Technical Assessment

1. Write a REST API which will manage tasks, each task is assigned to a user.

Presumptions

- As well as see only their own tasks, users can only perform an action on their tasks as well.
- The default status of a new task should be “Pending” if one is not supplied.
- An error should be thrown if the status is not one of
    - Pending
    - Doing
    - Blocked
    - Done
- Endpoints to include:
    - **Post** /users
    - **Post** /tasks
    - **Put** /tasks/<task-id> (this should be a put as the user will be required to provide the full object to update in the body)
    - **Delete** /tasks/<task-id>
    - **Get** /tasks
    - **Patch** /tasks/restore/<task-id> (using patch here as only one field of the task will be updated.)
- Put request should only update a non-deleted task where the owner is the current user
- Get request should only return non-deleted tasks for the current user
- Anyone should be able to create a new user, the email address and username should be unique

Out of Scope

- Pagination for get request though this should be implemented for a production system

2. If a task were to be deleted, it must be moved into a history table for record purposes.

Presumptions

Only the owner can delete a particular task
Should also store the date a task was deleted

Implementation

I decided that moving the record completely to another table is not the most efficient way of tracking this. It would require 2 almost identical tables and then looking forward to the “bonus items” the restoration of deleted tasks is also an ask.
I decided that implementing a history table which stores the user_id of who deleted it and the task_id of the deleted task alongside the date it was updated and the action performed so that we can see when and who performed a delete or restore action on it easily. This can also be

extended to track any action on a task such as editing the title, description etc.

There is a presumption that a user can only delete a task. In order to future-proof for if superuser permissions were to be implemented, the user_id is required in this table rather than grabbing it directly from the task.

3. Must make use of Black code formatting

Presumptions

- Black should be included in the required packages
- Can manually run black ./ before commiting.
- Default settings are fine

Out of Scope

- Commit hooks
- Pipeline linting

For a real production system, I would implement a git commit hook to check this before pushing code and also have a pipeline stage to check the formatting once pushed up to git.

4. Use of a lightweight framework, database and database migrations

I decided to use FastAPI with a sqlite database alongside sqlalchemy and alembic to manage database migrations.

Justifications

- FastAPI is a lightweight, fast microservice framework for python which I am already familiar with.
- Sqlite seemed to be the easiest to set up with sqlalchemy and is simpler in general to use especially for small projects like this. For a production system I would rather use postgres as it is faster and allows for user management where sqlite doesn't.
- Sqlalchemy as it is a nice, easy to implement interface to interact with the database
- Alembic is the standard database migration management tool to use with sqlalchemy so using this was less of a question.

5. Bonus items:

- Sub task support

  Did not implement, but could implement by including an additional 'parent task' field on the Task perhaps if it were to have identical fields.

- Restoring of deleted tasks

  Did implement. It sets the value of a field ‘deleted’ on the task to be false and puts an entry into the history table to log the action, user_id and date it was updated.

- Setting due dates for tasks

  Added an extra ‘due_date’ field to the task which is optional. The owner can set this on creation or update it with the put endpoint later on.

- Adding labels for tasks

  Did not implement but could do so by adding an additional table for labels with an id and title field. Each label would be unique, and could be referenced via a task-label linking table.

6. Additional Items

- Added poetry

  Requirements.txt would be perfectly reasonable as well but I am currently familiar with poetry and like how easy it is to add requirements to it, with the added benefit of it managing pyenv for you.

- Add some sample data on startup

  Rather than use .create_all() on the models in main.py I have used a lifespan method to run the database migrations which include adding in some sample data.