In the name of God…

دانشگاه لرستان

Hello, my name is **Amir Taha Nemati**, and you are reading the README file for my **Advanced Programming Languages (APL)** project.

This project was created as part of my university course. It is a simple web application that I built to practice the things I learned during the semester.

🔧 Technologies Used

- **Backend:** Python with FastAPI
- **Database:** SQLite
- **Frontend:** HTML, CSS, JavaScript
- **Other tools:** Docker and containers

📌 Project Overview

The project is a basic full-stack web application. It includes a backend API, a frontend user interface, and uses a database to store data. I used Docker to run the services in containers.

🔗 **My GitHub profile:** amirtaha nemati

📝 Project Tasks and Report

This section describes the steps I followed during the development and deployment of this project:

1. **Database Design:**
   I created three tables in the SQLite database: Student, Lecturer, and Course.
2. **CRUD Operations with FastAPI:**
   For each table (student, teacher, and course), I implemented Create, Read, Update, and Delete operations using FastAPI.
3. **Object-Oriented Design with Inheritance:**
   I created a base class called Human, and both Student and Lecturer classes inherit from it.
4. **Error Handling:**
   I used Python exceptions and FastAPI error responses to manage all possible errors in the system.
5. **Dockerization:**
   I created a `Dockerfile` and `docker-compose.yml` to containerize the backend, frontend, and database.
6. **GitHub Repository:**
   All project files were pushed from my IDE to my GitHub repository.
7. **Virtual Server Setup:**
   I temporarily rented a Linux-based VPS and installed necessary software (Docker, Git, Python, etc.) on it.
8. **Nginx Configuration:**
   I configured Nginx to act as a reverse proxy and serve the web application.
9. **Domain Setup:**
   I bought a `.ir` domain from IRNIC, then linked it to my server IP using Cloudflare.
10. **Running Containers on Server:**
    I ran the Docker containers on the VPS to make the APIs publicly available.
11. **Web Interface:**
    I created simple HTML pages for Create, Read, Update, and Delete operations.
12. **Public API Access:**
    All API documentation is available at:
    `http://amirtahanemati.ir/docs`

1. **Database Design:**

In this project, I created three main tables in the database: Student, Lecturer, and Course. These tables were created using SQLModel with a SQLite database. When the app starts, the tables are automatically created by the on_startup function in FastAPI.

```python
sqlite_file_name = "database.db"
sqlite_url = f"sqlite:///{sqlite_file_name}"
connect_args = {"check_same_thread": False}
engine = create_engine(sqlite_url, connect_args=connect_args)


def broader_search(search: str, fields: list, query):
    if search:
        search = f"%{search}%"
        conditions = [field.ilike(search) for field in fields]
        query = query.filter(or_(*conditions))
    return query


def create_db_and_tables():
    SQLModel.metadata.create_all(engine)


def get_session():
    with Session(engine) as session:
        yield session


SessionDep = Annotated[Session, Depends(get_session)]


@app.on_event("startup")
def on_startup():
    create_db_and_tables()
```

This part of the code sets up a connection to the SQLite database using SQLModel. First, the database file name is defined. Then, a create_engine function is used to connect to the database.The function create_db_and_tables creates all tables based on the models. This function is called in the FastAPI startup event, so the tables are created automatically when the application starts.

2. CRUD Operations with FastAPI:

This section focuses on the implementation of CRUD (Create, Read, Update, Delete) operations for the database tables (Student, Lecturer, and Course) using FastAPI. FastAPI was chosen as the primary framework for building the project's APIs, enabling fast and efficient handling of HTTP requests. Below is a sample of the code for CRUD operations on the Student table:

```
@app.post("/students/")
def create_student(student: Student, session: SessionDep) -> Student:…



@app.get("/students/")
def read_students(…



@app.get("/students/{student_id}")
def read_student(student_id: str, session: SessionDep) -> Student:…



@app.put("/students/{student_id}")
def update_student(student_id: str, student: Student, session: SessionDep) ->
Student:…
```

Details:

- **Create**: The create_student function adds a new student to the database.
- **Read**: The read_student function retrieves a student's information based on their ID.
- **Update**: The update_student function updates a student's information.
- **Delete**: The delete_student function removes a student from the database.
- SessionDep is used to manage the connection to the SQLite database.

✓ **Similar operations were implemented for the Lecturer and Course tables.**

3. **Object-Oriented Design with Inheritance:**

This section describes the implementation of object-oriented programming (OOP) using inheritance to structure the data models. A base class named Human was defined to include common attributes shared between students and lecturers. The Student and Lecturer classes inherit from this base class to avoid code duplication and ensure a cohesive structure. Below is the sample code for this design:

```
class Human(SQLModel):
    FName: str = Field(index=True)
    LName: str = Field(index=True)
    ID: str = Field(unique=True)
    Birth: str
    BornCity: str
    Address: str | None = Field(default=None)
    PostalCode: str | None = Field(default=None)
    Cphone: str | None = Field(default=None)
    Hphone: str | None = Field(default=None)

    @validator()…

class Student(Human, SQLModel, table=True):
    STID…
    …
class Lecturer(Human, SQLModel, table=True):
    LID…
    …
```

Details:

- **Human Base Class**: This class includes common attributes such as first name, last name, ID, birth date, city of birth, address, postal code, mobile phone, and home phone.
- **Inheritance**: The Student and Lecturer classes inherit from Human, each with a unique attribute (STID for students and LID for lecturers).
- **Validation**: The @validator decorator is used to ensure that first and last names contain only Persian characters.
- **SQLModel**: This framework is used to define database models and integrate them with SQLite.

4. Error Handling:

In this section, I'll explain how I handled errors in the project. To keep the app stable and manage potential issues properly, I used Python exceptions and FastAPI's error responses. Here's an example of the validation code for the first and last names in the Human class to show how error handling works:

For example, validator first name and last name:

```
    @validator("FName", "LName")
    def name_check(cls, v):
        if re.match(r'^[\u0600-\u06FF\s]+$', v):
            return v
        raise ValueError("نام و نام خانوادگی باید با حروف فارسی باشد")
```

For example, validator ID:

```
    @validator("ID")
    def ID_check(cls, v):
        if re.match(r'^\d{10}$', v):
            return v
        raise ValueError("کد ملی باید 10 رقم باشد")
```

Details:

- **Name Validation**: Using Pydantic's ~~@validator~~, I check that the first and last names only contain Persian characters. If the input isn't valid, a ValueError with a clear message is raised.
- **FastAPI Error Handling**: For CRUD operations, if something like a student ID isn't found, FastAPI returns an HTTP 404 response with a message like "Student not found" (as shown in page four).
- This approach ensures users get clear error messages, and the app handles issues gracefully instead of crashing.
- For other cases, like database errors or invalid inputs, I used Python exceptions to keep the system reliable.

5. Dockerization:

Dockerfile:

```
FROM python:3.13.0-slim

WORKDIR /app

COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt

COPY . .

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

***Dockerfile**: Created a Dockerfile for the backend using the python:3.13.0-slim image. Sets /app as the working directory, copies requirements.txt and installs dependencies, copies project files, and runs uvicorn on 0.0.0.0:8000.

docker-compose.yml:

```
version: '3.8' →

services:
  backend:
    build: .
    ports:
      - "8000:8000"
    volumes:
      - ./database.db:/app/database.db
    restart: unless-stopped

  frontend:
    image: nginx:alpine
    ports:
      - "80:80"
    volumes:
      - ./index.html:/usr/share/nginx/html/index.html
      - ./styles.css:/usr/share/nginx/html/styles.css
      - ./script.js:/usr/share/nginx/html/script.js
      - ./nginx.conf:/etc/nginx/conf.d/default.conf
    depends_on:
      - backend
    restart: unless-stopped
```

***docker-compose.yml:** Defined two services:

backend: Builds the backend image, maps port 8000:8000, mounts database.db for persistent data, and enables auto-restart.

frontend: Uses nginx:alpine image, maps port 80:80, mounts frontend files (index.html, styles.css, script.js) and Nginx config, and depends on the backend service.

requirements.txt:

```
fastapi
uvicorn
sqlmodel
pydantic
```

requirements.txt: Lists dependencies like fastapi, uvicorn, sqlmodel, and pydantic.

Purpose: Containerization ensures the project runs consistently across environments, making deployment and management easier.

6. GitHub Repository:

```
PS C:\Users\amirt\Desktop\APL_Final_Project> git init
```

Repository Setup: Initialized a local repository in the project folder (APL_Final_Project) using git init.

```
PS C:\Users\amirt\Desktop\APL_Final_Project> git config --global core.autocrlf true
```

Git Configuration: Set git config --global core.autocrlf true to handle line endings correctly across operating systems.

```
PS C:\Users\amirt\Desktop\APL_Final_Project> git add .
```

Adding Files: Staged all project files with git add ..

```
PS C:\Users\amirt\Desktop\APL_Final_Project> git commit -m "Initial commit"
```

```
PS C:\Users\amirt\Desktop\APL_Final_Project> git push -u origin main
```

Commit and Push: Committed changes with git commit -m "Initial commit" and pushed to the main branch on GitHub using git push -u origin main.

```
PS C:\Users\amirt\Desktop\APL_Final_Project> git remote -v
```

Remote Verification: Checked the remote connection with git remote -v to ensure the repository is linked to GitHub.

- ✓ **Purpose:** Uploaded all project code and files to GitHub for backup and to share the project with others.

7. Virtual Server Setup:

Server Rental: Rented a Ubuntu-based VPS.

Server Access:

```
ssh username@51.68.183.216
```

Software Installation:

```
sudo apt update
sudo apt upgrade -y
sudo apt install -y docker.io docker-compose git python3 python3-pip
```

Docker Activation:

```
sudo systemctl start docker
sudo systemctl enable docker
```

- ✓ **Purpose:** Prepared the server to publicly host the project.

8. Nginx Configuration:

Nginx Config File: Created an nginx.conf file that configures the web server to listen on port 80 and support the domains amirtahanemati.ir and www.amirtahanemati.ir.

Location Settings:

For the / path: Serves static files (like index.html) from /usr/share/nginx/html. If a file isn't found, it redirects to index.html.

For the /api/ path: Proxies requests to the backend service (http://backend:8000). Headers like Host, X-Real-IP, and X-Forwarded-For are set to pass correct information to the backend.

nginx.conf:

```
server {
    listen 80;
    server_name amirtahanemati.ir www.amirtahanemati.ir;

    location / {
        root /usr/share/nginx/html;
        try_files $uri /index.html;
    }

    location /api/ {
        proxy_pass http://backend:8000/;
        proxy_set_header Host $host;
        proxy_set_header X-Real-IP $remote_addr;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
    }
}
```

- ✓ **Purpose:** Nginx acts as a reverse proxy to properly serve the web interface and APIs to users.

9. Domain Setup:

Domain Purchase: Purchased the domain amirtahanemati.ir from IRNIC (Iran's domain registry at nic.ir).

IRNIC Registration: Created an account on nic.ir, registered the domain, and provided ownership details (like name and email).

Cloudflare Integration: Linked the domain to Cloudflare. In the Cloudflare panel, configured DNS servers and added the server IP (51.**.***.***) as an A Record.

DNS Configuration: Set up DNS records to point amirtahanemati.ir and www.amirtahanemati.ir to the server's IP.

- ✓ **Purpose:** These settings made the web application and APIs publicly accessible via the domain.

10. Running Containers on Server:

File Upload: Uploaded project files to the server using FileZilla, placing them in the /var/www/amirtahanemati directory.

Directory Access: Connected to the server via SSH and navigated to the project directory:

```
cd /var/www/amirtahanemati
```

Repository Update (Optional): If needed, updated the project code from GitHub using git pull.

Running docker-compose: In the project directory, executed the docker-compose.yml file to start the backend and frontend containers:

```
sudo docker-compose up -d
```

Container Verification: Checked that the backend (port 8000) and frontend (port 80) containers were running correctly:

```
docker ps
```

Public Access: After running the containers, APIs became accessible at http://amirtahanemati.ir/api/ and the web interface at http://amirtahanemati.ir.

- ✓ Purpose: Uploading files and running containers ensured the web application and APIs were properly deployed and accessible to users.

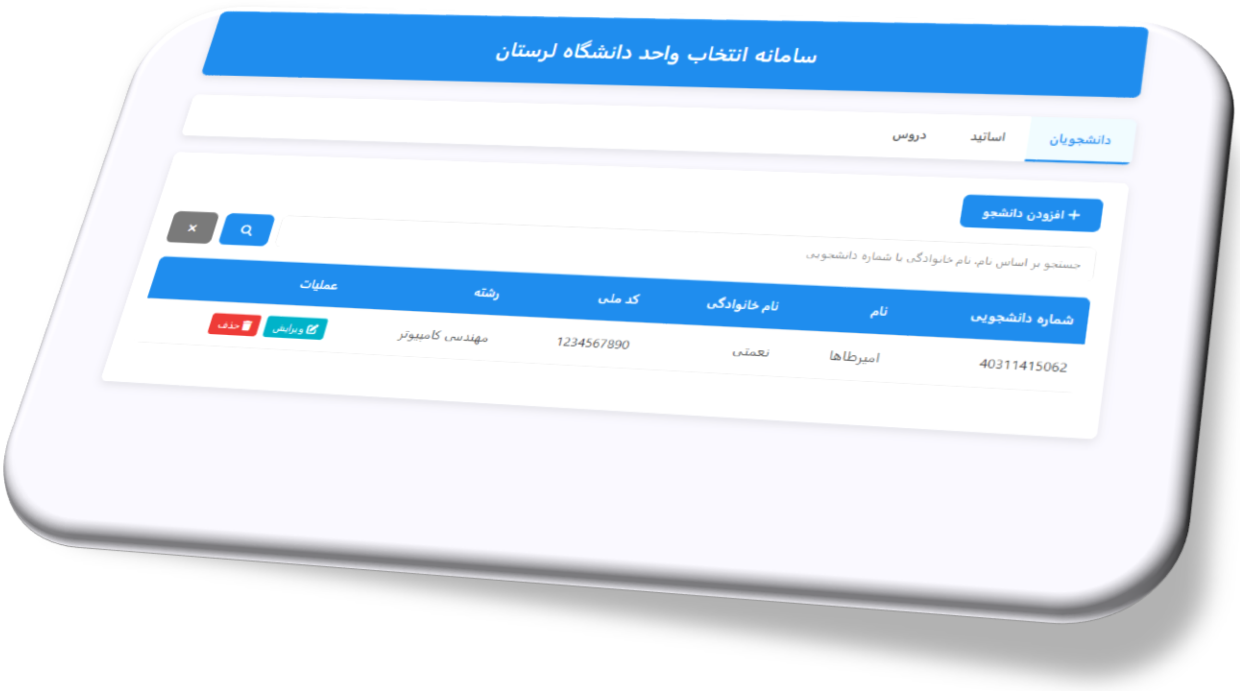
11. Web Interface:

HTML Pages: Built simple HTML pages for CRUD operations (Create, Read, Update, Delete). Each page includes input forms and buttons to interact with APIs.

CSS Styling: Used styles.css for visual design. Layout features a fixed header, minimal forms with smooth borders, and a blue-white color scheme for a clean, professional look.

JavaScript Interactivity: The script.js file handles API requests and displays results on the page. For example, a success message appears after adding a student.

Visual Layout: Pages follow a single-column design with prominent buttons and neatly arranged input fields for user convenience.



سامانه انتخاب واحد دانشگاه لرستان

شماره دانشجویی	نام	نام خانوادگی	کد ملی	رشته	عملیات
40311415062	امیرطاها	نعمتی	1234567890	مهندسی کامپیوتر	ویرایش حذف

✓ **Purpose:** The web interface provides an easy, user-friendly way to interact with APIs via a browser.

12. **Public API Access:**

API Documentation: All API documentation is available at http://amirtahanemati.ir/api. This page uses Swagger UI to display request and response details.

API Endpoint: APIs are accessible via the /api/ path on http://amirtahanemati.ir. For example, to get a list of students: http://amirtahanemati.ir/api/students/.

Access Method: Users can send HTTP requests (GET, POST, PUT, DELETE) to the APIs using tools like Postman or a browser.

Security: Requests are proxied to the backend via Nginx, and errors (like 404) return clear messages.

- ✓ **Purpose:** The APIs are publicly accessible, allowing users to interact with the application and perform CRUD operations.

Thanks for reading this README to the end! I hope this project gives you a good sense of my work. Best of luck! 🚀

🔗 **My GitHub profile:** [amirtaha nemati]