سیستم انتخاب واحد

Directory:

└── dtool

├── \_\_init\_\_.py

├── crud.py

├── model.py

├── validators.py

├──schemas.py

├──engineering\_fields.json

├──province.json

└──routers

├──\_\_init\_\_.py

├──course.py

├──master.py

├──student.py

├── db.py

├──docker-compose.yml

├──Dockerfile

├──main.py

├──requirements.txt

فایل db.py

from sqlalchemy import create\_engine

from sqlalchemy.ext.declarative import declarative\_base

from sqlalchemy.orm import sessionmaker

SQLALCHEMY\_DATABASE\_URL = 'postgresql://postgres:13644631@localhost:5432/student\_system'

engine = create\_engine(SQLALCHEMY\_DATABASE\_URL)

SessionLocal = sessionmaker(autocommit=False, autoflush=False, bind=engine)

Base = declarative\_base()

این ماژول برای تنطیم و ارتباط با یک پایگاه داده postgressql با استفاده ازORM ، sqlalchemy استفاده می شود که شامل یک URL که شامل اطلاعات موردنیازبرای وصل شدن به پایگاه داده postgressql است و از الگوی زیر پیروی می کند :

SQLALCHEMY\_DATABASE\_URL = "postgresql://user:password@postgresserver/db"

با استفاده از create\_engine یک موتور پایگاه داده می سازیم با پایگاه داده ارتباط برقرار کند ، sessionLocal برای ایجاد session ها جهت ارتباط و تعامل با پایگاه داده و Base برای ایجاد و تعریف مدل ها

فایل crud

برای هر کدام از اعمال create, read, update, delete یک مثال را توضیح می دهم .

def create\_student(db: Session, student: schemas.Student):

    """

    Create a new student record.

    Args:

        db (Session): Database session.

        student (schemas.Student): Student data.

    Returns:

        Student: Created student object.

    """

    db\_student = Student(\*\*student.model\_dump())

    db.add(db\_student)

    db.commit()

    db.refresh(db\_student)

    return db\_student

این تابع یک رکورد را برای دانشجو ایجاد می کند که از اسکیمای schemas.student استفاده می کند که به شی student تبدیل می شود ، تابع model\_dump اسکیما را تبدیل به دیکشنری می کند و سپس رکورد را به پایگاه داده add وcommit و refresh می کند

def get\_student(db: Session, stid: int):

    """

    Retrieve a student record.

    Args:

        db (Session): Database session.

        stid (int): Student ID.

    Returns:

        Student: Retrieved student object.

    Raises:

        HTTPException: If student with the given ID does not exist.

    """

    get\_std = db.query(Student).filter(Student.STID == stid).first()

    if get\_std is None :

        raise HTTPException(status\_code=400, detail="student is not found")

    return get\_std

این تابع id هر درس را میگیرد و اگر وجود نداشت یک استثنا با کد 400 به کاربر بر می گرداند و اگر رکورد وجود داشت ان را به کاربر نمایش می دهد

def update\_student(db: Session, stid: int, updated\_data: dict):

    """

    Update a student record.

    Args:

        db (Session): Database session.

        stid (int): Student ID.

        updated\_data (dict): Updated data for the student.

    Returns:

        Student: Updated student object.

    Raises:

        HTTPException: If student with the given ID does not exist.

    """

    student = db.query(Student).filter(Student.STID == stid).first()

    if student is None:

        raise HTTPException(status\_code=400, detail="student is not found")

    for key, value in updated\_data.items():

        if value is None:

            del updated\_data[key]

    for key, value in updated\_data.items():

        setattr(student, key, value)

    db.commit()

    db.refresh(student)

    return student

این تابع یک رکورد را با stid وداده های بروزشده ، بروز می کند ، اگر دانشجو وجود نداشته باشد یک استثنا برمیگرداند و داده های که تغییر داده شده را بروزرسانی میکند

def del\_std(db: Session, stid: int):

    """

    Delete a student record.

    Args:

        db (Session): Database session.

        stid (int): Student ID.

    Returns:

        str: Success message.

    Raises:

        HTTPException: If student with the given ID does not exist.

    """

    student = db.query(Student).filter(Student.STID == stid).first()

    if student is None:

        raise HTTPException(status\_code=400, detail="student is not found")

    db.delete(student)

    db.commit()

    return "delete record is succesful"

این تابع یک stid از کاربر گرفته و اگروجود نداشت یک استثنا بر می گرداند و اگر هم وجود داشت ان را رکورد را حذف میکند

فایل model.py

در این فایل با استفاده از sqlalchemy ما سه جدول به نام های student, master, lesson داریم که هیچ فیلدی نمی تواند خالی باشد و هر جدول یک کلید اصلی دارد که به ترتیب ایدی درس و استاد و دانشجو هستند

فایل schemas

این فایل شامل مدل های pydantic برای موجودیت های سیستم انتخاب واحد دانشجویان هستد که شامل مدل های برای دریافت اطلاعات و نشان دادن اطلاعات و اپدیت اطلاعات (که در این اسکیما باید همه فیلد ها اختیاری باشند) است

فایل validators

این فایل شامل توابعی برای اعتبار سنجی شهر و تاریخ و رشته تحصیلی و دانشکده و کد ملی است ، این توابع ساده هستند و از توضیح تک تک ان ها گذر می کنیم

فایل course در پوشه routers

def get\_db():

    """

    Provides a database session and ensures it is closed after use.

    """

    db = SessionLocal()

    try:

        yield db

    finally:

        db.close()

این تابع برای مدیریت جلسات پایگاه داده استفاده می شود ، دستور yield جلسه پایگاه داده را در اختیار کاربر قرار می دهد و وقتی استفاده از جلسه تمام می شود و بخش finally اجرا می شود و جلسه پایگاه داده به درستی بسته می شود

@router.get("/getcsr/{CID}", response\_model= schemas.Lesson, summary= "Get Course", description="Getting course information")

def get\_csr(CID: int, db: session = Depends(get\_db)):

    """

    Retrieve course information by course ID (CID).

    Args:

        CID (int): Course ID.

        db (session): Database session dependency.

    Returns:

        schemas.Lesson: Course information.

    """

    get\_ls = crud.get\_lsn(db, CID)

    return get\_ls

@router.post("/creatcsr", response\_model= schemas.Lesson, summary= "Create Course", description="Create a course ")

def course(ln: schemas.Lesson, db: session = Depends(get\_db)):

    """

    Create a new course with validation checks.

    Args:

        ln (schemas.Lesson): Course information.

        db (session): Database session dependency.

    Returns:

        schemas.Lesson: Created course.

    """

    errors = {}

    db\_std = db.query(model.Lesson).filter(model.Lesson.CID == ln.CID).first()

    if db\_std :

        raise HTTPException(status\_code= 400, detail="courseID already exists ,courseID must be unique")

    if len(str(ln.CID)) != 5 :

        errors["CID"] = "CID must be 5 digits"

    if len(ln.CName) > 25 or  not validators.check\_farsi\_name(ln.CName):

        errors["CName"] = "The maximum length of the string should be 25 and all characters should be Farsi"

    if not validators.college\_trust(ln.Department):

        errors["Department"] = "name college is wrong"

    if not re.match(r"[1-5]", ln.Credit):

        errors["credit"] = "The number of units must be between 1 and 4"

    if errors:

        raise HTTPException(status\_code = 400, detail={"detail": "Validation error", "errors": errors})

    else:

        db\_user = crud.create\_lesson(db, ln)

        return db\_user

@router.patch("/uptcsr/{CID}", response\_model=schemas.UpdateLesson, summary= "Update Course", description= "Update a course")

def upt\_cour(CID: int, ln : schemas.UpdateLesson, db: session = Depends(get\_db)):

    """

    Update existing course information.

    Args:

        CID (int): Course ID.

        ln (schemas.UpdateLesson): Updated course information.

        db (session): Database session dependency.

    Returns:

        schemas.UpdateLesson: Updated course information.

    """

    cour\_que = db.query(model.Lesson).filter(CID == model.Lesson.CID).first()

    if cour\_que is None:

        raise HTTPException( status\_code= 400, detail= "CID is not found")

    errors = {}

    update\_data = ln.model\_dump(exclude\_unset= True)

    if "CName" in update\_data and (len(ln.CName) > 25 or  not validators.check\_farsi\_name(ln.CName)):

        errors["CName"] = "The maximum length of the string should be 25 and all characters should be Farsi"

    if "Department" in update\_data and not validators.college\_trust(ln.Department):

        errors["Department"] = "name college is wrong"

    if "Credit" in update\_data and (not re.match(r"[1-5]", ln.Credit)):

        errors["credit"] = "The number of units must be between 1 and 4"

    if errors :

        raise HTTPException(status\_code=400, detail={"detail": "Validation error", "errors": errors})

    result = crud.update\_course(db, CID, update\_data)

    return result

@router.delete("/Delcsr/{CID}", summary= "Delete Course", description= "Delete a course")

def del\_lsn(CID: int, db: session = Depends(get\_db)):

    """

    Delete a course by course ID (CID).

    Args:

        CID (int): Course ID.

        db (session): Database session dependency.

    Returns:

        dict: Confirmation message for deletion.

    """

    del\_ls = crud.del\_lsn(db, CID)

    return del\_ls

این روتر شامل چهار api است

get\_csr با متود get بااستفاده از cid اطلاعات یک درس را بر می گرداند

course متد post برای ایجاد یک درس جدید با اعتبار سنجی های لازم

upd\_cour متد patch برای بروزرسانی اطلاعات یک درس با cid

del\_lsn متد delete برای حذف درس با cid

فایل course

از توضیح قسمت های تکراری صرف نظر می شود

شامل چهارapi است :

get\_msr

master

upd\_mas

del\_msr

قسمت اعتبار سنجی ایدی درس ها در master توضیح داده می شود

if mr.LCourseIDs:

        for course in mr.LCourseIDs:

                db\_lcourse = db.query(model.Lesson).filter(course == model.Lesson.CID).first()

                if not db\_lcourse:

                    errors["LCourseIDs"] = f"Lesson with ID {course} not found"

چک می کند اگر ایدی درس ها در پایگاه داده وجود نداشت یک استثنا به کاربر برگرداند

@router.patch("/UpdStu/{STID}", summary= "Update Student", description= "Update a student user")

def update\_std(STID: int, student\_update: schemas.UpdateStudent, db: session = Depends(get\_db)):

در این قسمت چک می کند اگر هر کدام از فیلد ها را کاربر وارد کرده بود اعتبار سنجی ان را انجام دهد

فایل student

مشابه روتر های قبلی است شامل چهار api برای چهار عملیات گرفتن اطلاعات و ثبت اطلاعات و اپدیت اطلاعات و حذف اطلاعات است

فایل main

app.include\_router(course.router)

app.include\_router(master.router)

app.include\_router(student.router)

روترهای مربوط به دوره‌ها (course)، اساتید (master) و دانشجویان (student) به برنامه اضافه می‌شوند.

def long\_running\_process():

    """

    Simulates a long-running process, allowing interruption with cleanup.

    """

    try:

        print("Performing a long-running process. Press Ctrl+C to interrupt.")

        for i in range(5):

            time.sleep(1)

            print(f"Processing step {i + 1}")

    except KeyboardInterrupt:

        print("\nInterrupted! Cleaning up before exiting.")

    finally:

        print("Exiting the program.")

این تابع یک فرایند طولانی را شبیه سازی می کند تا برنامه به طور صحیح با ctrl+c متوقف شود

if \_\_name\_\_ == "\_\_main\_\_":

    try:

        uvicorn.run(app)

    except KeyboardInterrupt:

        long\_running\_process()

اگر فایل به عنوان یک برنامه اصلی اجرا شود (به صورت اسکریپتی) سرور uvicorn برای اجرای fastapi راه اندازی می شود و در غیر این صورت تابع long\_running\_process() اجرا می شود تا برنامه به طورصحیح متوقف میکند .

فایل requirements

با دستور زیر فایل نیازمندی هارو ایجاد می کنیم



فایل Dockerfile

FROM python:3.12.0

ENV PYTHONDONTWRITEBYTECODE=1

ENV PYTHONUNBUFFERED=1

COPY requirements.txt .

RUN python -m pip install -r requirements.txt

WORKDIR /app

COPY . /app

RUN adduser -u 5678 --disabled-password --gecos "" appuser && chown -R appuser /app

USER appuser

EXPOSE 8000

CMD ["uvicorn", "main:app", "--host", "0.0.0.0", "--port", "8000"]

در این داکرفایل ابتدا از ایمیج رسمی پایتون استفاده می کنیم ، با اولین دستور ENV مانع از نوشتن فایل های bytecode می شویم و دومین دستورENV خروجی پایتون را بدون بافر نمایش می دهد ، در دو خط بعدی فایل نیازمندی ها به دایرکتوری کانتینر کپی می شود و نیازمندی ها را نصب می کند ، در دو خط بعدی دایرکتوری کاری را به /app تغییر می دهد و تمام فایل ها ودایرکتوری های موجود در مسیرفعلی را در /app ایمیج کپی می کند

در دو خط بعدی یک کاربر با نام appuser ایجاد کرده و مالکیت دایرکتوری را به ان اختصاص می دهد ، کاربر فعلی را به appuser تغییر می دهد تا از این به بعد دستورات به عنوان این کاربر انجام شوند

پورت 8000 برای دسترسی برنامه به وب بازمی کنیم و با خط اخر و دستور اجرایی برای راه اندازی uvicorn و درنتیجه برنامه fastapi را اجرامی کند .

docker-compose.yml

version: '3.8'

services:

  app:

    image: app

    build:

      context: .

      dockerfile: ./Dockerfile

    ports:

      - 8000:8000

    depends\_on:

      - db

  db:

    restart: always

    image: postgres

    environment:

      POSTGRES\_USER: postgres

      POSTGRES\_PASSWORD: 13644631

      POSTGRES\_DB: student\_system

    ports:

      - "5432:5432"

این فایل docker-compose.yml رای تنظیم و راه‌اندازی همزمان برنامه و پایگاه داده PostgreSQL با استفاده از Docker Compose استفاده می‌شود. سرویس برنامه از تصویر ساخته شده با Dockerfile خود استفاده می‌کند و به پایگاه داده PostgreSQL که به صورت مستقل با تصویر postgres اجرا می‌شود، وابسته است.

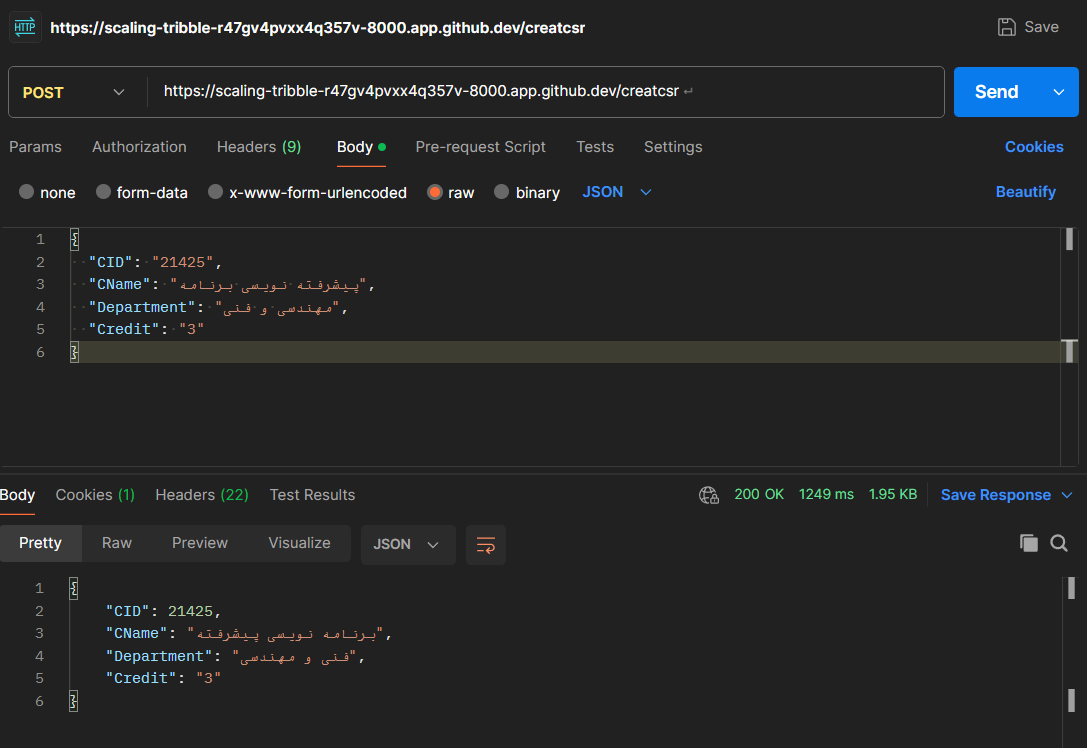
داکرایز کردن

با دستور زیر ایمیج های ما ساخته می شوند و کانتینر های شروع به کار می کنند



نتایج مقدار دهی پروژه

ساخت درس

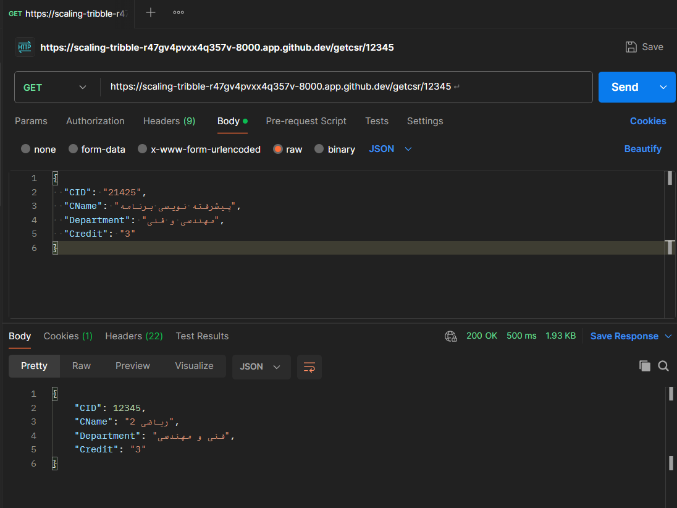


ارور دادن هنگام ورودی نامعتبر

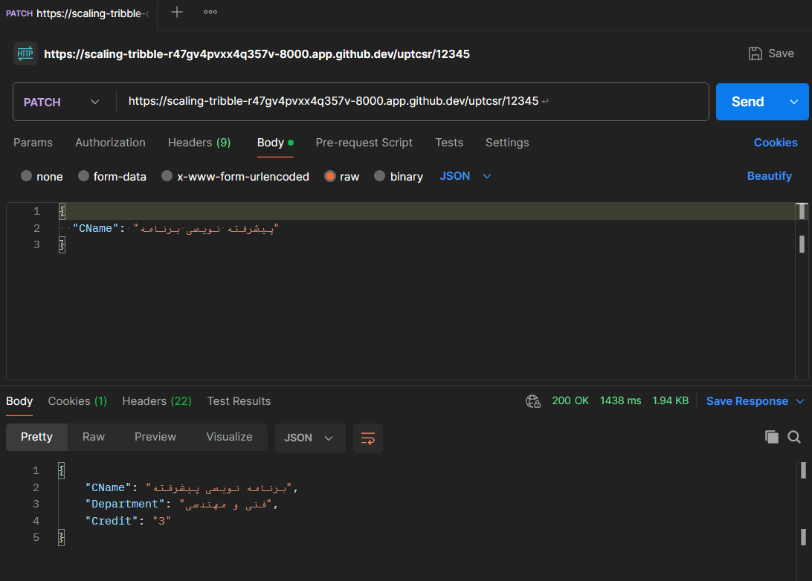
A screenshot of a computer program

Description automatically generated

گرفتن اطلاعات درس



اپدیت کردن درس

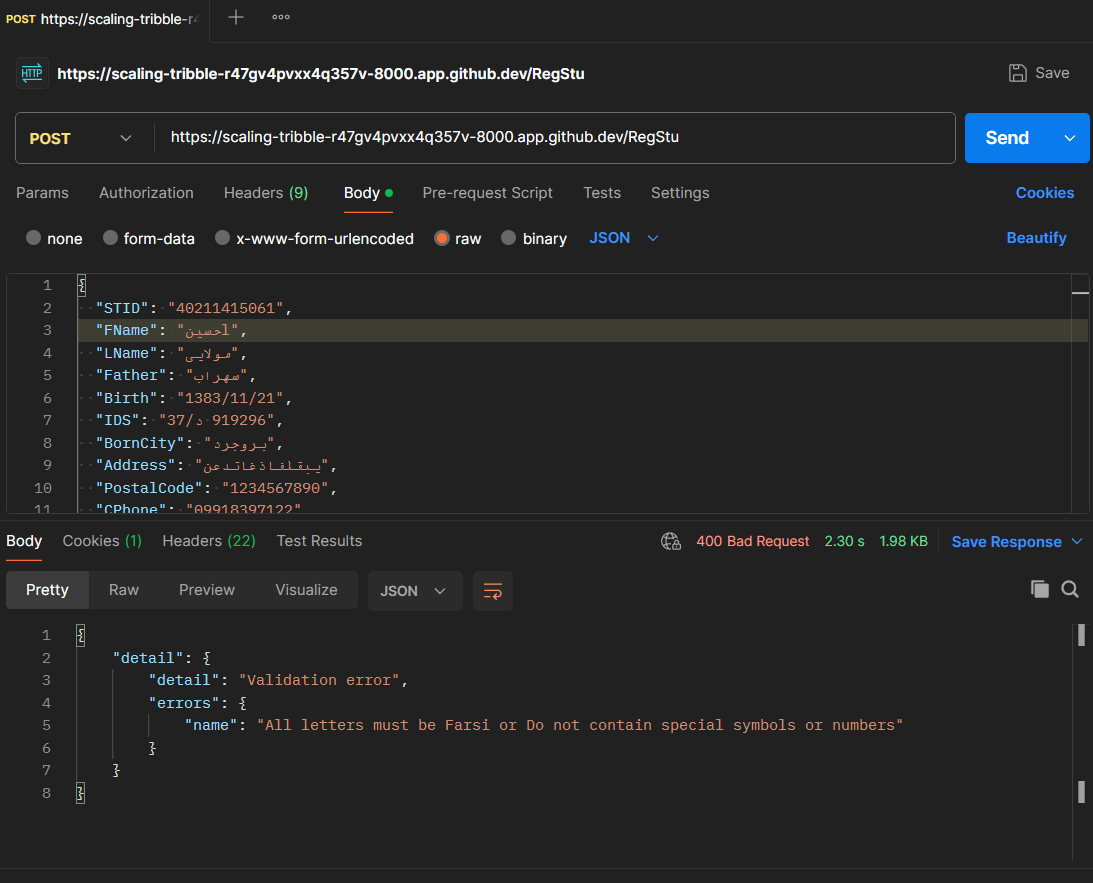


ساخت دانشجو

A screenshot of a computer

Description automatically generated

ارور هنگام مقداردهی نامناسب



اپدیت دانشجو

A screenshot of a computer

Description automatically generated