If we do this project exactly as planned, you’ll have **answers for nearly every FastAPI + backend system design interview question**.  
The only extra thing you might want is **a second smaller project** (microservice) for variety — so you can show you’ve done both monolith-style APIs and microservices.

# Day 1: FastAPI Project Setup + Async PostgreSQL + Dependency Injection + Basic CRUD

### What you will build and learn today:

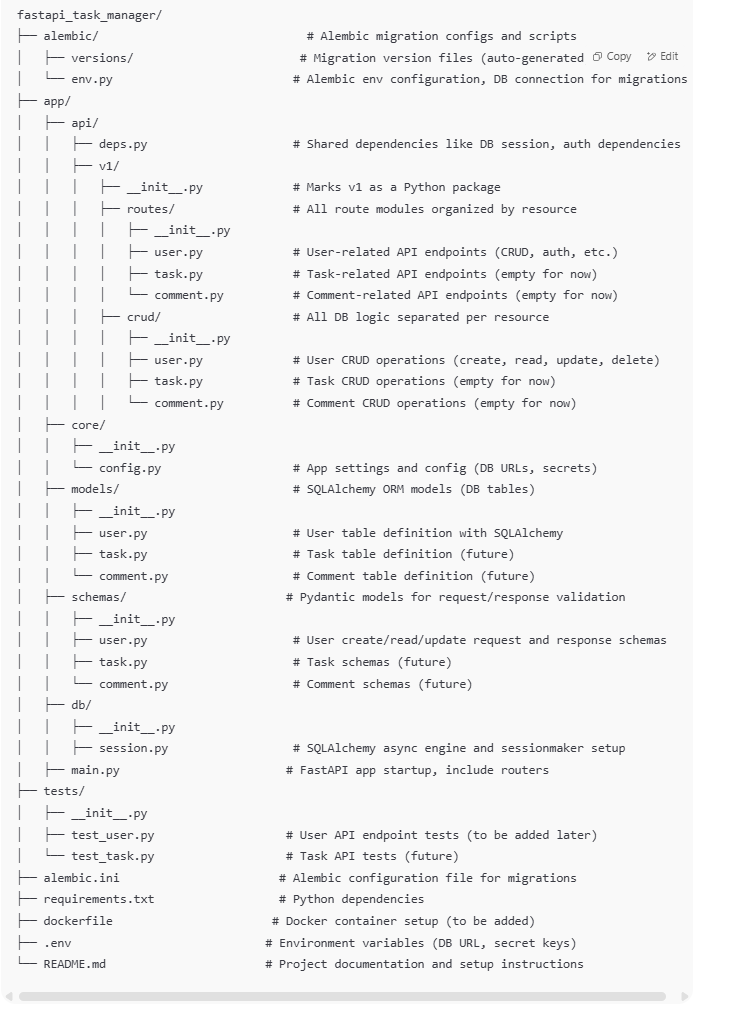
* Initialize a FastAPI project with best practices
* Configure async SQLAlchemy with PostgreSQL
* Use Alembic for database migrations (setup only, migrations on Day 2)
* Implement Dependency Injection (Depends) for DB sessions
* Create Pydantic models for request validation and response serialization
* Build first async CRUD endpoint (Create + Read) using DI
* Run FastAPI with Uvicorn
* Understand and implement production-ready folder structure
* Write clean, modular code with clear separation of concerns

### Project skills/topics covered (for your learning notes):

* **FastAPI app structure & setup**
* **Async SQLAlchemy** with PostgreSQL (asyncpg driver) — Data Access Layer
* **Dependency Injection** pattern in FastAPI
* **Pydantic validation** (request/response)
* **Basic CRUD operations** with async DB
* **Alembic setup** (migration tooling)
* **Running and testing app** locally
* **Code modularity and clean architecture**

**Summary of relationships:**

1. **User**
   * Has many **tasks**
   * Has many **comments**
2. **Task**
   * Belongs to one **user** (owner)
   * Has many **comments**
3. **Comment**
   * Belongs to one **task**
   * Belongs to one **user** (author)

****

**Why this structure?**

* **api/v1/routes/**: Organizes endpoints logically by resource (user.py, task.py, etc.)
* **api/v1/crud/**: Keeps DB logic separate for clean code and easy testing
* **api/deps.py**: Shared dependencies (DB sessions, authentication) accessible across all API versions
* **models/**: Represents DB schema with SQLAlchemy ORM classes only — no business logic
* **schemas/**: Defines data shapes for input/output validation with Pydantic (keeps API contracts clear)
* **db/session.py**: Centralized DB connection and async session setup

1. **Why do we have below files in both models/ and schemas/ folders?**

* user.py
* task.py
* comment.py
* \_\_init\_\_.py

Because these folders represent **different layers of the same domain entities**, and each layer has a specific responsibility:

| **Folder** | **Purpose** | **What goes inside those files (user.py, etc.)** |
| --- | --- | --- |
| **models/** | Defines the **database layer** — how data is stored and structured in the database (SQLAlchemy ORM classes). | Classes representing database tables with columns and relationships, e.g., User, Task, Comment as ORM models. |
| **schemas/** | Defines the **data validation and serialization layer** — how data is accepted from clients and returned in API responses (Pydantic models). | Classes defining the shape of input data (requests) and output data (responses), e.g., UserCreate, UserRead, TaskCreate, etc. |

**What about \_\_init\_\_.py?**

* The \_\_init\_\_.py files make these folders Python packages.
* They allow easier imports, e.g., importing all schemas or models from the folder in a clean way.
* You can also use \_\_init\_\_.py to expose certain classes or functions as a public API of the package.

**Why split by entity inside these folders?**

* It keeps code **organized and modular** by entity/domain (user, task, comment).
* Easy to **navigate, maintain, and test** each part of your app separately.
* Each entity’s models and schemas are **grouped logically** rather than lumping everything into a single file.

**Summary**

| **Folder** | **user.py, task.py, comment.py contain:** |
| --- | --- |
| **models/** | ORM models for tables and relationships (database representation) |
| **schemas/** | Pydantic models for API input/output validation and serialization |

1. **Why keep v1/ inside api/?**

**Versioning:** You might release **v2** of your API in the future with breaking changes but still keep **v1** running for older clients.

Pending

**1) Setup project environment**

pip install fastapi[all] sqlalchemy[asyncio] asyncpg alembic uvicorn python-dotenv passlib[bcrypt] PyJWT httpx

pip freeze > requirements.txt

**11) Run the app**

uvicorn app.main:app --reload

## Interview Questions & Answers (related to Day 1)

1. **Q: Why use Dependency Injection (Depends) for DB session?**  
   A: It cleanly manages resource lifecycle, promotes modularity and testability, and avoids global state.
2. **Q: Why use async SQLAlchemy with asyncpg?**  
   A: Async DB calls prevent blocking the event loop, improving scalability for concurrent clients.
3. **Q: What is the difference between models and schemas?**  
   A: Models map to DB tables; schemas validate and serialize input/output data.
4. **Q: Why separate API routes into versioned folders like v1?**  
   A: Enables backward-compatible API evolution without breaking clients.
5. **Q: What is orm\_mode in Pydantic?**  
   A: Allows Pydantic models to read data directly from ORM objects.

## What You Learned Today

* Created a real-world project structure used in production FastAPI apps
* Configured async DB with PostgreSQL using SQLAlchemy ORM and asyncpg
* Implemented DI for DB sessions and API request validation
* Defined multiple SQLAlchemy models with relationships
* Built Pydantic schemas for clean input/output data contracts
* Developed basic async CRUD endpoints for Users, Tasks, Comments
* Gained clarity on layered architecture and separation of concerns
* Prepared foundational knowledge for database migrations, auth, testing in coming days

**Day 2 Goals**

**JWT-based authentication:**

* User login endpoint that validates credentials and generates JWT token
  + Token creation (create\_access\_token)
* Token verification for protected routes
  + Token verification (get\_current\_user dependency)

**User registration endpoint** (somewhat covered but we can polish it with hashing + validations)

* POST /users

**Protect routes with auth dependency** — i.e. only allow access if user presents valid JWT token

* like user list, user detail, current user) using get\_current\_user

**User login endpoint** (POST /users/login) returning JWT token

**Password hashing securely** (already done hashing on creation, but we’ll also need to verify password during login)

* via passlib in security.py

**Schemas for UserLogin, Token, UserCreate, UserRead** to support above

**Define auth-related schemas:**

* Login input schema (username/email + password)
* Token response schema (access token, token type)
* Possibly token data schema (payload inside JWT)

**Quick checklist for Day 2 completion:**

* security.py with get\_password\_hash, verify\_password, create\_access\_token, authenticate\_user
* api/deps.py with get\_current\_user decoding and verifying JWT token
* schemas/user.py with login and token schemas defined
* api/v1/routes/user.py with registration, login, protected routes, /me endpoint
* Proper error handling and HTTP status codes (401, 400, 404)
* Testing of login and protected routes (using JWT tokens in headers)

**What’s still pending for Day 2:**

1. **Test cases** for:
   * User registration
   * User login
   * Accessing protected endpoint with valid token
   * Accessing protected endpoint with invalid/expired token

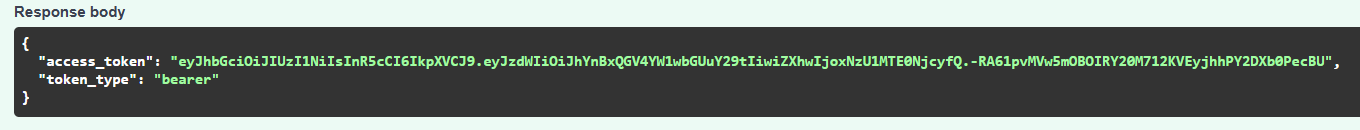
**Checklist to test your FastAPI app using Swagger UI** (<http://127.0.0.1:8002/docs>):

**1. User Registration (POST /api/v1/users/)**

* Test creating a new user with a unique email.
* Expect: 201 Created, response with user details (without password).
* Try creating the same user again, expect 400 error (email already registered).

**2. User Login (POST /api/v1/users/login)**

* Use the registered user's email and password.
* Expect: 200 OK, JSON response with access\_token and token\_type.



* Try wrong password/email, expect 401 Unauthorized.

**3. Get Current User Info (GET /api/v1/users/me)**

* Use the JWT token from login in the **Authorize** button (enter Bearer <token>).
* Expect: 200 OK with current user info.
* Try without token, expect 401 Unauthorized.

**4. Get All Users (GET /api/v1/users/)**

* Protected route: Use the token in Authorize.
* Expect: List of all users.
* Without token, expect 401 Unauthorized.

**5. Get User by ID (GET /api/v1/users/{user\_id})**

* Provide an existing user ID.
* Expect: User details.
* Provide a non-existent ID.
* Expect: 404 Not Found.

**6. Tasks and Comments endpoints (similar to users)**

* If implemented, test:
  + Creating tasks/comments
  + Fetching list, fetching by ID
  + Using JWT for protected routes
* Expect appropriate success or error responses.

**7. Auth endpoints (if separate)**

* If you have any separate /api/v1/auth/\* endpoints, test login, token refresh, etc.

**How to test in Swagger UI:**

* Click **Authorize** top right, enter token as Bearer your\_jwt\_token.
* Use the forms for each endpoint.
* Inspect request/response, status codes, and JSON data.

# Recap of files created/modified:

|  |  |
| --- | --- |
| app/core/config.py | JWT secret, expiry config |
| app/core/security.py | JWT creation and verification utils |
| app/schemas/auth.py | Auth-related Pydantic models |
| app/api/v1/crud/user.py | Authenticate user function |
| app/api/v1/routes/auth.py | Login route that returns JWT |
| app/api/deps.py | Auth dependency to verify JWT and user |
| app/api/v1/routes/user.py | Protected example route /users/me |

# What you learn in Day 2:

* How to write an auth login endpoint
* Pydantic models for auth input/output
* Config management for secrets and expiry
* How to securely hash and verify passwords (using passlib bcrypt)
* How to implement JWT authentication (create and verify tokens)
* How to secure FastAPI routes with dependencies (Depends(get\_current\_user))
* How to build login and registration endpoints with appropriate schemas
* How to protect sensitive endpoints, requiring valid authentication tokens
* How to structure code to separate concerns (crud, schemas, deps, routes, security)

# Sample interview Q&A:

**Q:** How does JWT authentication work in FastAPI?  
**A:** User logs in with credentials; if valid, server returns a signed JWT token. This token is sent with subsequent requests in Authorization header. The server verifies the token, extracts user info, and grants access.

**Q:** Why hash passwords and how?  
**A:** Passwords are hashed with strong algorithms like bcrypt so plain passwords are never stored. This protects user data even if DB is compromised.

**Q:** How do you protect routes in FastAPI?  
**A:** Use dependencies that verify JWT tokens and load user info before executing route logic. Unauthorized access returns 401.

**Q1: How do you securely store passwords?**  
A1: Use a strong hashing algorithm like bcrypt through passlib to hash passwords before storing them. Never store plain text passwords.

**Q2: What is JWT and why use it?**  
A2: JWT (JSON Web Token) is a compact, self-contained token format that carries claims and can be cryptographically signed. It’s used to securely transmit user identity in stateless authentication systems.

**Q3: How do you protect routes in FastAPI that require user authentication?**  
A3: Use FastAPI dependencies to verify tokens before route execution, e.g., a get\_current\_user function that decodes JWT, validates it, and fetches the user or raises a 401 error.

**Q4: How do you implement token expiration with JWT?**  
A4: When creating the JWT token, embed an expiration claim (exp) with the expiry time. During verification, tokens past expiry are rejected.

**Q5: How do you handle authentication errors in FastAPI?**  
A5: Raise HTTPException with status code 401 and appropriate error detail. Include WWW-Authenticate header to signal to clients that authentication is required.

**Industry-Standard Steps for Database & Table Setup**

**1. Create the Database Manually (Once)**

* You **create the PostgreSQL database manually** outside your app.
* This is usually done by a DBA or DevOps or by you during local dev.
* Tools:
  + psql CLI:

createdb mydatabase

* + PgAdmin UI or any SQL client
  + Docker compose with PostgreSQL service (automated DB creation)

**Why?**  
SQLAlchemy *only* manages tables & schema inside the database — it cannot create or manage the database itself. So this is a prerequisite.

**2. Manage Schema (Tables) Via Migrations**

* Instead of manually writing table creation code, **use a migration tool** like **Alembic** (officially recommended by SQLAlchemy).
* Alembic keeps track of database schema versions, supports upgrading/downgrading schema, and works great for teams and production environments.
* This means:
  + You define your models with SQLAlchemy ORM.
  + You generate migrations (alembic revision --autogenerate).
  + You apply migrations to update your database schema (alembic upgrade head).

**Why?**

* Avoids manual SQL scripts and error-prone manual table creation.
* Keeps DB schema versioned and consistent across dev, staging, prod.
* Easy to roll back changes if needed.

**3. Optionally: Automate Table Creation for Development**

* For quick local development or prototyping, you *can* call SQLAlchemy’s Base.metadata.create\_all(bind=engine) to create tables automatically.
* This is **not recommended for production**, because:
  + It does not handle schema changes or migrations well.
  + It won’t drop or modify tables safely.
* Use this only as a fallback or initial setup in local dev.

**4. Configure Your App to Connect to the Database**

* Your app connects to the existing PostgreSQL database with DATABASE\_URL.
* Your app code uses async SQLAlchemy sessions for queries, inserts, updates.
* Ensure connection pooling, retries, and error handling are configured properly.

## What should you do now?

* **Create your PostgreSQL database (e.g. taskdb) manually** on your local machine or dev server.
* **Set up Alembic in your project** to manage migrations.
* Use Alembic to generate and apply migrations for creating tables.
* Run your FastAPI app to connect and interact with the DB.
* Optionally, use Base.metadata.create\_all() for quick local setup but rely on migrations long term.

1. pgAdmin 4🡪 user: postgres, pwd: love vala 🡪 database created as taskdb
2. Create your tables (schemas) - You can create tables either:

* **Manually using SQL** in pgAdmin or any SQL client
* **Automatically via SQLAlchemy models + migrations** (recommended for maintainability)

\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**Checking if postgresql server is running**

On windows command prompt, type **psql –U postgres**

Enter password and if you see postgres= # prompt then it means your postgres server is running

\*\*\*\*\*\*\*\*\*\*\*\*\*\*

## Step 1: Define SQLAlchemy Models (Your DB Tables in Python)

e.g. app/models/user.py

## Step 2: Configure Database URL

In your app/core/config.py

## Step 3: Setup SQLAlchemy Async Engine & Session

In app/db/session.py:

## Step 4: Install and Initialize Alembic (DB migrations)

pip install alembic

## Step 5: Configure Alembic to connect to your PostgreSQL

Open alembic.ini and set:

sqlalchemy.url = postgresql+asyncpg://postgres:yourpassword@localhost:5432/taskdb

Then open alembic/env.py and modify it to use **async** engine with your models:

## Step 6: Create Alembic Migration Script (Auto-generate Tables)

Run this command to create a migration script from your models:

alembic revision --autogenerate -m "Create users table"

Alembic will scan your Base.metadata and detect the users table and create SQL to make it.

*Also, you have to do for other 3 tables too (i.e. comments, tasks and audit\_logs)*

*alembic revision --autogenerate -m "Add comments, tasks, audit\_logs tables"*

## *Step 7: Apply Migration to Create Tables in PostgreSQL*

alembic upgrade head

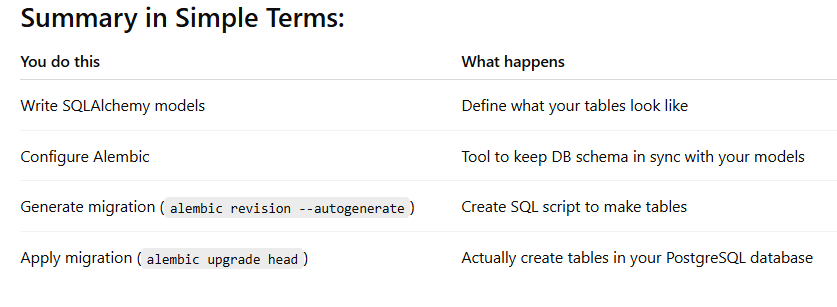
This will apply the migration and **create your users table** in the taskdb database.

## Step 8: Verify Tables Created

* Open pgAdmin or psql
* Check your taskdb database for the users table

## Step 9: Use the Tables in Your FastAPI Code

Now your FastAPI app can **read/write** to the tables using SQLAlchemy sessions.



NB:

**If you want to keep data**  
Don’t drop tables. Instead, let Alembic detect changes and add missing columns/defaults.

alembic revision --autogenerate -m "Sync all tables with models"

Then inspect the migration script Alembic generates — check for:

* ALTER TABLE tasks ADD COLUMN created\_at... with default now()
* Other table changes (users, comments, etc.)

Finally:

alembic upgrade head

**Day 3 Plan – API Expansion & Security**

1. **Add Authentication & Authorization**

* Implement **JWT-based login** (/auth/login).
* Create a get\_current\_user dependency that verifies the JWT.
* Protect certain endpoints (like POST /tasks) so only logged-in users can create.

1. **Seed Data for Testing**

* Add a script or endpoint to create a test user (so you can log in without manual DB inserts).
* Ensure POST /tasks works after authentication.

1. **Implement CRUD for Comments**

* POST /comments → Add a comment to a task (must be logged in).
* GET /comments/{task\_id} → Fetch comments for a given task.

1. **Add Relationships in Schemas**

* Make TaskRead include its comments list.
* Make CommentRead include the task’s title.

1. **Testing**

* Use **FastAPI’s TestClient** to write a couple of tests for:
  + Login
  + Creating a task
  + Adding a comment to that task

If we follow this, by end of Day 3 you’ll have:  
✅ Login system  
✅ Protected routes  
✅ Linked task–comment data  
✅ Basic automated tests

**Test your flow**

1. **Register a user** (/users/) → hash password stored.
2. **Login** (/auth/login) → get JWT token.
3. **Create task** (/tasks/ with Authorization: Bearer <token>).
4. **Get tasks** (/tasks/ with token) → only returns your tasks.
5. **Get task by ID** (/tasks/{id} with token).

Once these are verified, your **Day 3 tasks (JWT auth + task CRUD)** will be fully functional.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*



### ****File Responsibilities****

#### security.py (Core Security Utilities)

* **No DB calls here** — only crypto/token logic.
* verify\_password() → Compare plain & hashed passwords.
* get\_password\_hash() → Hash new passwords.
* create\_access\_token() → Sign JWT with expiry.

#### crud/user.py (Data Access Layer)

* Talks to the database via AsyncSession.
* get\_user\_by\_email() → Select user by email.
* create\_user() → Insert new user with hashed password.
* authenticate\_user() → Combines DB query + verify\_password() from security.py.

#### routes/user.py (API Layer)

* Receives HTTP requests.
* Calls CRUD functions to interact with DB.
* Calls create\_access\_token() from security.py for JWT generation.
* Handles HTTP errors (HTTPException) and status codes.

**Reality in companies / industry standard:**

* **APIs usually use JWT or OAuth2 tokens.**
* **Clients don’t send username/password every time.** They call a login endpoint once to get a token and then use the token for all requests.
* **Swagger UI is just a dev/test convenience.** The “Authorize” button in Swagger is **not part of the API itself** — it’s just there to make testing easier.

So in a real company:

* Your **API code** just needs to support token-based auth (Depends(oauth2\_scheme) in FastAPI).
* How Swagger prompts for the token is **not required** — it’s only for you as a developer/tester.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**Day 5:** External API integration for task metadata, Celery background jobs, ~~SendGrid emails.~~

1. **Set up Celery with Redis broker** in your project.
2. **Add a Celery worker** to run background jobs.
3. **Define a background task** → fetch metadata from an external API (example: placeholder API like https://jsonplaceholder.typicode.com/todos/{id}).
4. **Integrate background job into your routes** → when a task is created, kick off a background job that fetches extra info.
5. **Store the fetched metadata in DB** (we’ll extend your Task model to have a metadata column).

Here’s the **Celery + external API flow** mapped to your current FastAPI project structure with filenames:

 **User creates a task**

* Endpoint: POST /tasks/
* Your app/api/v1/routes/task.py creates the task via crud\_task.create\_task() and stores it in the database.

 **Background job is triggered**

* After creation, the route calls:
* fetch\_and\_update\_task.delay(task.id)
* This sends a **Celery task** to the worker queue without blocking the request.

 **Celery worker runs the job**

* The worker executes fetch\_external\_data\_and\_update(task\_id) in the background.
* It calls an **external API** to fetch additional metadata.

 **Update database with enriched data**

* Once data is fetched and processed, the worker calls:
* await crud\_task.update\_task\_metadata(db, task\_id, metadata)
* This writes the external API data back into the DB.

 **User reads the enriched task**

* Endpoint: GET /tasks/{id}
* Your TaskRead schema returns all task fields, including the enriched metadata added by the Celery worker.

✅ This shows **which file executes each step**:

| **Step** | **Action** | **File** |
| --- | --- | --- |
| 1 | User creates task | routes/task.py |
| 2 | Celery worker fetches API data | worker/celery\_app.py |
| 3 | Update DB with metadata | crud/task.py |
| 4 | User reads enriched task | routes/task.py |