

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

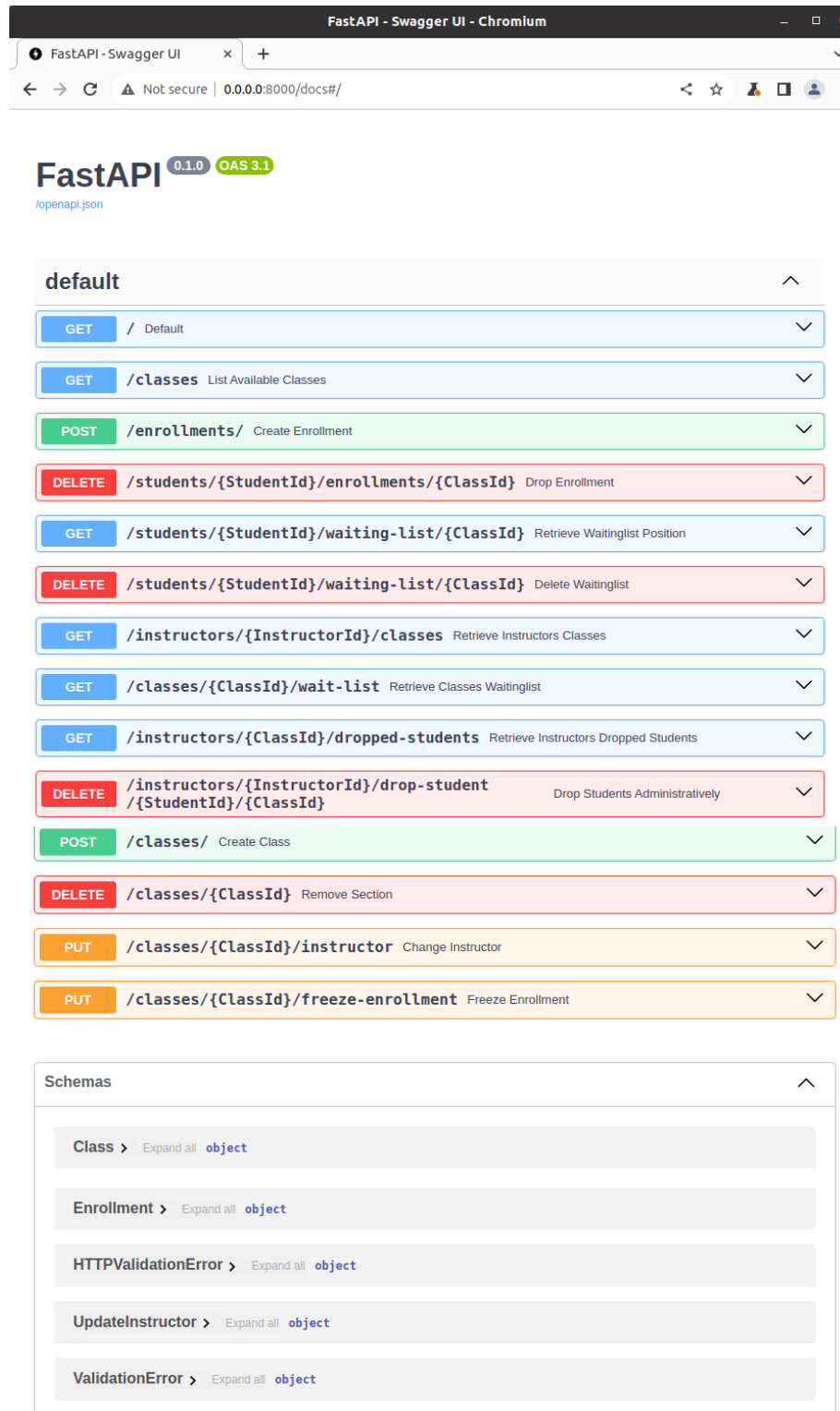
We used the FastAPI framework, Swagger documentation framework, and Foreman to develop our project in Python. Additionally, our database code makes use of the Python Standard Library's sqlite3 module.

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.



GET
/classes
List Available Classes

Parameters
Cancel

No parameters

Execute
Clear

Responses

Curl

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

Request URL

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

Code
Details

200

Response body

```
{
  "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",

```

Download

Response headers

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

Responses

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.

```

1  import contextlib
2  import sqlite3
3  from typing import Optional
4  from fastapi import FastAPI, Depends, Response, HTTPException, status
5  from fastapi.responses import RedirectResponse
6  from pydantic import BaseModel
7
8  class Enrollment(BaseModel):
9      StudentId: int
10     ClassId: int
11
12     class Class(BaseModel):
13         InstructorId: int
14         Department: str
15         CourseCode: str
16         SectionNumber: int
17         ClassName: str
18         MaxEnrollment: int = 40
19         AutomaticEnrollmentFrozen: int = 0
20
21     class UpdateInstructor(BaseModel):
22         InstructorId: int
23
24     app = FastAPI()
25
26
27     def get_db():
28         with contextlib.closing(sqlite3.connect("project1.db")) as db:
29             db.row_factory = sqlite3.Row
30             yield db
31
32     @app.get("/", status_code=status.HTTP_308_PERMANENT_REDIRECT)
33     def default():
34         return RedirectResponse("/docs")
35
36     """
37     API for Students Endpoints
38     """
39
40     # It lists the available classes to students
41     @app.get("/classes", status_code=status.HTTP_200_OK)
42     def list_available_classes(db: sqlite3.Connection = Depends(get_db)):
43         classes = db.execute("SELECT * FROM classes where CurrentEnrollment < MaxEnrollment")
44         if not classes:
45             raise HTTPException(
46                 status_code=status.HTTP_404_NOT_FOUND, detail="Classes not found"
47             )
48         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.

```
main.py > Class
54 # To enroll in a class
55 @app.post("/enrollments/", status_code=status.HTTP_201_CREATED)
56 def create_enrollment(
57     enrollment: Enrollment, response: Response, db: sqlite3.Connection = Depends(get_db)
58 ):
59     cur = db.execute("select CurrentEnrollment, MaxEnrollment, AutomaticEnrollmentFrozen from Classes where ClassId = ?",[enrollment.ClassId])
60     # checking if class exists
61     entry = cur.fetchone()
62     if(not entry):
63         raise HTTPException(
64             status_code=status.HTTP_409_CONFLICT,
65             detail= 'The Class Does Not Exist',
66         )
67     currentEnrollment, maxEnrollment, automaticEnrollmentFrozen = entry
68     # checking if student exist
69     cur = db.execute("Select * from Students where StudentId = ?",[enrollment.StudentId])
70     entry = cur.fetchone()
71     if(not entry):
72         raise HTTPException(
73             status_code=status.HTTP_409_CONFLICT,
74             detail= 'Student Does Not Exist',
75         )
76     # Checking if enrollment is possible
77     if(automaticEnrollmentFrozen==1):
78         raise HTTPException(
79             status_code=status.HTTP_409_CONFLICT,
80             detail= 'Enrollment is closed',
81         )
82
83     # Checking if student is enrolled in a course
84     cur = db.execute("Select * from Enrollments where ClassId = ? and StudentId = ? and dropped = 0",[enrollment.ClassId, enrollment.StudentId])
85     sameClasses = cur.fetchall()
86     if(sameClasses):
87         raise HTTPException(status_code=409, detail="You are already enrolled") #HTTP status code 409: "Conflict."
88
89     # Checking if Class is full then adding student to waitlist
90     if(currentEnrollment >= maxEnrollment):
91
92         # Checks if student is already on waitlist
93         cur = db.execute("Select * from WaitingLists where ClassId = ? and StudentId = ?",[enrollment.ClassId, enrollment.StudentId])
94         alreadyOnWaitlist = cur.fetchall()
95         if(alreadyOnWaitlist):
96             raise HTTPException(status_code=409, detail="You are already on waitlist") #HTTP status code 409, which stands for "Conflict."
97
98         # Checks that student is not on more than 3 waitlist (not checked)
99         cur = db.execute("Select * from Waitinglists where StudentId = ?",[enrollment.StudentId])
100         moreThanThree = cur.fetchall()
101         if(len(moreThanThree)>3):
102             raise HTTPException(status_code=409, detail="Class is full, already on three waitlists.") #HTTP status code 409, which stands for "Conflict."
103
```

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.

```
main.py > Class
104 # Adding to the waitlist if waitlist is not full
105 cur = db.execute("Select * from WaitingLists where ClassId = ?",[enrollment.ClassId])
106 entries = cur.fetchall()
107 if(len(entries)>=15):
108     raise HTTPException(status_code=403, detail="Waiting List if full for this class") # Forbidden
109 waitListPosition = len(entries)+1
110 e = dict(enrollment)
111 try:
112     cur = db.execute(
113         """
114         INSERT INTO WaitingLists(StudentID,ClassID,WaitingListPos,DateAdded)
115         VALUES(?, ?, ?, datetime('now'))
116         """
117         ,[enrollment.StudentId,enrollment.ClassId,waitListPosition]
118     )
119     db.commit()
120 except sqlite3.IntegrityError as e:
121     raise HTTPException(
122         status_code=status.HTTP_409_CONFLICT,
123         detail={"type": type(e).__name__, "msg": str(e)},
124     )
125 e["id"] = cur.lastrowid
126 response.headers["Location"] = f"/WaitingLists/{e['id']}"
127 message = f"Class is full you have been placed on waitlist position {waitListPosition}"
128 # Checks if student was enrolled earlier
129 raise HTTPException(status_code=400, detail=message)
130
131 # Checks if student was enrolled earlier
132 cur = db.execute("Select * from Enrollments where ClassId = ? and StudentId = ?",[enrollment.ClassId, enrollment.StudentId])
133 entry = cur.fetchone()
134 if(entry):
135     try:
136         db.execute("""
137             UPDATE Enrollments SET dropped = 0 where ClassId = ? and StudentId = ?
138             """
139             ,[enrollment.ClassId,enrollment.StudentId])
140         db.execute("""
141             UPDATE Classes SET CurrentEnrollment = ? where ClassId = ?
142             """
143             ,[(currentEnrollment+1),enrollment.ClassId])
144         db.commit()
145     except sqlite3.IntegrityError as e:
146         raise HTTPException(
147             status_code=status.HTTP_409_CONFLICT,
148             detail={"type": type(e).__name__, "msg": str(e)},
149         )
150     return {"Success":"Enrolled"}
151
152 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.

```
main.py x Procfile u
main.py > Class
151
152 else:
153     # Register Students if class is not full
154     e = dict(enrollment)
155     try:
156         cur = db.execute(
157             """
158             INSERT INTO enrollments(StudentId,ClassId,EnrollmentDate)
159             VALUES(:StudentId, :ClassId, datetime('now'))
160             """
161             , e,
162         )
163         # db.commit()
164         # Updating currentEnrollment
165         cur = db.execute("UPDATE Classes SET currentEnrollment = ? where ClassId = ?",[(currentEnrollment+1),enrollment.ClassId])
166         db.commit()
167     except sqlite3.IntegrityError as e:
168         raise HTTPException(
169             status_code=status.HTTP_409_CONFLICT,
170             detail={"type": type(e).__name__, "msg": str(e)},
171         )
172     e["id"] = cur.lastrowid
173     response.headers["Location"] = f"/enrollments/{e['id']}"
174     return {"Success":e}
175
176 # Delete enrollment of student
177 @app.delete("/students/{StudentId}/enrollments/{ClassId}",status_code=status.HTTP_200_OK)
178 def drop_enrollment(
179     StudentId:int, ClassId:int , db: sqlite3.Connection = Depends(get_db)
180 ):
181     cur = db.execute("select CurrentEnrollment, MaxEnrollment, AutomaticEnrollmentFrozen from Classes where ClassId = ?",[ClassId])
182     entries = cur.fetchone()
183     # check if class exists
184     if(not entries):
185         raise HTTPException(status_code=404, detail="Class does not exist")
186     currentEnrollment, maxEnrollment, automaticEnrollmentFrozen = entries
187
188     # checks if student exist
189     cur = db.execute("Select * from Students where StudentId = ?",[StudentId])
190     entry = cur.fetchone()
191     if(not entry):
192         raise HTTPException(
193             status_code=status.HTTP_409_CONFLICT,
194             detail= 'Student Does Not Exist',
195         )
196
197     # Checks if student was enrolled to the course
198     cur = db.execute("Select * from Enrollments where ClassId = ? and StudentId = ? and dropped = 0",[ClassId, StudentId])
199     entries = cur.fetchone()
200     if(not entries):
201         raise HTTPException(status_code=404, detail="You are not enrolled in this course") #student enrollement not found
202     # 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 x Procfile U
main.py > Class
204 # drops the course
205 try:
206     db.execute("""
207         UPDATE Enrollments SET dropped = 1 where ClassId = ? and StudentId = ?
208         """,
209         [ClassId, StudentId])
210     db.execute("""
211         UPDATE Classes SET CurrentEnrollment = ? where ClassId = ?
212         """,
213         [(currentEnrollment-1), ClassId])
214     db.commit()
215 except sqlite3.IntegrityError as e:
216     raise HTTPException(
217         status_code=status.HTTP_409_CONFLICT,
218         detail={"type": type(e).__name__, "msg": str(e)},
219     )
220 db.commit()
221 cur = db.execute("Select * from WaitingLists where ClassId = ? ORDER BY WaitingListPos ASC",[ClassId])
222 entry = cur.fetchone()
223 if (not automaticEnrollmentFrozen and (currentEnrollment-1)<maxEnrollment and entry):
224
225     # Enroll student who is on top of the waitlist
226     # Checks if student was enrolled to that course earlier
227     cur = db.execute("Select * from Enrollments where ClassId = ? and StudentId = ?",[ClassId, entry['StudentId']])
228     enrollment_entry = cur.fetchone()
229     if enrollment_entry:
230         try:
231             cur = db.execute("UPDATE Enrollments SET dropped = 0 where ClassId = ? and StudentId = ?",[ClassId, entry['StudentId']])
232             db.execute("""
233                 UPDATE Classes SET CurrentEnrollment = ? where ClassId = ?
234                 """,
235                 [(currentEnrollment), ClassId])
236             db.execute("""
237                 DELETE FROM WaitingLists WHERE StudentId = ? and ClassId= ?
238                 """,
239                 [entry['StudentId'], ClassId])
240
241             db.commit()
242         except sqlite3.IntegrityError as e:
243             raise HTTPException(
244                 status_code=status.HTTP_409_CONFLICT,
245                 detail={"type": type(e).__name__, "msg": str(e)},
246             )
247     else:
248         try:
249             cur = db.execute(
250                 """
251                 INSERT INTO enrollments(StudentId,ClassId,EnrollmentDate)
252                 VALUES(?, ?, datetime('now'))
253                 """,
254                 [entry['StudentId'], ClassId],
255             )
256             db.execute("""
257                 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 x Procfile U
main.py > Class
256 db.execute("""
257     DELETE FROM WaitingLists WHERE StudentId = ? and ClassId = ?
258     """,
259     [entry['StudentId'],ClassId])
260 db.execute("""
261     UPDATE Classes SET CurrentEnrollment = ? where ClassId = ?
262     """,
263     [(currentEnrollment),ClassId])
264 db.commit()
265 except sqlite3.IntegrityError as e:
266     raise HTTPException(
267         status_code=status.HTTP_409_CONFLICT,
268         detail={"type": type(e).__name__, "msg": str(e)},
269     )
270 # update waitlist positions
271 cur = db.execute("Select * from WaitingLists where ClassId = ? ORDER BY DateAdded ASC",[ClassId])
272 entries = cur.fetchall()
273 for entry in entries:
274     try:
275         db.execute("""
276             UPDATE WaitingLists SET WaitingListPos = ? where ClassId = ? and WaitListId = ?
277             """,
278             [(entry['WaitingListPos']-1),ClassId,entry['WaitListId']])
279         db.commit()
280     except sqlite3.IntegrityError as e:
281         raise HTTPException(
282             status_code=status.HTTP_409_CONFLICT,
283             detail={"type": type(e).__name__, "msg": str(e)},
284         )
285 db.commit()
286 return {
287     "Message": "Successfully dropped"
288 }
289 db.commit()
290 return {
291     "Message": "Successfully dropped"
292 }
293
294
295 # View Waiting List Position
296 @app.get("/students/{StudentId}/waiting-list/{ClassId}",status_code=status.HTTP_200_OK)
297 def retrieve_waitinglist_position(
298     StudentId: int, ClassId: int, db: sqlite3.Connection = Depends(get_db)
299 ):
300     # checks if student exist
301     cur = db.execute("Select * from Students where StudentId = ?",[StudentId])
302     entry = cur.fetchone()
303     if(not entry):
304         raise HTTPException(
305             status_code=status.HTTP_409_CONFLICT,
306             detail= 'Student Does Not Exist',
307         )
308     # checks if class exist
309     cur = db.execute("Select * from classes where ClassId = ?",[ClassId])
310     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.

```
main.py x Procfile U
main.py > Class
309 cur = db.execute("Select * from classes where ClassId = ?",[ClassId])
310 entry = cur.fetchone()
311 if(not entry):
312     raise HTTPException(
313         status_code=status.HTTP_409_CONFLICT,
314         detail= 'Class Does Not Exist',
315     )
316 cur = db.execute("SELECT * FROM WaitingLists WHERE StudentId = ? and ClassId= ?", [StudentId,ClassId])
317 waitingList = cur.fetchone()
318 if not waitingList:
319     raise HTTPException(
320         status_code=status.HTTP_404_NOT_FOUND, detail="Position not found"
321     )
322 return {
323     "data": waitingList,
324     "WaitingListPos":waitingList['WaitingListPos']
325 }
326
327
328 # Remove from Waiting List
329 @app.delete("/students/{StudentId}/waiting-list/{ClassId}",status_code=status.HTTP_200_OK)
330 def delete_waitinglist(
331     StudentId: int, ClassId: int, db: sqlite3.Connection = Depends(get_db)
332 ):
333     # checks if student exist
334     cur = db.execute("Select * from Students where StudentId = ?",[StudentId])
335     entry = cur.fetchone()
336     if(not entry):
337         raise HTTPException(
338             status_code=status.HTTP_409_CONFLICT,
339             detail= 'Student Does Not Exist',
340         )
341     # checks if class exist
342     cur = db.execute("Select * from classes where ClassId = ?",[ClassId])
343     entry = cur.fetchone()
344     if(not entry):
345         raise HTTPException(
346             status_code=status.HTTP_409_CONFLICT,
347             detail= 'Class Does Not Exist',
348         )
349     # Checks if entry exist in waitinglist
350     cur = db.execute("SELECT * FROM WaitingLists WHERE StudentId = ? and ClassId= ?", [StudentId,ClassId])
351     waitingList = cur.fetchone()
352     if not waitingList:
353         raise HTTPException(
354             status_code=status.HTTP_404_NOT_FOUND, detail="Not in Waitlist"
355         )
356     # updates waitlist positions
357     try:
358         cur = db.execute("Select * from WaitingLists where ClassId = ? and WaitListId > ?",[ClassId, waitingList['WaitListId']])
359         entries = cur.fetchall()
360         for entry in entries:
361             db.execute("""
362                 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.

```
main.py x Procfile U
main.py > Class
362         db.execute("UPDATE WaitingLists SET WaitingListPos = ? where ClassId = ? and WaitListId = ?
363         """
364         [(entry['WaitingListPos']-1),ClassId,entry['WaitListId']])
365         db.commit()
366         db.execute("DELETE FROM WaitingLists WHERE StudentId = ? and ClassId = ?", [StudentId,ClassId])
367         db.commit()
368     except sqlite3.IntegrityError as e:
369         db.rollback()
370         raise HTTPException(
371             status_code=status.HTTP_409_CONFLICT,
372             detail={"type": type(e).__name__, "msg": str(e)},
373         )
374     db.commit()
375     return {
376         "Message": "Successfully removed from the Waiting List"
377     }
378
379     """
380 API for Instructors Endpoints
381 """
382
383 # View Current Enrollment for Their Classes
384 @app.get("/instructors/{InstructorId}/classes",status_code=status.HTTP_200_OK)
385 def retrieve_Instructors_Classes(
386     InstructorId: int, db: sqlite3.Connection = Depends(get_db)
387 ):
388     # checks if instructor exist
389     cur = db.execute("Select * from instructors where InstructorId = ?",[InstructorId])
390     entry = cur.fetchone()
391     if(not entry):
392         raise HTTPException(
393             status_code=status.HTTP_409_CONFLICT,
394             detail= 'Instructor Does Not Exist',
395         )
396
397     cur = db.execute("SELECT classname,currentenrollment FROM Classes WHERE InstructorId = ?", [InstructorId])
398     instructorClasses = cur.fetchall()
399     if not instructorClasses:
400         raise HTTPException(
401             status_code=status.HTTP_404_NOT_FOUND, detail="Instructor does not have any classes"
402         )
403     return {
404         "instructorClasses": instructorClasses
405     }
406
407 # View the current waiting list for the course
408 @app.get("/classes/{ClassId}/wait-list",status_code=status.HTTP_200_OK)
409 def retrieve_Classes_WaitingList(
410     ClassId: int, db: sqlite3.Connection = Depends(get_db)
411 ):
412     # checks if class exist
413     cur = db.execute("Select * from classes where ClassId = ?",[ClassId])
414     entry = cur.fetchone()
415     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.

```
main.py x Procfile U
main.py > Class
414 entry = cur.fetchone()
415 if(not entry):
416     raise HTTPException(
417         status_code=status.HTTP_409_CONFLICT,
418         detail= 'Class Does Not Exist',
419     )
420
421 cur = db.execute("SELECT * FROM WaitingLists WHERE ClassId = ?", [ClassId])
422 classesWaitingList = cur.fetchall()
423 if not classesWaitingList:
424     raise HTTPException(
425         status_code=status.HTTP_404_NOT_FOUND, detail="Waiting List does not exist for this class"
426     )
427 return {
428     "Total Waitlisted Students": len(classesWaitingList),
429     "instructorClassesWaitingList": classesWaitingList
430 }
431
432 # View Students Who Have Dropped the Class
433 @app.get("/instructors/{ClassId}/dropped-students",status_code=status.HTTP_200_OK)
434 def retrieve_instructors_dropped_students(
435     ClassId:int, db: sqlite3.Connection = Depends(get_db)
436 ):
437     # checks if class exist
438     cur = db.execute("Select * from classes where ClassId = ?",[ClassId])
439     entry = cur.fetchone()
440     if(not entry):
441         raise HTTPException(
442             status_code=status.HTTP_409_CONFLICT,
443             detail= 'Class Does Not Exist',
444         )
445     cur = db.execute("SELECT * FROM Students WHERE StudentId in (SELECT StudentId FROM Enrollments WHERE ClassId = ? and Dropped = 1)", [C
446     studentsWhoDropped = cur.fetchall()
447     if not studentsWhoDropped:
448         raise HTTPException(
449             status_code=status.HTTP_404_NOT_FOUND, detail="No students have dropped this class"
450         )
451     return {
452         "Dropped Students": studentsWhoDropped
453     }
454
455 # Drop students administratively
456 @app.delete("/instructors/{InstructorId}/drop-student/{StudentId}/{ClassId}")
457 def drop_students_administratively(
458     InstructorId:int, StudentId:int, ClassId:int, db: sqlite3.Connection = Depends(get_db)
459 ):
460     # checks if instructor exist
461     cur = db.execute("Select * from instructors where InstructorId = ?",[InstructorId])
462     entry = cur.fetchone()
463     if(not entry):
464         raise HTTPException(
465             status_code=status.HTTP_409_CONFLICT,
466             detail= 'Instructor Does Not Exist',
467         )
468
```

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.

```
main.py U x  H Profile U
main.py > Class
468
469 # checks if class exists
470 cur = db.execute("select CurrentEnrollment, MaxEnrollment, AutomaticEnrollmentFrozen, InstructorId from Classes where ClassId = ?",[ClassId])
471
472 entries = cur.fetchone()
473 if(not entries):
474     raise HTTPException(status_code=404, detail="Class does not exist")
475 currentEnrollment, maxEnrollment, automaticEnrollmentFrozen, instructorId = entries
476
477 # checks if InstructorId is valid
478 if(InstructorId != instructorId):
479     raise HTTPException(status_code=403, detail="You are not the instructor of this class") # Forbidden status code
480
481 # Checks if student was enrolled to the course
482 cur = db.execute(
483     """
484     Select * from Enrollments where ClassId = ? and StudentId = ? and dropped = 0
485     """,
486     [ClassId, StudentId])
487 entries = cur.fetchone()
488 if(not entries):
489     raise HTTPException(status_code=404, detail="Student is not enrolled in this class") #Not Found
490
491 # drops the course
492 try:
493     db.execute("""
494         UPDATE Enrollments SET dropped = 1 where ClassId = ? and StudentId = ?
495         """,
496         [ClassId, StudentId])
497     db.execute("""
498         UPDATE Classes SET CurrentEnrollment = ? where ClassId = ?
499         """,
500         [(currentEnrollment-1), ClassId])
501     db.commit()
502 except sqlite3.IntegrityError as e:
503     db.rollback()
504     raise HTTPException(
505         status_code=status.HTTP_409_CONFLICT,
506         detail={"type": type(e).__name__, "msg": str(e)},
507     )
508
509 cur = db.execute("Select * from WaitingLists where ClassId = ? ORDER BY WaitingListPos ASC",[ClassId])
510 entry = cur.fetchone()
511 if (not automaticEnrollmentFrozen and (currentEnrollment-1)<maxEnrollment and entry):
512
513     # Enroll student who is on top of the waitlist
514     # Checks if student was enrolled to that course earlier
515     cur = db.execute("Select * from Enrollments where ClassId = ? and StudentId = ?",[ClassId, entry['StudentId']])
516     enrollment_entry = cur.fetchone()
517     if(enrollment_entry):
518         try:
519             cur = db.execute("UPDATE Enrollments SET dropped = 0 where ClassId = ? and StudentId = ?",[ClassId, entry['StudentId']])
520             db.execute("""
521                 UPDATE Classes SET CurrentEnrollment = ? where ClassId = ?
522                 """,
```

Ln 17, Col 20 Spaces:

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.

```
main.py U X Procfile U
main.py > Class
521         UPDATE Classes SET CurrentEnrollment = ? where ClassId = ?
522         """
523         [(currentEnrollment),ClassId])
524     db.execute("""
525         DELETE FROM WaitingLists WHERE StudentId = ? and ClassId= ?
526         """
527         [entry['StudentId'],ClassId])
528
529     db.commit()
530 except sqlite3.IntegrityError as e:
531     raise HTTPException(
532         status_code=status.HTTP_409_CONFLICT,
533         detail={"type": type(e).__name__, "msg": str(e)},
534     )
535 else: # adding him to the enrollments
536     try:
537         cur = db.execute(
538             """
539             INSERT INTO enrollments(StudentId,ClassId,EnrollmentDate)
540             VALUES(?, ?, datetime('now'))
541             """
542             [entry['StudentId'], ClassId],
543         )
544         db.execute("""
545             DELETE FROM WaitingLists WHERE StudentId = ? and ClassId= ?
546             """
547             [entry['StudentId'],ClassId])
548         db.execute("""
549             UPDATE Classes SET CurrentEnrollment = ? where ClassId = ?
550             """
551             [(currentEnrollment),ClassId])
552         db.commit()
553 except sqlite3.IntegrityError as e:
554     db.rollback()
555     raise HTTPException(
556         status_code=status.HTTP_409_CONFLICT,
557         detail={"type": type(e).__name__, "msg": str(e)},
558     )
559 # updating waitlist positions
560 cur = db.execute("Select * from WaitingLists where ClassId = ? ORDER BY DateAdded ASC",[ClassId])
561 entries = cur.fetchall()
562 for entry in entries:
563     try:
564         db.execute("""
565             UPDATE WaitingLists SET WaitingListPos = ? where ClassId = ? and WaitListId = ?
566             """
567             [(entry['WaitingListPos']-1),ClassId,entry['WaitListId'])
568         db.commit()
569 except sqlite3.IntegrityError as e:
570     db.rollback()
571     raise HTTPException(
572         status_code=status.HTTP_409_CONFLICT,
573         detail={"type": type(e).__name__, "msg": str(e)},
574     )
```

Ln 17, Col 20 Spaces:

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.

```
main.py x Procfile u
main.py > Class
573         detail={"type": type(e).__name__, "msg": str(e)},
574     )
575     db.commit()
576     return {
577         "Message": "Student Dropped Successfully"
578     }
579
580     db.commit()
581     return {
582         "Message": "Student Dropped Successfully"
583     }
584
585
586 ##### API for Register Endpoints #####
587
588 # Add New Classes and Sections
589 @app.post("/classes/", status_code=status.HTTP_201_CREATED)
590 def create_class(
591     class_: Class, response: Response, db: sqlite3.Connection = Depends(get_db)
592 ):
593
594     # checking if same class and section exist
595     cur = db.execute("Select * from classes where ClassName = ? and SectionNumber = ?",[class_.ClassName, class_.SectionNumber])
596     entry = cur.fetchone()
597     newClassId = 0
598     if(entry):
599         raise HTTPException(
600             status_code=status.HTTP_409_CONFLICT,
601             detail= 'Class Already Exist',
602         )
603     try:
604         cur = db.execute(
605             """
606             INSERT INTO Classes(InstructorId,Department,CourseCode,SectionNumber,
607             ClassName,CurrentEnrollment,MaxEnrollment,AutomaticEnrollmentFrozen)
608             VALUES(?, ?, ?, ?, ?, 0, ?, ?)
609             """
610             ,
611             [class_.InstructorId,class_.Department,class_.CourseCode,class_.SectionNumber,
612             class_.ClassName,class_.MaxEnrollment,class_.AutomaticEnrollmentFrozen]
613         )
614         newClassId = cur.lastrowid
615         db.commit()
616     except sqlite3.IntegrityError as e:
617         db.rollback()
618         raise HTTPException(
619             status_code=status.HTTP_409,
620             detail={"type": type(e).__name__, "msg": str(e)},
621         )
622     response.headers["Location"] = f"/classes/{newClassId}"
623     return {'status':"Class created successfully"}
624
625 # Remove Existing Sections
626 @app.delete("/classes/{ClassId}",status_code=status.HTTP_200_OK)
627 def remove_section(
628     ClassId:int , db: sqlite3.Connection = Depends(get_db)
629 ):
```

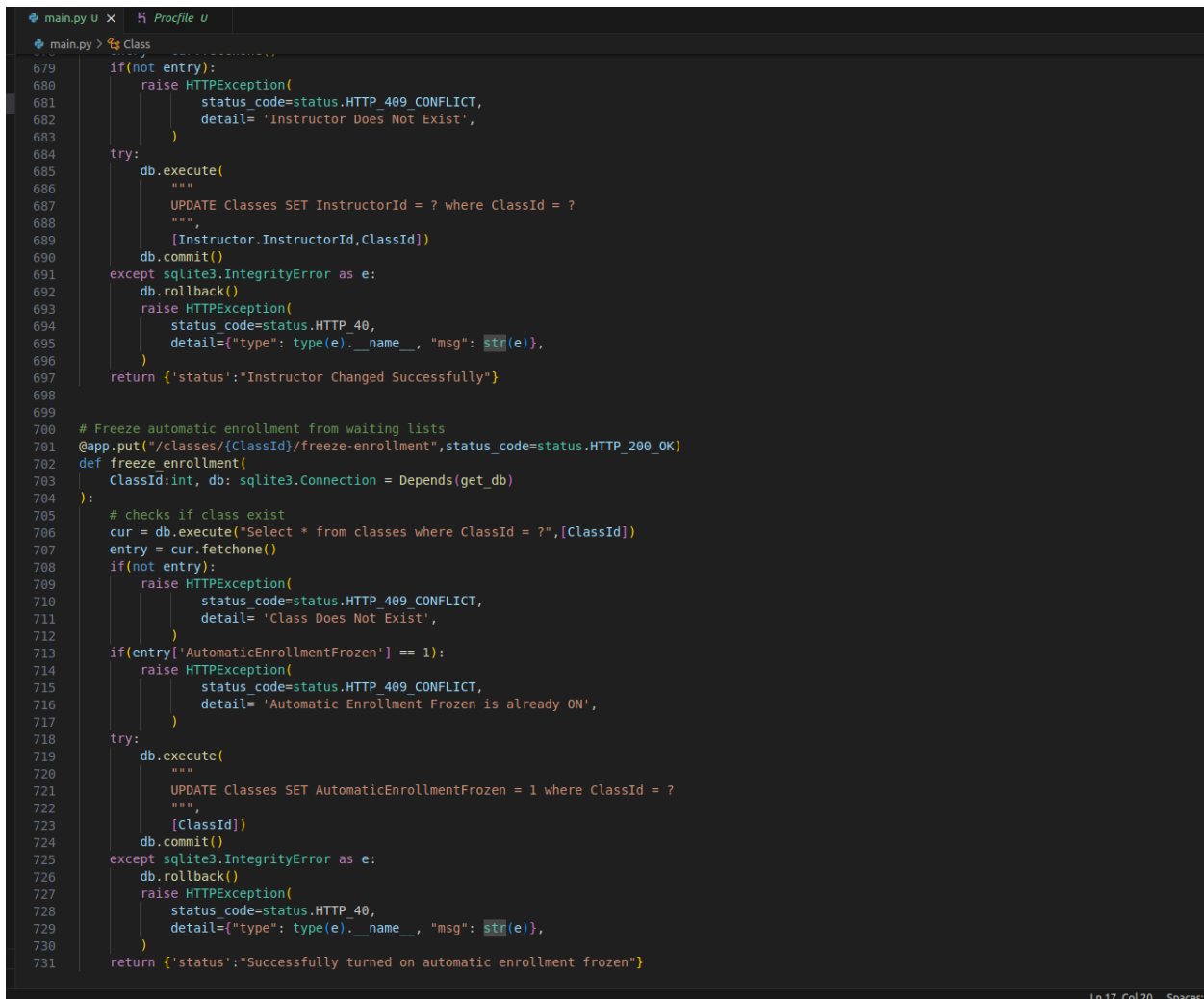
Ln 17, Col 20 Spaces

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.

```
main.py u  Profile u
main.py > Class
626 def remove_section(
627     ClassId:int , db: sqlite3.Connection = Depends(get_db)
628 ):
629     # checks if class exist
630     cur = db.execute("Select * from classes where ClassId = ?",[ClassId])
631     entry = cur.fetchone()
632     if(not entry):
633         raise HTTPException(
634             status_code=status.HTTP_409_CONFLICT,
635             detail= 'Class Does Not Exist',
636         )
637     try:
638         db.execute(
639             """
640             DELETE FROM Classes WHERE ClassId= ?
641             """
642             , [ClassId])
643         # Remove students from enrollments and waitlists
644         db.execute(
645             """
646             DELETE FROM Enrollments WHERE ClassId= ?
647             """
648             , [ClassId])
649         db.execute(
650             """
651             DELETE FROM WaitingLists WHERE ClassId= ?
652             """
653             , [ClassId])
654         db.commit()
655     except sqlite3.IntegrityError as e:
656         db.rollback()
657         raise HTTPException(
658             status_code=status.HTTP_40,
659             detail={"type": type(e).__name__, "msg": str(e)},
660         )
661     return {'status':"Class Deleted Successfully"}
662
663 # Change Instructor for a Section
664 @app.put("/classes/{ClassId}/instructor",status_code=status.HTTP_200_OK)
665 def change_instructor(
666     ClassId:int, Instructor:UpdateInstructor , db: sqlite3.Connection = Depends(get_db)
667 ):
668     # checks if class exist
669     cur = db.execute("Select * from classes where ClassId = ?",[ClassId])
670     entry = cur.fetchone()
671     if(not entry):
672         raise HTTPException(
673             status_code=status.HTTP_409_CONFLICT,
674             detail= 'Class Does Not Exist',
675         )
676     # checks if instructor exist
677     cur = db.execute("Select * from instructors where InstructorId = ?",[Instructor.InstructorId])
678     entry = cur.fetchone()
679     if(not entry):
680         raise HTTPException(
681             status_code=status.HTTP_409_CONFLICT,
682             detail= 'Instructor Does Not Exist',
683         )
684     # Update instructor for a section
685     cur = db.execute(
686         """
687         UPDATE Sections SET InstructorId = ? WHERE ClassId = ?
688         """
689         , [Instructor.InstructorId, ClassId])
690     db.commit()
691     return {'status':"Instructor Updated Successfully"}
```

Ln 17, Col 20 5spaces

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.



```
679     if(not entry):
680         raise HTTPException(
681             status_code=status.HTTP_409_CONFLICT,
682             detail= 'Instructor Does Not Exist',
683         )
684     try:
685         db.execute(
686             """
687             UPDATE Classes SET InstructorId = ? where ClassId = ?
688             """
689             [Instructor.InstructorId,ClassId])
690         db.commit()
691     except sqlite3.IntegrityError as e:
692         db.rollback()
693         raise HTTPException(
694             status_code=status.HTTP_40,
695             detail={"type": type(e).__name__, "msg": str(e)},
696         )
697     return {'status':"Instructor Changed Successfully"}
698
699
700 # Freeze automatic enrollment from waiting lists
701 @app.put("/classes/{ClassId}/freeze-enrollment",status_code=status.HTTP_200_OK)
702 def freeze_enrollment(
703     ClassId:int, db: sqlite3.Connection = Depends(get_db)
704 ):
705     # checks if class exist
706     cur = db.execute("Select * from classes where ClassId = ?",[ClassId])
707     entry = cur.fetchone()
708     if(not entry):
709         raise HTTPException(
710             status_code=status.HTTP_409_CONFLICT,
711             detail= 'Class Does Not Exist',
712         )
713     if(entry['AutomaticEnrollmentFrozen'] == 1):
714         raise HTTPException(
715             status_code=status.HTTP_409_CONFLICT,
716             detail= 'Automatic Enrollment Frozen is already ON',
717         )
718     try:
719         db.execute(
720             """
721             UPDATE Classes SET AutomaticEnrollmentFrozen = 1 where ClassId = ?
722             """
723             [ClassId])
724         db.commit()
725     except sqlite3.IntegrityError as e:
726         db.rollback()
727         raise HTTPException(
728             status_code=status.HTTP_40,
729             detail={"type": type(e).__name__, "msg": str(e)},
730         )
731     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.

```
main.py U  project1_database.py U X
project1_database.py > create_database
1 import sqlite3
2 import datetime
3
4 def create_database():
5     conn = sqlite3.connect("project1.db")
6     cursor = conn.cursor()
7
8     cursor.execute('''
9         CREATE TABLE IF NOT EXISTS Instructors (
10             InstructorId INTEGER PRIMARY KEY,
11             FirstName TEXT,
12             LastName TEXT,
13             Email TEXT
14         )
15     ''')
16     conn.execute("insert into Instructors(FirstName,LastName,Email) values('Mike','Garcia','MikeGarcia@gmail.com');")
17     conn.execute("insert into Instructors(FirstName,LastName,Email) values('Denise','Jones','DeniseJones@gmail.com');")
18     conn.execute("insert into Instructors(FirstName,LastName,Email) values('Zack','Smith','ZackSmith@gmail.com');")
19
20
21     cursor.execute('''
22         CREATE TABLE IF NOT EXISTS Classes (
23             ClassId INTEGER PRIMARY KEY,
24             InstructorId INT REFERENCES Instructors(InstructorId),
25             Department TEXT,
26             CourseCode TEXT,
27             SectionNumber INTEGER,
28             ClassName TEXT,
29             CurrentEnrollment INTEGER,
30             MaxEnrollment INTEGER,
31             AutomaticEnrollmentFrozen INTEGER DEFAULT 0
32         )
33     ''')
34     conn.execute("insert into Classes(Department,CourseCode,SectionNumber,ClassName,InstructorID,\
35         CurrentEnrollment,MaxEnrollment) values('Computer Science','CPSC351',1,\
36         'Operating Systems',5,30,45);")
37     conn.execute("insert into Classes(Department,CourseCode,SectionNumber,ClassName,InstructorID,\
38         CurrentEnrollment,MaxEnrollment) values('Computer Science','CPSC240',2,\
39         'Assembly',4,23,30);")
40     conn.execute("insert into Classes(Department,CourseCode,SectionNumber,ClassName,InstructorID,\
41         CurrentEnrollment,MaxEnrollment) values('Computer Science','CPSC223',3,\
42         'Python',3,30,35);")
43
44
45     cursor.execute('''
46         CREATE TABLE IF NOT EXISTS Students (
47             StudentId INTEGER PRIMARY KEY,
48             FirstName TEXT,
49             LastName TEXT,
50             Email TEXT
51         )
52     ''')
53     conn.execute("insert into Students(FirstName,LastName,Email) values('FirstName_1','LastName_1','abc1@gmail.com');")
54     conn.execute("insert into Students(FirstName,LastName,Email) values('FirstName_2','LastName_2','abc2@gmail.com');")
55
```

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.

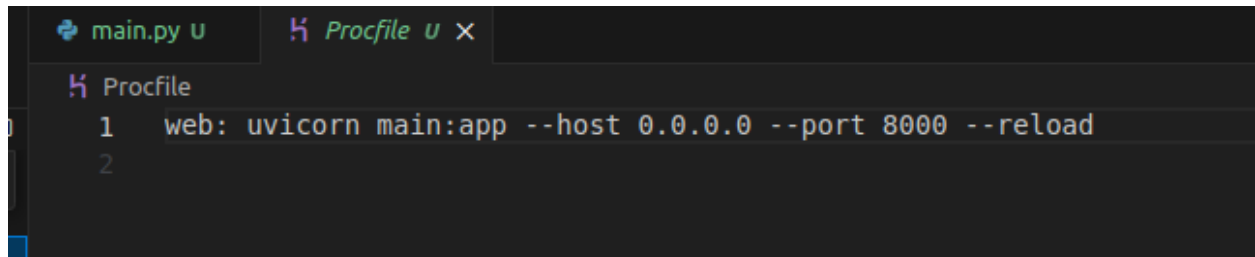
Finally, the connection is committed and closed, then the database is created.

```
main.py U project1_database.py U X
project1_database.py > create_database
54 conn.execute("insert into Students(FirstName,LastName,Email) values('FirstName_2','LastName_2','abc2@gmail.com');")
55 conn.execute("insert into Students(FirstName,LastName,Email) values('FirstName_3','LastName_3','abc3@gmail.com');")
56 conn.execute("insert into Students(FirstName,LastName,Email) values('FirstName_4','LastName_4','abc4@gmail.com');")
57 conn.execute("insert into Students(FirstName,LastName,Email) values('FirstName_5','LastName_5','abc5@gmail.com');")
58 conn.execute("insert into Students(FirstName,LastName,Email) values('FirstName_6','LastName_6','abc6@gmail.com');")
59 conn.execute("insert into Students(FirstName,LastName,Email) values('FirstName_7','LastName_7','abc7@gmail.com');")
60 conn.execute("insert into Students(FirstName,LastName,Email) values('FirstName_8','LastName_8','abc8@gmail.com');")
61 conn.execute("insert into Students(FirstName,LastName,Email) values('FirstName_9','LastName_9','abc9@gmail.com');")
62 conn.execute("insert into Students(FirstName,LastName,Email) values('FirstName_10','LastName_10','abc10@gmail.com');")
63
64
65 cursor.execute('''
66     CREATE TABLE IF NOT EXISTS Enrollments (
67         EnrollmentId INTEGER PRIMARY KEY,
68         StudentId INT REFERENCES Students(StudentId),
69         ClassId INT REFERENCES Classes(ClassId),
70         EnrollmentDate TEXT,
71         Dropped INT DEFAULT 0
72     )
73 ''')
74
75 # conn.execute("insert into Enrollments(StudentID,ClassID,EnrollmentDate) values(1,1,datetime('now'))");")
76
77 cursor.execute('''
78     CREATE TABLE IF NOT EXISTS WaitingLists (
79         WaitListId INTEGER PRIMARY KEY,
80         StudentId INT REFERENCES Students(StudentId),
81         ClassId INT REFERENCES Classes(ClassId),
82         WaitingListPos INT,
83         DateAdded TEXT
84     )
85 ''')
86
87 # conn.execute("insert into waitingLists(StudentID,ClassID,WaitingListPos,DateAdded) values(1,1,1,datetime('now'))");")
88
89
90
91 conn.commit()
92 conn.close()
93
94 create_database()
95
96
97 # conn = sqlite3.connect("pro1.db")
98 # cursor = conn.cursor()
99 # conn.execute("insert into Classes(Department,CourseCode,SectionNumber,ClassName,InstructorID,\
100 #     CurrentEnrollment,MaxEnrollment) values('Computer Science','CPSC541',1,\
101 #     'Advance Software Process',2,40,40);")
102 # conn.commit()
103 # conn.close()
```

Ln 13, Col 23 Spaces:

Procfile

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

A screenshot of a code editor with a dark theme. At the top, there are two tabs: 'main.py U' and 'Procfile U x'. The 'Procfile U' tab is active, showing the content of the Procfile. The text in the Procfile is: '1 web: uvicorn main:app --host 0.0.0.0 --port 8000 --reload' followed by a blank line '2'.

```
1 web: uvicorn main:app --host 0.0.0.0 --port 8000 --reload
2
```