FastAPI Task Management Application: Detailed Notes

Grok 3

May 13, 2025

1 Introduction

This document provides detailed notes on a FastAPI-based task management application. The application allows creating and retrieving users, as well as managing tasks associated with users. It uses FastAPI for the API, Pydantic for data validation, and in-memory lists as a database. The code is split into three files: `models.py`, `schemas.py`, and `main.py`. Below, each file is explained line by line in a beginner-friendly manner.

2 models.py: In-Memory Database

The `models.py` file simulates a database using two lists to store users and tasks.

2.1 Code

```
1 from typing import List
2 from schemas import UserRead, Task # type: ignore
3
4 users_db: List[UserRead] = []
5 tasks_db: List[Task] = []
```

2.2 Explanation

- **Line 1: `from typing import List`**
  Imports the `List` type to specify that a variable is a list of specific types (e.g., a list of `UserRead` objects).
- **Line 2: `from schemas import UserRead, Task`**
  Imports `UserRead` and `Task` classes from `schemas.py`. These define the structure of users and tasks.
- **Line 2: `# type: ignore`**
  Suppresses type-checking warnings from tools like `mypy`, used to handle potential import issues.

- **Line 4: `users_db: List[UserRead] = []`**
  Creates an empty list `users_db` to store user data. The type `List[UserRead]` means it holds `UserRead` objects.
- **Line 5: `tasks_db: List[Task] = []`**
  Creates an empty list `tasks_db` to store task data, holding `Task` objects.

Note: These lists act as an in-memory database. Data is lost when the application restarts. In a real application, use a database like PostgreSQL.

3 schemas.py: Data Models with Pydantic

The `schemas.py` file defines data models using Pydantic for input validation and serialization.

3.1 Code

```
1  from pydantic import BaseModel, EmailStr, constr, field_validator
2  from datetime import date
3  from typing import Optional, Annotated
4
5  class UserCreate(BaseModel):
6      username: Annotated[str, constr(min_length=3, max_length=20)]
7      email: EmailStr
8
9  class UserRead(BaseModel):
10     id: int
11     username: str
12     email: EmailStr
13
14 class TaskBase(BaseModel):
15     title: str
16     description: Optional[str] = None
17     due_date: date
18     status: str = "pending"
19
20     @field_validator("due_date")
21     @classmethod
22     def due_date_cannot_be_in_past(cls, v):
23         if v < date.today():
24             raise ValueError("Due date cannot be in the past")
25         return v
26
27     @field_validator("status")
28     @classmethod
29     def validate_status(cls, v):
30         allowed_statuses = {"pending", "in_progress",
               "completed"}
31         if v not in allowed_statuses:
32             raise ValueError(f"Status must be one of:
                   {allowed_statuses}")
```

```
33          return v
34
35  class TaskCreate(TaskBase):
36      user_id: int
37
38  class Task(TaskBase):
39      id: int
40      user_id: int
```

3.2 Explanation

3.2.1 Imports

- **Lines 1–3**: Import Pydantic tools (`BaseModel`, `EmailStr`, `constr`, `field_validator`), `date` for due dates, and `Optional`, `Annotated` for type hints.

3.2.2 UserCreate Class

- **Line 5: `class UserCreate(BaseModel):`**
  Defines a model for creating users.
- **Line 6: `username: Annotated[str, constr(min_length=3, max_length=20)]`**
  A string field with 3–20 character length constraints.
- **Line 7: `email: EmailStr`**
  A field that must be a valid email address.

3.2.3 UserRead Class

- **Line 9: `class UserRead(BaseModel):`**
  Defines a model for reading user data.
- **Lines 10–12**: Includes `id` (integer), `username` (string), and `email` (email address).

3.2.4 TaskBase Class

- **Line 14: `class TaskBase(BaseModel):`**
  A base model for task data, shared by other task models.
- **Line 15: `title: str`**
  A required string for the task title.
- **Line 16: `description: Optional[str] = None`**
  An optional string for the task description.
- **Line 17: `due_date: date`**
  A required date for the tasks due date.
- **Line 18: `status: str = "pending"`**
  A string for the task status, defaulting to `"pending"`.
- **Lines 20–24: `due_date_cannot_be_in_past`**
  Validates that the due date is not in the past.

- **Lines 26–30: `validate_status`**
  Ensures the status is one of `pending`, `in_progress`, or `completed`.

3.2.5 TaskCreate and Task Classes

- **Line 32: `class TaskCreate(TaskBase):`**
  Inherits `TaskBase`, adds `user_id` for creating tasks.
- **Line 35: `class Task(TaskBase):`**
  Inherits `TaskBase`, adds `id` and `user_id` for task responses.

4 main.py: FastAPI Application

The `main.py` file defines the FastAPI application and API endpoints.

4.1 Code

```
1  from typing import List
2  from fastapi import FastAPI, HTTPException
3  from schemas import UserCreate, UserRead, TaskCreate, Task #
       type: ignore
4  from models import users_db, tasks_db
5
6  app = FastAPI()
7
8  # Auto-incrementing IDs
9  user_id_counter = 1
10 task_id_counter = 1
11
12 @app.post("/users/", response_model=UserRead)
13 def create_user(user: UserCreate):
14     global user_id_counter
15     new_user = UserRead(id=user_id_counter, **user.dict())
16     users_db.append(new_user)
17     user_id_counter += 1
18     return new_user
19
20 @app.get("/users/{user_id}", response_model=UserRead)
21 def get_user(user_id: int):
22     for user in users_db:
23         if user.id == user_id:
24             return user
25     raise HTTPException(status_code=404, detail="User not found")
26
27 @app.post("/tasks/", response_model=Task)
28 def create_task(task: TaskCreate):
29     global task_id_counter
30     # Verify user exists
31     if not any(user.id == task.user_id for user in users_db):
32         raise HTTPException(status_code=404, detail="User not
           found")
```

```
33      new_task = Task(id=task_id_counter, **task.dict())
34      tasks_db.append(new_task)
35      task_id_counter += 1
36      return new_task
37
38  @app.get("/tasks/{task_id}", response_model=Task)
39  def get_task(task_id: int):
40      for task in tasks_db:
41          if task.id == task_id:
42              return task
43      raise HTTPException(status_code=404, detail="Task not found")
44
45  @app.put("/tasks/{task_id}", response_model=Task)
46  def update_task_status(task_id: int, status: str):
47      allowed_statuses = {"pending", "in_progress", "completed"}
48      if status not in allowed_statuses:
49          raise HTTPException(status_code=400, detail=f"Status
               must be one of: {allowed_statuses}")
50      for task in tasks_db:
51          if task.id == task_id:
52              task.status = status
53              return task
54      raise HTTPException(status_code=404, detail="Task not found")
55
56  @app.get("/users/{user_id}/tasks", response_model=List[Task])
57  def get_user_tasks(user_id: int):
58      # Verify user exists
59      if not any(user.id == user_id for user in users_db):
60          raise HTTPException(status_code=404, detail="User not
               found")
61      user_tasks = [task for task in tasks_db if task.user_id ==
           user_id]
62      return user_tasks
```

4.2 Explanation

4.2.1 Setup

- **Lines 1–4**: Import necessary modules and models.
- **Line 6: `app = FastAPI()`**
  Creates the FastAPI application.
- **Lines 9–10**: Initialize counters for user and task IDs.

4.2.2 Create User Endpoint

- **Line 12: `@app.post("/users/", response_model=UserRead)`**
  Defines a POST endpoint to create users.
- **Lines 13–18**: Creates a new user, assigns an ID, adds it to `users_db`, and returns it.

4.2.3 Get User Endpoint

- **Line 20: `@app.get("/users/{user_id}", response_model=UserRead)`**
  Defines a GET endpoint to retrieve a user by ID.
- **Lines 21–25**: Searches for the user; returns it or raises a 404 error.

4.2.4 Create Task Endpoint

- **Line 27: `@app.post("/tasks/", response_model=Task)`**
  Defines a POST endpoint to create tasks.
- **Lines 28–34**: Verifies the user exists, creates a task, and adds it to `tasks_db`.

4.2.5 Get Task Endpoint

- **Line 36: `@app.get("/tasks/{task_id}", response_model=Task)`**
  Defines a GET endpoint to retrieve a task by ID.
- **Lines 37–41**: Returns the task or raises a 404 error.

4.2.6 Update Task Status Endpoint

- **Line 43: `@app.put("/tasks/{task_id}", response_model=Task)`**
  Defines a PUT endpoint to update a tasks status.
- **Lines 44–51**: Validates the status, updates the task, and returns it.

4.2.7 Get User Tasks Endpoint

- **Line 53: `@app.get("/users/{user_id}/tasks", response_model=List[Task])`**
  Defines a GET endpoint to list a users tasks.
- **Lines 54–58**: Verifies the user exists and returns their tasks.

5 Key Concepts

- **FastAPI**: A Python framework for building APIs, known for speed and automatic documentation.
- **Pydantic**: Validates data (e.g., ensuring valid emails or due dates).
- **In-Memory Database**: Uses lists for storage; data is temporary.
- **HTTP Methods**: `POST` (create), `GET` (retrieve), `PUT` (update).
- **HTTP Status Codes**: `404` (not found), `400` (bad request).

6 Running the Application

1. Save `main.py`, `schemas.py`, and `models.py` in a folder.
2. Install dependencies:

```
pip install fastapi uvicorn pydantic
```

3. Run the server:

```
uvicorn main:app --reload
```

4. Access `http://127.0.0.1:8000/docs` for interactive documentation.