

OAuth with FastAPI

write a code of oauth using fastapi + sqlalchemy

Here's a minimal but complete OAuth2 + JWT auth setup using **FastAPI** + **SQLAlchemy**.

I'll split it into files so you can drop it into a project and run.

1. Install dependencies

bash

```
pip install fastapi uvicorn sqlalchemy python-jose[cryptography]
passlib[bcrypt] python-multipart
```

2. database.py

python

```
# database.py
from sqlalchemy import create_engine
from sqlalchemy.orm import sessionmaker, declarative_base

# For demo: SQLite. Replace with your PostgreSQL URL if needed.
# Example for Postgres:
# SQLALCHEMY_DATABASE_URL = "postgresql://user:password@localhost/dbname"
SQLALCHEMY_DATABASE_URL = "sqlite:///./test.db"

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

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

Base = declarative_base()

def get_db():
```

```
db = SessionLocal()
try:
    yield db
finally:
    db.close()
```

3. models.py

```
python

# models.py
from sqlalchemy import Column, Integer, String, Boolean, DateTime, func
from database import Base

class User(Base):
    __tablename__ = "users"

    id = Column(Integer, primary_key=True, index=True)
    username = Column(String(50), unique=True, index=True, nullable=False)
    email = Column(String(100), unique=True, index=True, nullable=False)
    hashed_password = Column(String(255), nullable=False)
    is_active = Column(Boolean, default=True)
    created_at = Column(DateTime(timezone=True),
server_default=func.now())
```

4. schemas.py (Pydantic models)

```
python

# schemas.py
from pydantic import BaseModel, EmailStr

class UserBase(BaseModel):
    username: str
    email: EmailStr

class UserCreate(UserBase):
    password: str

class UserRead(UserBase):
    id: int
    is_active: bool
```

```

class Config:
    from_attributes = True # Pydantic v2 (use orm_mode = True for v1)

class Token(BaseModel):
    access_token: str
    token_type: str

class TokenData(BaseModel):
    username: str | None = None

```

5. security.py (password hashing + JWT + OAuth2PasswordBearer)

```

python

# security.py
from datetime import datetime, timedelta
from typing import Optional

from fastapi import Depends, HTTPException, status
from fastapi.security import OAuth2PasswordBearer
from jose import JWTError, jwt
from passlib.context import CryptContext
from sqlalchemy.orm import Session

from database import get_db
from models import User
from schemas import TokenData

# In production, Load this from env variable
SECRET_KEY = "super-secret-key-change-me"
ALGORITHM = "HS256"
ACCESS_TOKEN_EXPIRE_MINUTES = 30

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

oauth2_scheme = OAuth2PasswordBearer(tokenUrl="token") # POST /token

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)

```

```
def create_access_token(data: dict, expires_delta: Optional[timedelta] = None) -> str:
    to_encode = data.copy()
    expire = datetime.utcnow() + (expires_delta or timedelta(minutes=ACCESS_TOKEN_EXPIRE_MINUTES))
    to_encode.update({"exp": expire})
    encoded_jwt = jwt.encode(to_encode, SECRET_KEY, algorithm=ALGORITHM)
    return encoded_jwt


def get_user_by_username(db: Session, username: str) -> User | None:
    return db.query(User).filter(User.username == username).first()


def authenticate_user(db: Session, username: str, password: str) -> User | None:
    user = get_user_by_username(db, username)
    if not user:
        return None
    if not verify_password(password, user.hashed_password):
        return None
    return user


async def get_current_user(
    token: str = Depends(oauth2_scheme),
    db: Session = Depends(get_db),
) -> User:
    credentials_exception = HTTPException(
        status_code=status.HTTP_401_UNAUTHORIZED,
        detail="Could not validate credentials.",
        headers={"WWW-Authenticate": "Bearer"},
    )

    try:
        payload = jwt.decode(token, SECRET_KEY, algorithms=[ALGORITHM])
        username: str | None = payload.get("sub")
        if username is None:
            raise credentials_exception
        token_data = TokenData(username=username)
    except JWTError:
        raise credentials_exception

    user = get_user_by_username(db, username=token_data.username)
    if user is None:
```

```
        raise credentials_exception
    return user

async def get_current_active_user(
    current_user: User = Depends(get_current_user),
) -> User:
    if not current_user.is_active:
        raise HTTPException(status_code=400, detail="Inactive user")
    return current_user
```

6. main.py (FastAPI app with OAuth endpoints)

python

```
# main.py
from datetime import timedelta

from fastapi import FastAPI, Depends, HTTPException, status
from fastapi.middleware.cors import CORSMiddleware
from fastapi.security import OAuth2PasswordRequestForm
from sqlalchemy.orm import Session

from database import Base, engine, get_db
from models import User
from schemas import UserCreate, UserRead, Token
from security import (
    get_password_hash,
    authenticate_user,
    create_access_token,
    get_current_active_user,
    ACCESS_TOKEN_EXPIRE_MINUTES,
)

# Create DB tables
Base.metadata.create_all(bind=engine)

app = FastAPI(title="FastAPI OAuth2 + SQLAlchemy Example")

# Optional CORS
app.add_middleware(
    CORSMiddleware,
    allow_origins=[ "*"], # change in production
    allow_credentials=True,
    allow_methods=[ "*" ],
```

```
allow_headers=[ "*" ],
)

@app.post("/register", response_model=UserRead,
status_code=status.HTTP_201_CREATED)
def register_user(user_in: UserCreate, db: Session = Depends(get_db)):
    # Check if username or email already exists
    existing_user = (
        db.query(User)
        .filter((User.username == user_in.username) | (User.email ==
user_in.email))
        .first()
    )
    if existing_user:
        raise HTTPException(
            status_code=400,
            detail="Username or email already registered",
        )

    hashed_password = get_password_hash(user_in.password)

    user = User(
        username=user_in.username,
        email=user_in.email,
        hashed_password=hashed_password,
    )
    db.add(user)
    db.commit()
    db.refresh(user)
    return user

@app.post("/token", response_model=Token)
def login_for_access_token(
    form_data: OAuth2PasswordRequestForm = Depends(),
    db: Session = Depends(get_db),
):
    user = authenticate_user(db, form_data.username, form_data.password)
    if not user:
        raise HTTPException(
            status_code=status.HTTP_401_UNAUTHORIZED,
            detail="Incorrect username or password",
            headers={"WWW-Authenticate": "Bearer"},
        )
```

```

access_token_expires = timedelta(minutes=ACCESS_TOKEN_EXPIRE_MINUTES)
access_token = create_access_token(
    data={"sub": user.username},
    expires_delta=access_token_expires,
)
return {"access_token": access_token, "token_type": "bearer"}


@app.get("/me", response_model=UserRead)
async def read_users_me(current_user: User =
Depends(get_current_active_user)):
    return current_user


@app.get("/")
def root():
    return {"message": "OAuth2 + JWT with FastAPI & SQLAlchemy"}

```

7. How to test quickly

Run the app:

```

bash

uvicorn main:app --reload

```

Then:

1. Open docs: <http://127.0.0.1:8000/docs>
2. `POST /register` to create a user.
3. `POST /token` (use form: `username`, `password`) → get `access_token`.
4. Click **Authorize** in Swagger, paste `Bearer <your_token>`.
5. Call `GET /me` → you should see the current user details.

If you tell me your current folder structure / DB (SQLite vs Postgres), I can tweak this to match your exact project layout.

ok, code is working perfectly. now explain me everything i need to know about authenticator in fastapi, i will ask to explain code in next prompt

Nice, glad it worked! 🎉

Let's zoom out and understand **how authentication actually works in FastAPI** so that when we explain the code later, it all clicks.

I'll break it into concepts:

1. Authentication vs Authorization (super important)

- **Authentication** = "Who are you?"
 - Verifying identity (username+password, token, API key, etc.).
- **Authorization** = "What are you allowed to do?"
 - Permissions, roles, scopes (admin vs normal user, etc.).

FastAPI doesn't enforce a specific system; it gives you **tools** (dependencies + security utilities) so *you* define how to authenticate and authorize.

2. How FastAPI handles auth: Dependencies as "guards"

FastAPI's secret weapon = **dependency injection**.

You've already seen:

```
python

async def get_current_user(token: str = Depends(oauth2_scheme), db:
    Session = Depends(get_db)):
    ...
```

Key idea:

- You attach a function like `get_current_user` to a route using `Depends`.
- That function:
 - Extracts credentials (token/header/etc.).
 - Validates them.
 - Returns a `User` object (or raises `HTTPException` if invalid).
- Any route that includes `current_user: User = Depends(get_current_user)` becomes **protected**.

So `get_current_user` acts like your **authenticator**.

3. Security "schemes" in FastAPI

FastAPI has special helpers in `fastapi.security` that know how to extract credentials from the request.

Most common ones:

- `OAuth2PasswordBearer` → reads token from `Authorization: Bearer <token>`
- `OAuth2PasswordRequestForm` → parses `username` & `password` from form data (used in `/token` login endpoint)
- `HTTPBasic` → simple `Authorization: Basic ...` username/password (not JWT-based)
- API key based:
 - `APIKeyHeader` (read from header)
 - `APIKeyQuery` (query param)
 - `APIKeyCookie` (cookie)

In your code, you used:

```
python
```

```
oauth2_scheme = OAuth2PasswordBearer(tokenUrl="token")
```

This does **two things**:

1. Tells FastAPI:
"My auth method is: get a Bearer token from the Authorization header."
2. Integrates with docs (`/docs`): you get the **Authorize** button.

It does **not** validate the token. It only **extracts** it.

Validation is your job (`jwt.decode`, checking user exists, etc.).

4. OAuth2 in FastAPI (password flow + JWT)

FastAPI's official tutorial uses:

- OAuth2 "password" flow for login:
 - Client sends `username + password` → `/token` endpoint.
 - Server returns an `access_token`.
- JWT as the token format.

Flow:

1. User sends credentials to `/token`:

```
http
```

```
POST /token
```

```
Content-Type: application/x-www-form-urlencoded
```

```
username=...&password=...
```

In FastAPI, that's `OAuth2PasswordRequestForm`.

2. Server:

- Verifies username + password (your `authenticate_user` function).
- Creates a JWT token with something like:

```
python
```

```
data = {"sub": user.username, "exp": <time>}
token = jwt.encode(data, SECRET_KEY, algorithm=ALGORITHM)
```

3. Client stores that token and sends it in all future requests:

```
http
```

```
Authorization: Bearer <token>
```

4. Protected routes call `get_current_user()` which:

- Extracts token via `oauth2_scheme`.
- Decodes it with `jwt.decode`.
- Looks up the user in DB.
- Raises 401 if anything fails.

This is the core **OAuth2 + JWT authenticator pattern** in FastAPI.

5. JWT fundamentals (because this is what your “authenticator” really checks)

A JWT is basically:

```
text
```

```
header.payload.signature
```

- **Header** → algo (`HS256`), type (`JWT`).
- **Payload** → claims:
 - `sub` = subject (usually user id/username).
 - `exp` = expiry time.
 - You can add any data: roles, scopes, etc.
- **Signature** = HMAC(secret, header+payload).

Your authenticator (`get_current_user`) does:

- `jwt.decode(token, SECRET_KEY, algorithms=[ALGORITHM])`
- If:

- Signature invalid → token tampered → 401
 - `exp` in past → expired → 401
 - `sub` missing → invalid → 401
-

6. Password hashing & user verification

You **never** store plain passwords.

FastAPI examples use **Passlib + bcrypt**:

- `get_password_hash(password)` → bcrypt hash.
- `verify_password(plain, hashed)` → verifies user's password at login.

So the **authenticator**:

1. Finds user from DB via username.
2. Verifies password using `verify_password`.
3. Returns user if correct, `None` if not.

In your code, this logic is usually inside `authenticate_user`.

7. The `get_current_user` pattern (the real “authenticator”)

This is the core function you'll see in almost every FastAPI auth setup:

```
python

async def get_current_user(
    token: str = Depends(oauth2_scheme),
    db: Session = Depends(get_db),
) -> User:
    # 1. Decode JWT
    # 2. Get username from "sub"
    # 3. Verify user exists in DB
    # 4. Return user or raise 401
```

This function is:

- Used as a **dependency** in routes.
- Central place where:
 - Token is verified.
 - User is fetched.
 - Any extra checks (revoked, banned, etc.) can be added.

You often layer it:

```
python

async def get_current_active_user(
    current_user: User = Depends(get_current_user),
):
    if not current_user.is_active:
        raise HTTPException(400, "Inactive user")
    return current_user
```

Then in routes:

```
python

@app.get("/me")
async def read_me(current_user: User = Depends(get_current_active_user)):
    return current_user
```

8. Authorization: scopes and roles (next level)

Once basic auth works, you might want:

- **Scopes** → like permissions inside the token, e.g.:
 - "scopes": ["items:read", "items:write"]
- **Roles** → stored in DB for user, e.g. `role="admin"`.

FastAPI has:

```
python

from fastapi import Security
from fastapi.security import OAuth2PasswordBearer

oauth2_scheme = OAuth2PasswordBearer(
    tokenUrl="token",
    scopes={"admin": "Admin access", "user": "Regular access"},
)

async def get_current_user(
    security_scopes: SecurityScopes,
    token: str = Depends(oauth2_scheme),
):
    # decode JWT, check scopes in payload against security_scopes.scopes
```

Then in routes:

```
python
```

```
@app.get("/admin")
async def admin_panel(current_user: User = Security(get_current_user,
scopes=["admin"])):
    ...

```

This is how **authorization** builds on top of your authenticator.

9. Authentication vs Middleware

You *could* do auth using middleware, but **FastAPI prefers dependencies** because:

- Per-route control.
- Better integration with OpenAPI/docs.
- Easier composition (`get_current_user` → `get_current_admin_user` etc.).
- Testable and reusable.

So stick with dependency style like you're already doing.

10. Common patterns & pitfalls

Patterns

- Single source of truth:
 - One `get_current_user` function that everyone uses.
- Layered dependencies:
 - `get_current_user` → `get_current_active_user` → `get_admin_user`.
- Short-lived JWTs and maybe refresh tokens (more advanced).

Pitfalls

- Using weak/guessable `SECRET_KEY`.
 - Not setting `exp` → tokens never expire.
 - Storing plaintext passwords.
 - Returning too much user info in token or `/me` route (e.g., `hashed_password`).
 - Forgetting to protect sensitive routes by adding `Depends(get_current_user)`.
-

11. Mental model to keep in your head

When you think "authenticator in FastAPI", think this pipeline:

1. User logs in

→ `/token` endpoint

- `authenticate_user` verifies password
- `create_access_token` returns JWT

2. User calls a protected route

- Sends `Authorization: Bearer <token>`
- `oauth2_scheme` extracts token
- `get_current_user` validates token + loads user
- Route gets `current_user` object
- You decide what they are allowed to do.

Everything is just **dependencies + JWT + DB query**.

If you want, in the next message you can say:

Explain `security.py` line by line
or
Explain login and /me flow step by step

...and I'll map this theory directly to your code.

[explain main.py](#)

Alright, let's tear down `main.py` piece by piece and connect it to the auth concepts we discussed. I'll quote small chunks and explain what exactly they do.

1. Imports

python

```
from datetime import timedelta

from fastapi import FastAPI, Depends, HTTPException, status
from fastapi.middleware.cors import CORSMiddleware
from fastapi.security import OAuth2PasswordRequestForm
from sqlalchemy.orm import Session

from database import Base, engine, get_db
from models import User
from schemas import UserCreate, UserRead, Token
from security import (
    get_password_hash,
    authenticate_user,
    create_access_token,
```

```
    get_current_active_user,  
    ACCESS_TOKEN_EXPIRE_MINUTES,  
)
```

What's going on:

- `FastAPI` – create the app instance.
- `Depends` – use dependency injection (for DB, current user, form data, etc.).
- `HTTPException`, `status` – to return proper error responses.
- `CORSMiddleware` – allow frontends (React, etc.) to call the API from different domains.
- `OAuth2PasswordRequestForm` – parses `username` and `password` from a form in `/token`.

From your own modules:

- `database`:
 - `Base`, `engine` – for creating tables.
 - `get_db` – dependency that gives a SQLAlchemy session (`Session`).
- `models.User` – SQLAlchemy model for users table.
- `schemas` – Pydantic models for request/response shapes:
 - `UserCreate` – input for `/register`.
 - `UserRead` – output for user data.
 - `Token` – shape of `/token` response.
- `security`:
 - `get_password_hash` – hash plain password before saving.
 - `authenticate_user` – verify username + password at login.
 - `create_access_token` – build JWT token.
 - `get_current_active_user` – dependency to fetch logged-in user.
 - `ACCESS_TOKEN_EXPIRE_MINUTES` – token expiry duration.

So `main.py` is just wiring: it uses helpers from other files to expose API endpoints.

2. Create tables

```
python  
  
# Create DB tables  
Base.metadata.create_all(bind=engine)
```

This is a one-liner that tells SQLAlchemy:

“Look at all models inheriting from `Base` (like `User`) and create their tables in the database if they don't already exist.”

- `Base` knows about `User` because `models.py` imported `Base` and defined class `User(Base)`.
- `engine` is connected to your DB (SQLite in example).

In production, you'd usually use migrations (Alembic), but for a mini project this is fine.

3. Create the FastAPI app

```
python
```

```
app = FastAPI(title="FastAPI OAuth2 + SQLAlchemy Example")
```

- Creates the actual FastAPI app object.
- `title` is shown in `/docs` and OpenAPI schema.

Everything below with `@app.get`, `@app.post` attaches routes to this `app`.

4. CORS Middleware

```
python
```

```
app.add_middleware(  
    CORSMiddleware,  
    allow_origins=[ "*"], # change in production  
    allow_credentials=True,  
    allow_methods=[ "*"],  
    allow_headers=[ "*"],  
)
```

- Adds **CORS** (Cross-Origin Resource Sharing) rules.
- This lets a frontend running at another origin (e.g. `http://localhost:3000`) call your API.
- `allow_origins=["*"]` = allow any domain (good for dev, bad for production).
- `allow_methods=["*"]`, `allow_headers=["*"]` – allow all HTTP methods and headers.

This has nothing to do with authentication directly; it's just to avoid browser blocking requests from your frontend.

5. `/register` endpoint – create a new user

```
python
```

```

@app.post("/register", response_model=UserRead,
status_code=status.HTTP_201_CREATED)
def register_user(user_in: UserCreate, db: Session = Depends(get_db)):
    # Check if username or email already exists
    existing_user = (
        db.query(User)
        .filter((User.username == user_in.username) | (User.email ==
user_in.email))
        .first()
    )
    if existing_user:
        raise HTTPException(
            status_code=400,
            detail="Username or email already registered",
        )

    hashed_password = get_password_hash(user_in.password)

    user = User(
        username=user_in.username,
        email=user_in.email,
        hashed_password=hashed_password,
    )
    db.add(user)
    db.commit()
    db.refresh(user)
    return user

```

What this route does conceptually

- **Goal:** Register a new user in the system.
- **Method:** POST /register
- **Body:** JSON as UserCreate :

```

json

{
    "username": "aditya",
    "email": "a@a.com",
    "password": "secret"
}

```

- **Response:** UserRead (e.g., id, username, email, is_active; no password).

Step-by-step:

1. Signature & dependencies

```
python
```

```
def register_user(user_in: UserCreate, db: Session = Depends(get_db)):
```

- `user_in` is automatically validated against `UserCreate` schema.
- `db` is injected by FastAPI via `get_db()` → SQLAlchemy session.

2. Check if user already exists

```
python
```

```
existing_user = (
    db.query(User)
    .filter((User.username == user_in.username) | (User.email ==
user_in.email))
    .first()
)
if existing_user:
    raise HTTPException(
        status_code=400,
        detail="Username or email already registered",
)
```

- Queries DB for any user with same username **or** email.
- If found, raises `400 Bad Request` with a clear message.

3. Hash the password

```
python
```

```
hashed_password = get_password_hash(user_in.password)
```

- Uses `passlib` (inside `get_password_hash`) to generate bcrypt hash.
- You never store `user_in.password` directly.

4. Create and save user

```
python
```

```
user = User(
    username=user_in.username,
    email=user_in.email,
    hashed_password=hashed_password,
)
db.add(user)
db.commit()
db.refresh(user)
return user
```

- Creates a `User` model instance.

- `db.add` → mark to insert.
- `db.commit` → actually writes to DB.
- `db.refresh` → reloads from DB (gets `id`, `created_at`, etc.).
- Returns the user; FastAPI auto-converts to `UserRead` due to `response_model=UserRead`.

This endpoint is **public** (no authentication). Anyone can register.

6. `/token` endpoint – login and get JWT

python

```
@app.post("/token", response_model=Token)
def login_for_access_token(
    form_data: OAuth2PasswordRequestForm = Depends(),
    db: Session = Depends(get_db),
):
    user = authenticate_user(db, form_data.username, form_data.password)
    if not user:
        raise HTTPException(
            status_code=status.HTTP_401_UNAUTHORIZED,
            detail="Incorrect username or password",
            headers={"WWW-Authenticate": "Bearer"},
        )

    access_token_expires = timedelta(minutes=ACCESS_TOKEN_EXPIRE_MINUTES)
    access_token = create_access_token(
        data={"sub": user.username},
        expires_delta=access_token_expires,
    )
    return {"access_token": access_token, "token_type": "bearer"}
```

What this route does conceptually

- **Goal:** Authenticate user and return an access token (JWT).
- **Method:** `POST /token`
- **Body:** form data (not JSON), fields: `username`, `password`.
- **Response:** `Token` (i.e. `{"access_token": "...", "token_type": "bearer"}`).

This is the **OAuth2 password flow token endpoint**.

Step-by-step:

1. Signature & dependencies

python

```
form_data: OAuth2PasswordRequestForm = Depends()
db: Session = Depends(get_db)
```

- `OAuth2PasswordRequestForm` automatically:
 - Reads `username` and `password` from `application/x-www-form-urlencoded` body.
 - Also supports `scope`, `client_id`, etc. (for full OAuth2).
- So, `form_data.username` and `form_data.password` are available.
- `db` is a DB session, as usual.

2. Authenticate user

```
python
```

```
user = authenticate_user(db, form_data.username, form_data.password)
if not user:
    raise HTTPException(
        status_code=status.HTTP_401_UNAUTHORIZED,
        detail="Incorrect username or password",
        headers={"WWW-Authenticate": "Bearer"},
    )
```

Inside `authenticate_user` (from `security.py`):

- Find user by username.
- Verify password with `verify_password` (bcrypt check).
- Return user if valid, `None` if invalid.

If invalid:

- Return `401 Unauthorized`.
- `WWW-Authenticate: Bearer` header is required by some OAuth2 tools and UIs.

3. Create JWT access token

```
python
```

```
access_token_expires = timedelta(minutes=ACCESS_TOKEN_EXPIRE_MINUTES)
access_token = create_access_token(
    data={"sub": user.username},
    expires_delta=access_token_expires,
)
```

- `ACCESS_TOKEN_EXPIRE_MINUTES` = e.g. 30.
- Payload includes `"sub": user.username` → "subject" of token.
- `create_access_token` adds `"exp"` and signs it with your `SECRET_KEY`.

4. Return token

```
python
```

```
return {"access_token": access_token, "token_type": "bearer"}
```

This matches `Token` schema and conforms to what OAuth2 “password flow” expects.
Swagger `/docs` uses this format when you click **Authorize**.

This endpoint is also **public** (no token needed to get a token 😊).

7. `/me` endpoint – get current logged-in user

python

```
@app.get("/me", response_model=UserRead)
async def read_users_me(current_user: User =
Depends(get_current_active_user)):
    return current_user
```

What this route does conceptually

- **Goal:** Return details of the currently authenticated user.
- **Method:** GET `/me`
- **Requires:** `Authorization: Bearer <token>` header.

Step-by-step:

1. Dependency: `get_current_active_user`

python

```
current_user: User = Depends(get_current_active_user)
```

- `get_current_active_user` is defined in `security.py`.
- Internally it calls `get_current_user`, which:
 - Uses `oauth2_scheme` to extract token from header.
 - Decodes JWT and pulls out username (`sub`).
 - Loads user from DB.
- Then `get_current_active_user` checks `user.is_active`.

If:

- No token, invalid token, expired token, or user not found → raises 401.
- User inactive → raises 400 (or whatever you coded).

Only if everything is fine does it return a `User` object.

2. Return current user

python

```
    return current_user
```

FastAPI returns it as `UserRead` due to `response_model=UserRead`.

This endpoint is **protected**. This is your classic “who am I?” endpoint.

Any other protected route will use the same pattern:

```
python
```

```
@app.get("/secure-stuff")
def secure_stuff(current_user: User = Depends(get_current_active_user)):
    ...
```

8. Root endpoint / – health check / welcome

```
python
```

```
@app.get("/")
def root():
    return {"message": "OAuth2 + JWT with FastAPI & SQLAlchemy"}
```

Just a simple route:

- Method: `GET /`
- No auth.
- Useful to quickly check if server is running.

9. Putting it all together – request flow

Register

1. Client → `POST /register` with JSON `username`, `email`, `password`.
2. `main.py`:
 - Hash password.
 - Save user to DB.
 - Return user (without password).

Login

1. Client → `POST /token` with form-encoded `username` & `password`.
2. `main.py`:
 - `authenticate_user` verifies credentials.
 - `create_access_token` generates JWT.

- Returns `{access_token, token_type}`.

Access protected route

1. Client → `GET /me` with header:

```
http
Authorization: Bearer <access_token>
```

2. `main.py`:

- `get_current_active_user` (from `security.py`) is called as a dependency.
- It extracts token, decodes it, gets user from DB.
- If valid, `current_user` is passed into the route.
- `/me` returns current user info.

So `main.py` is basically:

- Defining the **API surface** (endpoints).
 - Connecting requests to the auth logic (`security.py`) and DB (`database.py` / `models.py`).
-

If you want, next we can do a **line-by-line breakdown of `security.py`**, since that's where the real "authenticator brain" lives (JWT, password verification, `get_current_user`, etc.).

[explain `security.py`](#)

Nice, this is the real "brain" of your auth system, so understanding `security.py` deeply will make everything click. I'll walk through it in logical chunks.

I'll assume this is your `security.py`:

```
python

from datetime import datetime, timedelta
from typing import Optional

from fastapi import Depends, HTTPException, status
from fastapi.security import OAuth2PasswordBearer
from jose import JWTError, jwt
from passlib.context import CryptContext
from sqlalchemy.orm import Session

from database import get_db
from models import User
```

```
from schemas import TokenData

# In production, load this from env variable
SECRET_KEY = "super-secret-key-change-me"
ALGORITHM = "HS256"
ACCESS_TOKEN_EXPIRE_MINUTES = 30

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

oauth2_scheme = OAuth2PasswordBearer(tokenUrl="token") # POST /token


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)


def create_access_token(data: dict, expires_delta: Optional[timedelta] = None) -> str:
    to_encode = data.copy()
    expire = datetime.utcnow() + (expires_delta or
timedelta(minutes=ACCESS_TOKEN_EXPIRE_MINUTES))
    to_encode.update({"exp": expire})
    encoded_jwt = jwt.encode(to_encode, SECRET_KEY, algorithm=ALGORITHM)
    return encoded_jwt


def get_user_by_username(db: Session, username: str) -> User | None:
    return db.query(User).filter(User.username == username).first()


def authenticate_user(db: Session, username: str, password: str) -> User | None:
    user = get_user_by_username(db, username)
    if not user:
        return None
    if not verify_password(password, user.hashed_password):
        return None
    return user


async def get_current_user(
    token: str = Depends(oauth2_scheme),
```

```

db: Session = Depends(get_db),
) -> User:
    credentials_exception = HTTPException(
        status_code=status.HTTP_401_UNAUTHORIZED,
        detail="Could not validate credentials.",
        headers={"WWW-Authenticate": "Bearer"},
    )

try:
    payload = jwt.decode(token, SECRET_KEY, algorithms=[ALGORITHM])
    username: str | None = payload.get("sub")
    if username is None:
        raise credentials_exception
    token_data = TokenData(username=username)
except JWTError:
    raise credentials_exception

user = get_user_by_username(db, username=token_data.username)
if user is None:
    raise credentials_exception
return user

async def get_current_active_user(
    current_user: User = Depends(get_current_user),
) -> User:
    if not current_user.is_active:
        raise HTTPException(status_code=400, detail="Inactive user")
    return current_user

```

1. Imports and constants

python

```

from datetime import datetime, timedelta
from typing import Optional

from fastapi import Depends, HTTPException, status
from fastapi.security import OAuth2PasswordBearer
from jose import JWTError, jwt
from passlib.context import CryptContext
from sqlalchemy.orm import Session

from database import get_db

```

```
from models import User
from schemas import TokenData
```

- `datetime`, `timedelta` → used to set JWT expiry.
- `Depends` → makes this file usable as dependencies for routes.
- `HTTPException`, `status` → to return 401, 400, etc.
- `OAuth2PasswordBearer` → FastAPI helper to extract Bearer token from request header.
- `JWTError`, `jwt` → from `python-jose`, used for encoding/decoding JWT.
- `CryptContext` → from Passlib, manages password hashing and verification.
- `Session` → SQLAlchemy session type.
- `get_db` → dependency that provides a DB session.
- `User` → SQLAlchemy user model.
- `TokenData` → Pydantic model with optional `username`, used when decoding tokens.

So this file knows about:

- DB (via `get_db` + `Session` + `User`),
- user schema (`TokenData`),
- and security primitives (JWT, hashing, OAuth2).

2. Security configuration

python

```
# In production, Load this from env variable
SECRET_KEY = "super-secret-key-change-me"
ALGORITHM = "HS256"
ACCESS_TOKEN_EXPIRE_MINUTES = 30
```

- `SECRET_KEY`
 - Used to sign JWT.
 - Anyone with this key can generate valid tokens.
 - In real apps: keep in environment variables (`.env`) and **never commit** to git.
- `ALGORITHM = "HS256"`
 - HMAC-SHA256, symmetric key algorithm.
 - Same `SECRET_KEY` used to sign and verify.
- `ACCESS_TOKEN_EXPIRE_MINUTES = 30`
 - Default validity of an access token.

This block defines **how** your tokens will be constructed and verified.

3. Password hashing context

```
python
```

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

- Creates a hashing “context” with the `bcrypt` algorithm.
- This object:
 - knows how to `hash (.hash())`
 - and how to `verify (.verify())`.

`deprecated="auto"` just means Passlib can manage upgrades to better hashing schemes in the future.

4. OAuth2PasswordBearer – defining how to extract token

```
python
```

```
oauth2_scheme = OAuth2PasswordBearer(tokenUrl="token") # POST /token
```

This is **very important**.

What it does:

- Specifies an **OAuth2 “password flow”** where:
 - the client will get a token from `POST /token`,
 - then send that token as a **Bearer token** in the `Authorization` header.
- It doesn’t validate tokens.

It only:

- Looks at the incoming HTTP request headers.
- Extracts the value of `Authorization: Bearer <token>`.
- Returns that token string when used as a dependency.

Also:

- It integrates with `/docs` so Swagger knows:
 - there’s a security scheme named `OAuth2PasswordBearer`,
 - and it adds the nice **Authorize** button.

You’ll see this used here:

```
python
```

```
token: str = Depends(oauth2_scheme)
```

That’s how you tell FastAPI: “for this endpoint, please read the Bearer token from the header.”

5. Password helper functions

verify_password

python

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

- Takes:
 - plain_password → what user typed at login.
 - hashed_password → what is stored in DB.
- Returns:
 - True if they match.
 - False otherwise.

Internally, `pwd_context.verify`:

- reads the bcrypt hash,
- compares correctly,
- handles salts, etc.

get_password_hash

python

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

- Used when registering a new user or changing password.
- Takes a plain password.
- Returns a bcrypt hash string to store in DB.

So:

- Use `get_password_hash()` when saving.
- Use `verify_password()` when authenticating.

6. JWT creation

python

```
def create_access_token(data: dict, expires_delta: Optional[timedelta] =  
    None) -> str:  
    to_encode = data.copy()  
    expire = datetime.utcnow() + (expires_delta or  
        timedelta(minutes=ACCESS_TOKEN_EXPIRE_MINUTES))
```

```
    to_encode.update({"exp": expire})
    encoded_jwt = jwt.encode(to_encode, SECRET_KEY, algorithm=ALGORITHM)
    return encoded_jwt
```

This function creates your **JWT access token**.

Step-by-step:

1. `data` is usually something like:

```
python
{"sub": user.username}
```

- `sub` = “subject” of token (who it belongs to).

2. Copy it so you don’t mutate caller’s dict:

```
python
to_encode = data.copy()
```

3. Calculate expiry time:

```
python
expire = datetime.utcnow() + (expires_delta or
timedelta(minutes=ACCESS_TOKEN_EXPIRE_MINUTES))
```

- If `expires_delta` is passed in (e.g., 30 minutes), use that.
- Else, use default `ACCESS_TOKEN_EXPIRE_MINUTES`.
- Uses UTC time (good practice).

4. Add `exp` claim:

```
python
to_encode.update({"exp": expire})
```

- JWT spec says `exp` should be a timestamp (library handles conversion).
- When decoding, if current time > `exp` → token is considered expired.

5. Encode the JWT:

```
python
encoded_jwt = jwt.encode(to_encode, SECRET_KEY, algorithm=ALGORITHM)
```

- `python-jose` creates `header.payload.signature`.
- Signed with `SECRET_KEY` using `HS256`.

6. Return the token as a string.

This is used in `main.py` inside `/token` route.

7. DB helper – get user by username

python

```
def get_user_by_username(db: Session, username: str) -> User | None:  
    return db.query(User).filter(User.username == username).first()
```

Simple helper:

- Given the DB session and username,
- Query the `users` table,
- Return the first matching row (or `None`).

This is used in `authenticate_user` and in `get_current_user`.

8. Authenticating username + password

python

```
def authenticate_user(db: Session, username: str, password: str) -> User |  
None:  
    user = get_user_by_username(db, username)  
    if not user:  
        return None  
    if not verify_password(password, user.hashed_password):  
        return None  
    return user
```

This function is called when user logs in.

Flow:

1. Fetch the user:

python

```
user = get_user_by_username(db, username)
```

- If no such user → return `None` (auth fails).

2. Verify password:

python

```
if not verify_password(password, user.hashed_password):
    return None
```

- Compares plain password with hashed password from DB.

3. If both pass:

```
python
```

```
    return user
```

- Caller can then generate a token.

This is intentionally **simple**: it only checks identity, not whether user is active or has roles. That's handled in higher-level functions (like `get_current_active_user`).

9. Getting the current user from the token

```
python
```

```
async def get_current_user(
    token: str = Depends(oauth2_scheme),
    db: Session = Depends(get_db),
) -> User:
    credentials_exception = HTTPException(
        status_code=status.HTTP_401_UNAUTHORIZED,
        detail="Could not validate credentials.",
        headers={"WWW-Authenticate": "Bearer"},
    )

    try:
        payload = jwt.decode(token, SECRET_KEY, algorithms=[ALGORITHM])
        username: str | None = payload.get("sub")
        if username is None:
            raise credentials_exception
        token_data = TokenData(username=username)
    except JWTError:
        raise credentials_exception

    user = get_user_by_username(db, username=token_data.username)
    if user is None:
        raise credentials_exception
    return user
```

This is your **core authenticator**.

Function signature

```
python
```

```
token: str = Depends(oauth2_scheme),  
db: Session = Depends(get_db),
```

- `Depends(oauth2_scheme)` :
 - FastAPI runs `oauth2_scheme(request)` internally,
 - extracts the token from header `Authorization: Bearer <token>`,
 - gives the raw token string to `token` parameter.
- `db` is the DB session.

So this function **always** receives:

- the token from the request,
- and a DB session.

Prepare a reusable 401 exception

```
python
```

```
credentials_exception = HTTPException(  
    status_code=status.HTTP_401_UNAUTHORIZED,  
    detail="Could not validate credentials.",  
    headers={"WWW-Authenticate": "Bearer"},  
)
```

- Standard 401 response required for failed auth.
- `WWW-Authenticate: Bearer` tells the client that it should authenticate via Bearer token.

Decode JWT and extract username

```
python
```

```
try:  
    payload = jwt.decode(token, SECRET_KEY, algorithms=[ALGORITHM])  
    username: str | None = payload.get("sub")  
    if username is None:  
        raise credentials_exception  
    token_data = TokenData(username=username)  
except JWTError:  
    raise credentials_exception
```

- `jwt.decode(...)` :
 - Verifies signature using `SECRET_KEY` and `ALGORITHM`.
 - Checks `exp` (expiry); raises if token is expired or invalid.
- If token is invalid / tampered / expired:

- `JWTError` is thrown → you return 401.

If valid:

- `payload` is a dict with the claims you encoded, e.g.:

```
python
```

```
{"sub": "aditya", "exp": 1733445678}
```

- Get `sub`:

```
python
```

```
username = payload.get("sub")
```

- If `sub` missing → also invalid → 401.
- Wrap it in `TokenData` for structured type (optional, but neat).

Load the user from DB

```
python
```

```
user = get_user_by_username(db, username=token_data.username)
if user is None:
    raise credentials_exception
return user
```

- Use the username from the token to fetch the user.
- If user doesn't exist (e.g., deleted since token was issued):
 - Consider token invalid → 401.
- Otherwise, return the `User` object.

This is what your routes receive when they do:

```
python
```

```
current_user: User = Depends(get_current_user)
```

So `get_current_user` does:

1. Extract token from HTTP headers.
2. Decode & validate JWT.
3. Grab username from token.
4. Load user from DB.
5. Return user object to the endpoint.

10. Enforcing “active” users

```
python
```

```
async def get_current_active_user(
    current_user: User = Depends(get_current_user),
) -> User:
    if not current_user.is_active:
        raise HTTPException(status_code=400, detail="Inactive user")
    return current_user
```

This is a **layered dependency**.

- It depends on `get_current_user`.
 - So it only runs if token is valid and user exists.
- Then it checks:

```
python
```

```
if not current_user.is_active:
    raise HTTPException(status_code=400, detail="Inactive user")
```

- `is_active` is a column in your `User` model.
 - You can deactivate a user in DB (set `is_active=False`).
 - All endpoints using `get_current_active_user` will then reject them.

Any route that wants only active users can now do:

```
python
```

```
@app.get("/me")
async def read_users_me(current_user: User =
Depends(get_current_active_user)):
    return current_user
```

You can add more layers like:

- `get_current_admin_user`
- `get_current_user_with_scope("items:write")`, etc.

11. Big picture: how `security.py` fits into everything

- `verify_password` / `get_password_hash` → handle **password security**.
- `create_access_token` → creates **JWT** after successful login.
- `authenticate_user` → checks **username + password** at login.
- `oauth2_scheme` → tells FastAPI where to read the **token from**.
- `get_current_user` → **core authenticator** for protected routes.
- `get_current_active_user` → adds **extra rule** on top (must be active).

In `main.py`:

- `/register` uses `get_password_hash` to store safe passwords.
 - `/token` uses `authenticate_user` + `create_access_token` to log users in.
 - `/me` uses `get_current_active_user` to return the currently logged-in user.
-

If you want, next I can:

- Show how to add **roles** (like `is_admin`) and make `get_current_admin_user`, or
- Show how to add **scopes** using FastAPI's `Security` and
`OAuth2PasswordBearer(scopes=...)`, or
- Help you plug this into a **real PostgreSQL setup**.