, where the previous instructor id will be overwritten. The freeze enrollment endpoint has parameters class id and a connection to the database. It checks to make sure that the class exists, otherwise it returns error code 409. If it does exist, the class is frozen so that automatic enrollment from waiting lists is disabled, and it returns its status.

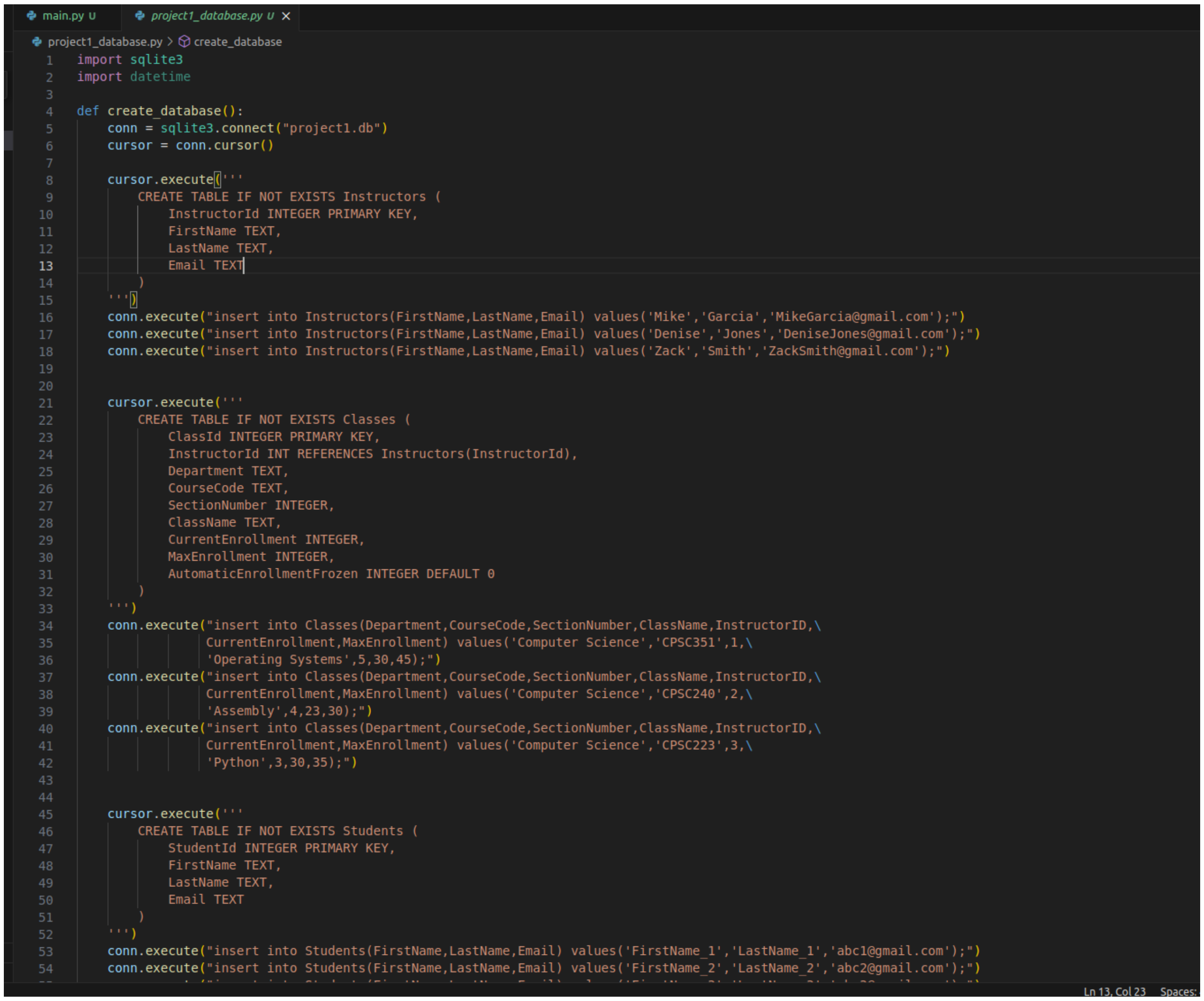


```python
    if(not entry):
        raise HTTPException(
                status_code=status.HTTP_409_CONFLICT,
                detail= 'Instructor Does Not Exist',
            )
    try:
        db.execute(
            """
            UPDATE Classes SET InstructorId = ? where ClassId = ?
            """,
            [Instructor.InstructorId,ClassId])
        db.commit()
    except sqlite3.IntegrityError as e:
        db.rollback()
        raise HTTPException(
            status_code=status.HTTP_40,
            detail={"type": type(e).__name__, "msg": str(e)},
        )
    return {'status':"Instructor Changed Successfully"}


# Freeze automatic enrollment from waiting lists
@app.put("/classes/{ClassId}/freeze-enrollment",status_code=status.HTTP_200_OK)
def freeze_enrollment(
    ClassId:int, db: sqlite3.Connection = Depends(get_db)
):
    # checks if class exist
    cur = db.execute("Select * from classes where ClassId = ?",[ClassId])
    entry = cur.fetchone()
    if(not entry):
        raise HTTPException(
                status_code=status.HTTP_409_CONFLICT,
                detail= 'Class Does Not Exist',
            )
    if(entry['AutomaticEnrollmentFrozen'] == 1):
        raise HTTPException(
                status_code=status.HTTP_409_CONFLICT,
                detail= 'Automatic Enrollment Frozen is already ON',
            )
    try:
        db.execute(
            """
            UPDATE Classes SET AutomaticEnrollmentFrozen = 1 where ClassId = ?
            """,
            [ClassId])
        db.commit()
    except sqlite3.IntegrityError as e:
        db.rollback()
        raise HTTPException(
            status_code=status.HTTP_40,
            detail={"type": type(e).__name__, "msg": str(e)},
        )
    return {'status':"Successfully turned on automatic enrollment frozen"}
```

Project1_database.py

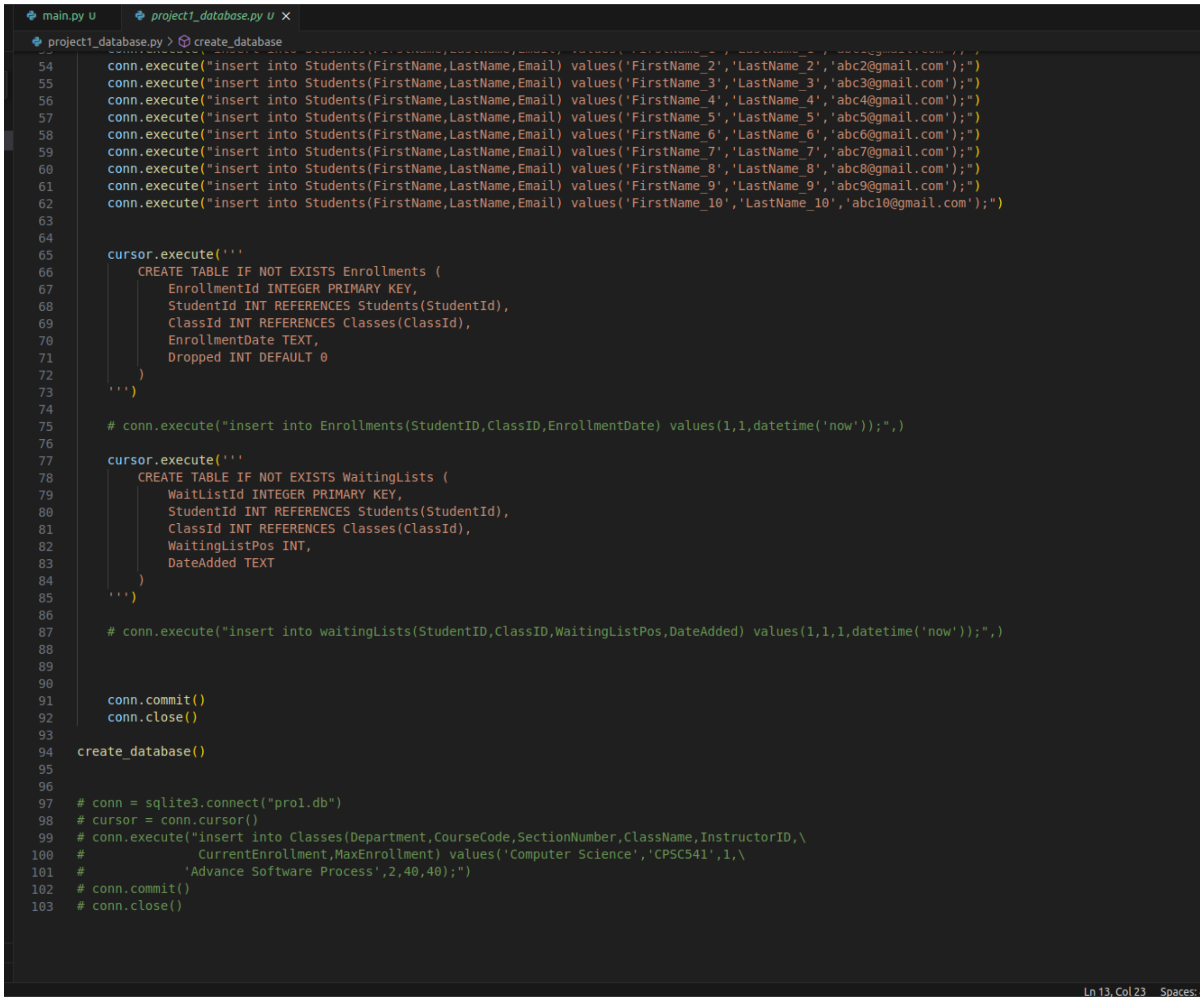
The following screenshot is where the database is initially being created. Here we import sqlite3 to allow interactions with the database. First we set up a connection to the sqlite3 database and call it "project1.db." We set up a cursor instance so that we can execute SQLite statements. We

then create an Instructors table including an instructor id as its primary key, first and last name, and email. We then insert data into the Instructors table.
A Classes table is created as well, including a class id as its primary key, an instructor id that is referenced from the instructors table, the department, course code, section number, class name, a number of students enrolled in the class, the max number of students allowed to enroll in the class, and the class's automatic enrollment. Data is then input into the Classes table.
We create a Students table, where student id is the primary key, lead by the first and last name, and the student's email. We input data into the Students table too.



```python
import sqlite3
import datetime

def create_database():
    conn = sqlite3.connect("project1.db")
    cursor = conn.cursor()

    cursor.execute('''
        CREATE TABLE IF NOT EXISTS Instructors (
            InstructorId INTEGER PRIMARY KEY,
            FirstName TEXT,
            LastName TEXT,
            Email TEXT
        )
    ''')
    conn.execute("insert into Instructors(FirstName,LastName,Email) values('Mike','Garcia','MikeGarcia@gmail.com');")
    conn.execute("insert into Instructors(FirstName,LastName,Email) values('Denise','Jones','DeniseJones@gmail.com');")
    conn.execute("insert into Instructors(FirstName,LastName,Email) values('Zack','Smith','ZackSmith@gmail.com');")


    cursor.execute('''
        CREATE TABLE IF NOT EXISTS Classes (
            ClassId INTEGER PRIMARY KEY,
            InstructorId INT REFERENCES Instructors(InstructorId),
            Department TEXT,
            CourseCode TEXT,
            SectionNumber INTEGER,
            ClassName TEXT,
            CurrentEnrollment INTEGER,
            MaxEnrollment INTEGER,
            AutomaticEnrollmentFrozen INTEGER DEFAULT 0
        )
    ''')
    conn.execute("insert into Classes(Department,CourseCode,SectionNumber,ClassName,InstructorID,\
                CurrentEnrollment,MaxEnrollment) values('Computer Science','CPSC351',1,\
                'Operating Systems',5,30,45);")
    conn.execute("insert into Classes(Department,CourseCode,SectionNumber,ClassName,InstructorID,\
                CurrentEnrollment,MaxEnrollment) values('Computer Science','CPSC240',2,\
                'Assembly',4,23,30);")
    conn.execute("insert into Classes(Department,CourseCode,SectionNumber,ClassName,InstructorID,\
                CurrentEnrollment,MaxEnrollment) values('Computer Science','CPSC223',3,\
                'Python',3,30,35);")


    cursor.execute('''
        CREATE TABLE IF NOT EXISTS Students (
            StudentId INTEGER PRIMARY KEY,
            FirstName TEXT,
            LastName TEXT,
            Email TEXT
        )
    ''')
    conn.execute("insert into Students(FirstName,LastName,Email) values('FirstName_1','LastName_1','abc1@gmail.com');")
    conn.execute("insert into Students(FirstName,LastName,Email) values('FirstName_2','LastName_2','abc2@gmail.com');")
```

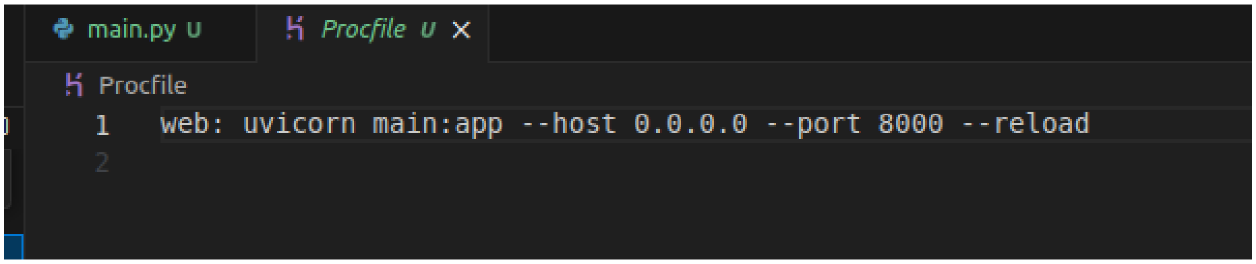
We then create an Enrollments table that is for every different instance of student that enrolls into a specific class. It uses an enrollment id as its primary key, uses the student id and class id from the Students and Classes databases respectively, sets up an enrollment date, and a status to show if a student is dropped or not.
A Waiting List table is created. It uses a waitlist id as its primary key, a student id from the Student database, a class id from the Classes database, a waiting list position, and whenever the waiting list was established.
Finally, the connection is committed and closed, then the database is created.



```python
    conn.execute("insert into Students(FirstName,LastName,Email) values('FirstName_2','LastName_2','abc2@gmail.com');")
    conn.execute("insert into Students(FirstName,LastName,Email) values('FirstName_3','LastName_3','abc3@gmail.com');")
    conn.execute("insert into Students(FirstName,LastName,Email) values('FirstName_4','LastName_4','abc4@gmail.com');")
    conn.execute("insert into Students(FirstName,LastName,Email) values('FirstName_5','LastName_5','abc5@gmail.com');")
    conn.execute("insert into Students(FirstName,LastName,Email) values('FirstName_6','LastName_6','abc6@gmail.com');")
    conn.execute("insert into Students(FirstName,LastName,Email) values('FirstName_7','LastName_7','abc7@gmail.com');")
    conn.execute("insert into Students(FirstName,LastName,Email) values('FirstName_8','LastName_8','abc8@gmail.com');")
    conn.execute("insert into Students(FirstName,LastName,Email) values('FirstName_9','LastName_9','abc9@gmail.com');")
    conn.execute("insert into Students(FirstName,LastName,Email) values('FirstName_10','LastName_10','abc10@gmail.com');")


    cursor.execute('''
        CREATE TABLE IF NOT EXISTS Enrollments (
            EnrollmentId INTEGER PRIMARY KEY,
            StudentId INT REFERENCES Students(StudentId),
            ClassId INT REFERENCES Classes(ClassId),
            EnrollmentDate TEXT,
            Dropped INT DEFAULT 0
        )
    ''')

    # conn.execute("insert into Enrollments(StudentID,ClassID,EnrollmentDate) values(1,1,datetime('now'));",)

    cursor.execute('''
        CREATE TABLE IF NOT EXISTS WaitingLists (
            WaitListId INTEGER PRIMARY KEY,
            StudentId INT REFERENCES Students(StudentId),
            ClassId INT REFERENCES Classes(ClassId),
            WaitingListPos INT,
            DateAdded TEXT
        )
    ''')

    # conn.execute("insert into waitingLists(StudentID,ClassID,WaitingListPos,DateAdded) values(1,1,1,datetime('now'));",)



    conn.commit()
    conn.close()

create_database()


# conn = sqlite3.connect("pro1.db")
# cursor = conn.cursor()
# conn.execute("insert into Classes(Department,CourseCode,SectionNumber,ClassName,InstructorID,\
#               CurrentEnrollment,MaxEnrollment) values('Computer Science','CPSC541',1,\
#             'Advance Software Process',2,40,40);")
# conn.commit()
# conn.close()
```

Procfile

The procfile holds the information necessary for usage with foreman to properly start up a FastAPI web server with uvicorn.



```
web: uvicorn main:app --host 0.0.0.0 --port 8000 --reload
```