Dream Journal — FastAPI Backend (SOLID)

A complete, modular FastAPI backend for a Dream Journal app: users can register/login, create/update/delete dreams, upload images (or attach AI-generated art), like/unlike, and comment. The code is structured into **repositories** (pure data access), **services** (business rules), and **routers** (HTTP layer) with **dependency injection**.

> This document contains runnable code snippets for all files. Copy them into a project folder matching the tree below, install requirements, run Alembic migrations, and start the server.

🗂️ Folder Structure

```
app/
├─ core/                # Config, security, utils
├─ db/                  # Engine, session, base
├─ models/              # SQLAlchemy models
├─ schemas/             # Pydantic schemas
├─ repositories/        # Data access layer (CRUD only)
├─ services/            # Business logic (uses repositories)
├─ api/
│  ├─ deps.py           # FastAPI dependencies
│  └─ v1/
│     ├─ auth.py
│     ├─ dreams.py
│     ├─ comments.py
│     └─ likes.py
├─ main.py              # FastAPI app factory
└─ __init__.py
```

> **SOLID Mapping** * **S**RP: Each file/class has a single purpose (e.g., `DreamRepository` only talks to DB, `DreamService` only holds dream business rules). * **O**CP: New features (e.g., notifications) can be added by new services/routers without editing existing classes. * **L**SP: Service interfaces return schema types; substitutable implementations (e.g., switch repository to cache-backed) won't break callers. * **I**SP: Small, focused repositories/services rather than a mega-interface. * **D**IP: Routers depend on **abstractions** (interfaces via type hints) injected via FastAPI deps (`get_db`, `get_current_user`).

🔧 `requirements.txt`

```
alembic==1.13.2
fastapi==0.111.0
python-dotenv==1.0.1
pydantic==2.8.2
```

```
pydantic-settings==2.4.0
SQLAlchemy==2.0.34
uvicorn[standard]==0.30.5
passlib[bcrypt]==1.7.4
PyJWT==2.9.0
python-multipart==0.0.9
```

🔐 `.env.example`

```
APP_NAME=DreamJournalAPI
ENV=dev
API_V1_STR=/api/v1
SECRET_KEY=change-me-super-secret
ACCESS_TOKEN_EXPIRE_MINUTES=60
ALGORITHM=HS256
SQLALCHEMY_DATABASE_URI=sqlite:///./dreams.db
MEDIA_DIR=./media
```

Copy to `.env` and edit. For PostgreSQL: `postgresql+psycopg2://user:pass@localhost:5432/dreams` (install `psycopg2-binary`).

`app/core/config.py`

```python
"""Application settings using pydantic-settings.
- SRP: Only configuration responsibility.
- DIP: Other layers depend on this abstraction, not env access directly.
"""
from pydantic_settings import BaseSettings
from pydantic import Field

class Settings(BaseSettings):
    app_name: str = Field(default="DreamJournalAPI")
    env: str = Field(default="dev")
    api_v1_str: str = Field(default="/api/v1")

    secret_key: str
    algorithm: str = Field(default="HS256")
    access_token_expire_minutes: int = Field(default=60)

    sqlalchemy_database_uri: str
    media_dir: str = Field(default="./media")

    class Config:
        env_file = ".env"
```

```
settings = Settings()
```

🔒 app/core/security.py

```
"""Security helpers for hashing and JWT tokens.
- SRP: Crypto + token utilities only.
- OCP: Algorithms/claims extendable without modifying users of this module.
"""
from datetime import datetime, timedelta, timezone
from typing import Any, Optional
import jwt
from passlib.context import CryptContext
from app.core.config import settings

pwd_context = CryptContext(schemes=["bcrypt"], deprecated="auto")


def create_access_token(subject: str | int, expires_minutes: Optional[int] =
None, extra: Optional[dict[str, Any]] = None) -> str:
    expire = datetime.now(tz=timezone.utc) +
timedelta(minutes=expires_minutes or settings.access_token_expire_minutes)
    to_encode: dict[str, Any] = {"sub": str(subject), "exp": expire}
    if extra:
        to_encode.update(extra)
    return jwt.encode(to_encode, settings.secret_key,
algorithm=settings.algorithm)


def verify_password(plain_password: str, hashed_password: str) -> bool:
    return pwd_context.verify(plain_password, hashed_password)


def get_password_hash(password: str) -> str:
    return pwd_context.hash(password)
```

🗄️ app/db/session.py

```
"""SQLAlchemy engine and session factory.
- SRP: DB connectivity.
- DIP: Others receive `Session` via FastAPI dependency.
"""
from sqlalchemy import create_engine
from sqlalchemy.orm import sessionmaker, DeclarativeBase
from app.core.config import settings
```

```
import os

os.makedirs(settings.media_dir, exist_ok=True)

engine = create_engine(
    settings.sqlalchemy_database_uri,
    connect_args={"check_same_thread": False} if
settings.sqlalchemy_database_uri.startswith("sqlite") else {},
)

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

class Base(DeclarativeBase):
    pass
```

🧱 Models — `app/models/user.py`, `dream.py`, `comment.py`, `like.py`

```
# app/models/user.py
from sqlalchemy.orm import Mapped, mapped_column, relationship
from sqlalchemy import String, Integer, DateTime, func
from app.db.session import Base

class User(Base):
    __tablename__ = "users"
    id: Mapped[int] = mapped_column(Integer, primary_key=True, index=True)
    email: Mapped[str] = mapped_column(String(255), unique=True, index=True,
nullable=False)
    username: Mapped[str] = mapped_column(String(50), unique=True,
index=True, nullable=False)
    hashed_password: Mapped[str] = mapped_column(String(255), nullable=False)
    created_at: Mapped[DateTime] = mapped_column(DateTime(timezone=True),
server_default=func.now())

    dreams = relationship("Dream", back_populates="author", cascade="all,
delete-orphan")
    comments = relationship("Comment", back_populates="author",
cascade="all, delete-orphan")
    likes = relationship("Like", back_populates="user", cascade="all, delete-
orphan")
```

```
# app/models/dream.py
from sqlalchemy.orm import Mapped, mapped_column, relationship
from sqlalchemy import Integer, String, Text, ForeignKey, DateTime, func
from app.db.session import Base
```

```
class Dream(Base):
    __tablename__ = "dreams"
    id: Mapped[int] = mapped_column(Integer, primary_key=True, index=True)
    title: Mapped[str] = mapped_column(String(200), nullable=False)
    content: Mapped[str] = mapped_column(Text, nullable=False)
    image_url: Mapped[str | None] = mapped_column(String(500))
    ai_image_url: Mapped[str | None] = mapped_column(String(500))

    author_id: Mapped[int] = mapped_column(ForeignKey("users.id",
ondelete="CASCADE"))
    author = relationship("User", back_populates="dreams")

    created_at: Mapped[DateTime] = mapped_column(DateTime(timezone=True),
server_default=func.now())
    updated_at: Mapped[DateTime] = mapped_column(DateTime(timezone=True),
server_default=func.now(), onupdate=func.now())

    comments = relationship("Comment", back_populates="dream", cascade="all,
delete-orphan")
    likes = relationship("Like", back_populates="dream", cascade="all,
delete-orphan")
```

```
# app/models/comment.py
from sqlalchemy.orm import Mapped, mapped_column, relationship
from sqlalchemy import Integer, Text, ForeignKey, DateTime, func
from app.db.session import Base

class Comment(Base):
    __tablename__ = "comments"
    id: Mapped[int] = mapped_column(Integer, primary_key=True)
    content: Mapped[str] = mapped_column(Text, nullable=False)

    dream_id: Mapped[int] = mapped_column(ForeignKey("dreams.id",
ondelete="CASCADE"))
    author_id: Mapped[int] = mapped_column(ForeignKey("users.id",
ondelete="CASCADE"))

    created_at: Mapped[DateTime] = mapped_column(DateTime(timezone=True),
server_default=func.now())

    dream = relationship("Dream", back_populates="comments")
    author = relationship("User", back_populates="comments")
```

```
# app/models/like.py
from sqlalchemy.orm import Mapped, mapped_column, relationship
from sqlalchemy import Integer, ForeignKey, UniqueConstraint, DateTime, func
from app.db.session import Base

class Like(Base):
```

```
    __tablename__ = "likes"
    __table_args__ = (UniqueConstraint("user_id", "dream_id",
name="uq_user_dream_like"),)

    id: Mapped[int] = mapped_column(Integer, primary_key=True)
    user_id: Mapped[int] = mapped_column(ForeignKey("users.id",
ondelete="CASCADE"))
    dream_id: Mapped[int] = mapped_column(ForeignKey("dreams.id",
ondelete="CASCADE"))
    created_at: Mapped[DateTime] = mapped_column(DateTime(timezone=True),
server_default=func.now())

    user = relationship("User", back_populates="likes")
    dream = relationship("Dream", back_populates="likes")
```

SOLID Note (Models): - **SRP**: Models only define persistence structure; no business logic. - **OCP**: Add new fields/tables without touching services/routers if contracts unchanged.

🗒️ Schemas — `app/schemas/*.py`

```python
# app/schemas/user.py
from pydantic import BaseModel, EmailStr, Field
from datetime import datetime

class UserBase(BaseModel):
    email: EmailStr
    username: str = Field(min_length=3, max_length=50)

class UserCreate(UserBase):
    password: str = Field(min_length=6)

class UserRead(UserBase):
    id: int
    created_at: datetime

    class Config:
        from_attributes = True

class Token(BaseModel):
    access_token: str
    token_type: str = "bearer"
```

```python
# app/schemas/dream.py
from pydantic import BaseModel, Field, HttpUrl
from datetime import datetime
from typing import Optional
```

```python
class DreamBase(BaseModel):
    title: str = Field(min_length=1, max_length=200)
    content: str = Field(min_length=1)

class DreamCreate(DreamBase):
    pass

class DreamUpdate(BaseModel):
    title: Optional[str] = Field(default=None, min_length=1, max_length=200)
    content: Optional[str] = Field(default=None, min_length=1)

class DreamRead(DreamBase):
    id: int
    image_url: Optional[str] = None
    ai_image_url: Optional[str] = None
    author_id: int
    created_at: datetime
    updated_at: datetime

    class Config:
        from_attributes = True
```

```python
# app/schemas/comment.py
from pydantic import BaseModel, Field
from datetime import datetime
from typing import Optional

class CommentBase(BaseModel):
    content: str = Field(min_length=1)

class CommentCreate(CommentBase):
    dream_id: int

class CommentRead(CommentBase):
    id: int
    dream_id: int
    author_id: int
    created_at: datetime

    class Config:
        from_attributes = True
```

```python
# app/schemas/like.py
from pydantic import BaseModel
from datetime import datetime

class LikeRead(BaseModel):
    id: int
    user_id: int
```

```
    dream_id: int
    created_at: datetime

    class Config:
        from_attributes = True
```

SOLID Note (Schemas): - **LSP**: All routers/services return these DTOs; alternate service impls can swap in without breaking type contracts. - **ISP**: Separate schemas for create/update/read duties.

🗃️Repositories — pure CRUD

```
# app/repositories/base.py
"""Base repository with common helpers.
- SRP: Reusable DB helpers only.
- OCP: Extend for new entities without changing this file.
"""
from typing import Generic, TypeVar, Type
from sqlalchemy.orm import Session

T = TypeVar("T")

class Repository(Generic[T]):
    def __init__(self, model: Type[T]):
        self.model = model

    def get(self, db: Session, id: int) -> T | None:
        return db.get(self.model, id)

    def list(self, db: Session, offset: int = 0, limit: int = 50):
        return db.query(self.model).offset(offset).limit(limit).all()

    def add(self, db: Session, obj: T) -> T:
        db.add(obj)
        db.commit()
        db.refresh(obj)
        return obj

    def delete(self, db: Session, obj: T) -> None:
        db.delete(obj)
        db.commit()
```

```
# app/repositories/user_repo.py
from sqlalchemy.orm import Session
from sqlalchemy import select
from app.models.user import User
from app.repositories.base import Repository
```

```
class UserRepository(Repository[User]):
    def __init__(self):
        super().__init__(User)

    def get_by_email(self, db: Session, email: str) -> User | None:
        return db.execute(select(User).where(User.email ==
email)).scalar_one_or_none()

    def get_by_username(self, db: Session, username: str) -> User | None:
        return db.execute(select(User).where(User.username ==
username)).scalar_one_or_none()
```

```
# app/repositories/dream_repo.py
from sqlalchemy.orm import Session
from sqlalchemy import select
from app.models.dream import Dream
from app.repositories.base import Repository

class DreamRepository(Repository[Dream]):
    def __init__(self):
        super().__init__(Dream)

    def list_by_author(self, db: Session, author_id: int, offset: int = 0,
limit: int = 50):
        stmt = select(Dream).where(Dream.author_id ==
author_id).offset(offset).limit(limit)
        return db.execute(stmt).scalars().all()
```

```
# app/repositories/comment_repo.py
from sqlalchemy.orm import Session
from sqlalchemy import select
from app.models.comment import Comment
from app.repositories.base import Repository

class CommentRepository(Repository[Comment]):
    def __init__(self):
        super().__init__(Comment)

    def list_for_dream(self, db: Session, dream_id: int):
        return db.execute(select(Comment).where(Comment.dream_id ==
dream_id)).scalars().all()
```

```
# app/repositories/like_repo.py
from sqlalchemy.orm import Session
from sqlalchemy import select, and_
from app.models.like import Like
from app.repositories.base import Repository
```

```
class LikeRepository(Repository[Like]):
    def __init__(self):
        super().__init__(Like)

    def get_by_user_and_dream(self, db: Session, user_id: int, dream_id: int)
-> Like | None:
        stmt = select(Like).where(and_(Like.user_id == user_id, Like.dream_id
== dream_id))
        return db.execute(stmt).scalar_one_or_none()
```

SOLID Note (Repos): - **SRP**: Only database operations; no auth/validation. - **DIP**: Services depend on repo abstractions, not ORM session specifics.

🧰Services — business logic

```
# app/services/auth_service.py
"""Authentication service.
- SRP: Auth flows only (register/login/token verify).
- DIP: Uses UserRepository abstraction and security helpers.
"""
from sqlalchemy.orm import Session
from fastapi import HTTPException, status
from app.schemas.user import UserCreate
from app.models.user import User
from app.repositories.user_repo import UserRepository
from app.core.security import get_password_hash, verify_password,
create_access_token

class AuthService:
    def __init__(self, users: UserRepository):
        self.users = users

    def register(self, db: Session, data: UserCreate) -> User:
        if self.users.get_by_email(db, data.email):
            raise HTTPException(status_code=400, detail="Email already
registered")
        if self.users.get_by_username(db, data.username):
            raise HTTPException(status_code=400, detail="Username already
taken")
        user = User(email=data.email, username=data.username,
hashed_password=get_password_hash(data.password))
        return self.users.add(db, user)

    def login(self, db: Session, email: str, password: str) -> str:
        user = self.users.get_by_email(db, email)
        if not user or not verify_password(password, user.hashed_password):
            raise HTTPException(status_code=status.HTTP_401_UNAUTHORIZED,
detail="Invalid credentials")
```

```
        return create_access_token(subject=user.id, extra={"username":
user.username})
```

```
# app/services/dream_service.py
from sqlalchemy.orm import Session
from fastapi import HTTPException
from typing import Optional
from app.repositories.dream_repo import DreamRepository
from app.models.dream import Dream
from app.schemas.dream import DreamCreate, DreamUpdate

class DreamService:
    def __init__(self, dreams: DreamRepository):
        self.dreams = dreams

    def create(self, db: Session, author_id: int, data: DreamCreate,
image_url: Optional[str] = None, ai_image_url: Optional[str] = None) ->
Dream:
        dream = Dream(title=data.title, content=data.content,
author_id=author_id, image_url=image_url, ai_image_url=ai_image_url)
        return self.dreams.add(db, dream)

    def update(self, db: Session, dream_id: int, author_id: int, data:
DreamUpdate) -> Dream:
        dream = self.dreams.get(db, dream_id)
        if not dream or dream.author_id != author_id:
            raise HTTPException(status_code=404, detail="Dream not found or
unauthorized")
        if data.title is not None:
            dream.title = data.title
        if data.content is not None:
            dream.content = data.content
        db.commit(); db.refresh(dream)
        return dream

    def delete(self, db: Session, dream_id: int, author_id: int) -> None:
        dream = self.dreams.get(db, dream_id)
        if not dream or dream.author_id != author_id:
            raise HTTPException(status_code=404, detail="Dream not found or
unauthorized")
        self.dreams.delete(db, dream)
```

```
# app/services/comment_service.py
from sqlalchemy.orm import Session
from fastapi import HTTPException
from app.repositories.comment_repo import CommentRepository
from app.models.comment import Comment
from app.schemas.comment import CommentCreate
```

```python
class CommentService:
    def __init__(self, comments: CommentRepository):
        self.comments = comments

    def add(self, db: Session, author_id: int, data: CommentCreate) ->
Comment:
        comment = Comment(content=data.content, dream_id=data.dream_id,
author_id=author_id)
        return self.comments.add(db, comment)

    def delete(self, db: Session, comment_id: int, author_id: int) -> None:
        comment = self.comments.get(db, comment_id)
        if not comment or comment.author_id != author_id:
            raise HTTPException(status_code=404,
detail="Comment not found or unauthorized")
        self.comments.delete(db, comment)
```

```python
# app/services/like_service.py
from sqlalchemy.orm import Session
from app.repositories.like_repo import LikeRepository
from app.models.like import Like

class LikeService:
    def __init__(self, likes: LikeRepository):
        self.likes = likes

    def toggle(self, db: Session, user_id: int, dream_id: int) -> tuple[bool,
int]:
        """Toggle like; returns (is_liked_now, total_count)."""
        existing = self.likes.get_by_user_and_dream(db, user_id, dream_id)
        if existing:
            self.likes.delete(db, existing)
            count = db.query(Like).filter(Like.dream_id == dream_id).count()
            return (False, count)
        like = Like(user_id=user_id, dream_id=dream_id)
        self.likes.add(db, like)
        count = db.query(Like).filter(Like.dream_id == dream_id).count()
        return (True, count)
```

```python
# app/services/image_service.py (optional AI generation stub)
from pathlib import Path
from fastapi import UploadFile
from app.core.config import settings

class ImageService:
    """Handles saving uploads and (optionally) calling AI providers.
    - SRP: Media responsibilities only.
    - OCP: You can subclass to integrate OpenAI/Stability without changing
callers.
```

```
    """
    def save_upload(self, file: UploadFile) -> str:
        dest = Path(settings.media_dir) / file.filename
        with dest.open("wb") as f:
            f.write(file.file.read())
        return str(dest)

    def generate_ai_image(self, prompt: str) -> str:
        """Stub: integrate real provider and return a URL/path."""
        # In a real impl, call external API and save result.
        return ""
```

SOLID Note (Services): - **SRP**: Each service focuses on one domain. - **OCP**: Add new validation flows by extension (e.g., premium dreams) without editing existing. - **DIP**: Services depend on repos; routers depend on services.

🧰API Dependencies — `app/api/deps.py`

```
from typing import Generator
from fastapi import Depends, HTTPException, status
from fastapi.security import OAuth2PasswordBearer
from sqlalchemy.orm import Session
import jwt
from app.db.session import SessionLocal
from app.core.config import settings

reusable_oauth2 = OAuth2PasswordBearer(tokenUrl=f"{settings.api_v1_str}/auth/
login")

def get_db() -> Generator[Session, None, None]:
    db = SessionLocal()
    try:
        yield db
    finally:
        db.close()


def get_current_user_id(token: str = Depends(reusable_oauth2)) -> int:
    try:
        payload = jwt.decode(token, settings.secret_key,
algorithms=[settings.algorithm])
        return int(payload["sub"])
    except Exception:
        raise HTTPException(status_code=status.HTTP_401_UNAUTHORIZED,
detail="Could not validate credentials")
```

📬 Routers — `app/api/v1/auth.py`

```python
from fastapi import APIRouter, Depends, HTTPException, status, Form
from sqlalchemy.orm import Session
from app.api.deps import get_db
from app.services.auth_service import AuthService
from app.repositories.user_repo import UserRepository
from app.schemas.user import UserCreate, UserRead, Token

router = APIRouter(prefix="/auth", tags=["auth"])

@router.post("/register", response_model=UserRead)
def register(data: UserCreate, db: Session = Depends(get_db)):
    service = AuthService(UserRepository())
    return service.register(db, data)

@router.post("/login", response_model=Token)
def login(email: str = Form(...), password: str = Form(...), db: Session =
Depends(get_db)):
    service = AuthService(UserRepository())
    token = service.login(db, email=email, password=password)
    return {"access_token": token, "token_type": "bearer"}
```

📬 Routers — `app/api/v1/dreams.py`

```python
from typing import Optional
from fastapi import APIRouter, Depends, UploadFile, File, Form
from sqlalchemy.orm import Session
from app.api.deps import get_db, get_current_user_id
from app.schemas.dream import DreamCreate, DreamUpdate, DreamRead
from app.services.dream_service import DreamService
from app.services.image_service import ImageService
from app.repositories.dream_repo import DreamRepository

router = APIRouter(prefix="/dreams", tags=["dreams"])

dream_service = DreamService(DreamRepository())
img_service = ImageService()

@router.post("/", response_model=DreamRead)
async def create_dream(
    title: str = Form(...),
    content: str = Form(...),
    image: Optional[UploadFile] = File(None),
    db: Session = Depends(get_db),
    user_id: int = Depends(get_current_user_id),
):
```

```
    image_url = img_service.save_upload(image) if image else None
    data = DreamCreate(title=title, content=content)
    return dream_service.create(db, author_id=user_id, data=data,
image_url=image_url)

@router.get("/", response_model=list[DreamRead])
def list_dreams(db: Session = Depends(get_db)):
    return DreamRepository().list(db)

@router.get("/{dream_id}", response_model=DreamRead)
def get_dream(dream_id: int, db: Session = Depends(get_db)):
    dream = DreamRepository().get(db, dream_id)
    return dream

@router.patch("/{dream_id}", response_model=DreamRead)
def update_dream(dream_id: int, data: DreamUpdate, db: Session =
Depends(get_db), user_id: int = Depends(get_current_user_id)):
    return dream_service.update(db, dream_id, user_id, data)

@router.delete("/{dream_id}", status_code=204)
def delete_dream(dream_id: int, db: Session = Depends(get_db), user_id: int =
Depends(get_current_user_id)):
    dream_service.delete(db, dream_id, user_id)
    return None
```

📫 Routers — `app/api/v1/comments.py`

```
from fastapi import APIRouter, Depends
from sqlalchemy.orm import Session
from app.api.deps import get_db, get_current_user_id
from app.schemas.comment import CommentCreate, CommentRead
from app.services.comment_service import CommentService
from app.repositories.comment_repo import CommentRepository

router = APIRouter(prefix="/comments", tags=["comments"])

comment_service = CommentService(CommentRepository())

@router.post("/", response_model=CommentRead)
def add_comment(data: CommentCreate, db: Session = Depends(get_db), user_id:
int = Depends(get_current_user_id)):
    return comment_service.add(db, user_id, data)

@router.delete("/{comment_id}", status_code=204)
def delete_comment(comment_id: int, db: Session = Depends(get_db), user_id:
int = Depends(get_current_user_id)):
```

```
    comment_service.delete(db, comment_id, user_id)
    return None
```

📬 Routers — `app/api/v1/likes.py`

```
from fastapi import APIRouter, Depends
from sqlalchemy.orm import Session
from app.api.deps import get_db, get_current_user_id
from app.services.like_service import LikeService
from app.repositories.like_repo import LikeRepository

router = APIRouter(prefix="/likes", tags=["likes"])

like_service = LikeService(LikeRepository())

@router.post("/{dream_id}/toggle")
def toggle_like(dream_id: int, db: Session = Depends(get_db), user_id: int =
Depends(get_current_user_id)):
    liked, count = like_service.toggle(db, user_id, dream_id)
    return {"liked": liked, "count": count}
```

🗜️ `app/main.py`

```
"""FastAPI application factory.
- SRP: App setup and router registration only.
- OCP: New routers mountable without changing core code.
"""
from fastapi import FastAPI
from fastapi.middleware.cors import CORSMiddleware
from app.core.config import settings
from app.db.session import engine, Base
from app.api.v1 import auth as auth_router
from app.api.v1 import dreams as dreams_router
from app.api.v1 import comments as comments_router
from app.api.v1 import likes as likes_router

# Create tables if not using Alembic (demo convenience)
Base.metadata.create_all(bind=engine)

app = FastAPI(title=settings.app_name)

app.add_middleware(
    CORSMiddleware,
    allow_origins=["*"], allow_credentials=True,
    allow_methods=["*"], allow_headers=["*"],
```

```
)

api = settings.api_v1_str
app.include_router(auth_router.router, prefix=api)
app.include_router(dreams_router.router, prefix=api)
app.include_router(comments_router.router, prefix=api)
app.include_router(likes_router.router, prefix=api)

@app.get("/")
async def root():
    return {"name": settings.app_name, "version": 1}
```

🗃️Alembic (optional, recommended for prod)

Initialize once:

```
alembic init migrations
```

In `alembic.ini` set `sqlalchemy.url = <from .env>` or read from env. In `env.py`:

```
from app.db.session import Base
from app.models.user import User
from app.models.dream import Dream
from app.models.comment import Comment
from app.models.like import Like
# target_metadata = Base.metadata
```

Generate and apply:

```
alembic revision --autogenerate -m "init"
alembic upgrade head
```

🧪Quick Start

```
python -m venv .venv && source .venv/bin/activate
pip install -r requirements.txt
cp .env.example .env
uvicorn app.main:app --reload
```

Auth flow

- `POST /api/v1/auth/register` body JSON: `{ "email": "a@b.com", "username": "akhil", "password": "secret123" }`

- `POST /api/v1/auth/login` form-data: `email`, `password` → returns `{access_token}`
- Send `Authorization: Bearer <token>` for protected endpoints.

Dreams

- `POST /api/v1/dreams/` multipart form with `title`, `content`, optional `image`.
- `GET /api/v1/dreams/` list.
- `PATCH /api/v1/dreams/{id}` JSON `{title?, content?}`.
- `DELETE /api/v1/dreams/{id}`.

Comments & Likes

- `POST /api/v1/comments/` JSON `{ dream_id, content }`.
- `POST /api/v1/likes/{dream_id}/toggle`.

🪠 How SOLID is enforced (file-by-file)

- **Repositories** (DB only) keep the **S**ingle responsibility clear; services never reach ORM directly.
- **Services** encapsulate business invariants (ownership checks on update/delete, toggle semantics) → callers are **O**pen to add new flows (e.g., moderation) by composing services.
- Returning Pydantic schemas ensures **L**iskov substitution: another `DreamService` implementation can be swapped in (e.g., cached) without breaking routers.
- **I**nterface segregation: small repositories/services avoid god-interfaces.
- **D**ependency inversion: HTTP routers depend on high-level services and dependency providers (`get_db`, token parser) rather than low-level infra.

🔗 Next Extensions (non-breaking)

- AI image provider: implement `ImageService.generate_ai_image()` and add an endpoint `/dreams/{id}/ai-art`.
- Rate limiting: mount a FastAPI dependency at router level (OCP).
- Notifications: add a new `NotificationService` + router.
- Admin routes: separate router with role checks in `AuthService`.

This backend is deliberately minimal yet complete. It cleanly separates concerns and makes it easy to extend without modifying stable components, exemplifying **SOLID** in practice.