**Hour 1**

**🧠 Day 2 — Hour 1 Notes (Password Hashing & Signup)**

**🔹 1. Why Password Hashing?**

* We never store the **real password** in the database.
* Instead, we store a **hashed** version (an unreadable string).
* Hashing protects users even if the database is leaked.

**🔹 2. What is passlib?**

* passlib is a Python library used for **secure password hashing**.
* It supports many hashing algorithms (bcrypt, sha256, argon2, etc.).
* We used it to **encrypt** passwords before saving them in the database.

**🔹 3. What is CryptContext?**

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

* This creates a **context (setup)** for hashing passwords.
* schemes=["bcrypt"] → means we’ll use **bcrypt algorithm**.
* deprecated="auto" → automatically marks old schemes as deprecated if we change hashing methods later.

**🔹 4. How to Hash a Password**

hash\_password = context.hash(user.password)

✅ This line:

* Takes the plain password (e.g., "Ali123")
* Converts it into a hashed version like:
* $2b$12$hJ7...
* Then you store that hash in the database instead of the real password.

**🔹 5. Common Error — bcrypt version issue**

**Error:**

AttributeError: module 'bcrypt' has no attribute '\_\_about\_\_'

**Reason:**  
You had an **incompatible bcrypt version**.

**Fix:**

pip install bcrypt==4.0.1 passlib==1.7.4 --force-reinstall

**🔹 6. Response Validation Error (id missing)**

**Error:**

ResponseValidationError: Field required: id

**Reason:**  
You returned the **input user model (UserWrite)**, not the one saved to the DB.

**Fix:**  
Return this instead:

return user\_detail

Because user\_detail contains the **auto-generated id** from the database.

**🔹 7. Final Correct Signup Route**

@app.post("/users/signup", response\_model=UserRead)

def register\_user(user: UserWrite, db: Session = Depends(get\_db)):

hash\_password = context.hash(user.password)

user\_detail = Users(name=user.name, email=user.email, password=hash\_password)

db.add(user\_detail)

db.commit()

db.refresh(user\_detail)

return user\_detail

✅ This route:

* Takes the user’s input.
* Hashes their password.
* Saves the user to the database.
* Returns the saved user (excluding the password).

**🔹 8. Example Signup Request**

**POST Request to /users/signup**

{

"name": "Ali",

"email": "Ali@gmail.com",

"password": "Ali123"

}

**Response:**

{

"id": 1,

"name": "Ali",

"email": "Ali@gmail.com"

}

Password is **not returned** — it’s stored as a **hash** in the database.

✅ **Summary**

| **Concept** | **Explanation** |
| --- | --- |
| Password Hashing | Converts plain password into unreadable form. |
| Library Used | passlib with CryptContext. |
| Algorithm | bcrypt. |
| Function | context.hash(password) to hash it. |
| Error Fixed | bcrypt version issue + response model issue. |
| Final Result | Working signup API with secure password storage. |

**Hour**

**🔹 Hour 2 Notes — Login + JWT Token**

**🧩 1. What is JWT**

* **JWT (JSON Web Token)** is used to verify a user’s identity after login.
* Once user logs in → server gives a **token** → user sends this token in every next request.
* The token proves that the user is authenticated.

**🔐 2. Library used**

* We used **PyJWT** (pip install PyJWT) to **create and verify** tokens.
* It helps encode (create) and decode (read/verify) JWT tokens securely.

**⚙️ 3. JWT Main Components**

JWT has three parts:

1. **Header** → algorithm used (e.g., HS256)
2. **Payload** → data inside token (e.g., user email)
3. **Signature** → proof token is real and not modified

Example token:

xxxxx.yyyyy.zzzzz

**🧮 4. How we create a JWT token**

We created this function:

def get\_jwt\_token(data: dict):

to\_encode = data.copy() # copy the user data

expiry = datetime.utcnow() + timedelta(minutes=expiry\_time) # expiry time

to\_encode.update({"exp": expiry}) # add expiry field

encoded\_jwt = jwt.encode(to\_encode, secret\_key, algorithm=Algorithm)

return encoded\_jwt

**Explanation:**

* data.copy() → make a copy of user info.
* datetime.utcnow() → current time.
* timedelta(minutes=expiry\_time) → how long the token will be valid.
* "exp" → expiry claim (when token expires).
* jwt.encode() → creates the actual JWT string.

**🧠 5. Why "sub" key in token**

* "sub" means **subject** — it’s a standard JWT field.
* It usually stores **who the token belongs to** (user’s email or ID).
* Example:
* {"sub": user.email}

**🔑 6. Login Route Logic**

@app.post("/login")

def login(credentials: Login, db: Session = Depends(get\_db)):

user = db.query(Users).filter(Users.email == credentials.email).first()

if not user:

return {"error": "Invalid Credentials"}

if not context.verify(credentials.password, user.password):

return {"error": "Invalid Password"}

access\_token = get\_jwt\_token(data={"sub": user.email})

return {"access\_token": access\_token, "token\_type": "bearer"}

**Step-by-step:**

1. User enters email + password.
2. Server checks if user exists in DB.
3. Password verified using passlib.
4. If correct → create JWT token using get\_jwt\_token().
5. Return token as response.

**⏳ 7. Role of timedelta**

* timedelta(minutes=30) → sets token expiry time.
* Example:  
  If current time is 10:00 → token expires at 10:30.
* After expiry, the token becomes invalid.

**⚠️ 8. Error handling**

* If email not found → return "Invalid Credentials".
* If password wrong → return "Invalid Password".

**✅ 9. What you completed in Hour 2**

* Implemented secure **login route**.
* Added **password verification** using passlib.
* Generated **JWT token** using PyJWT.
* Understood key JWT concepts: sub, exp, timedelta, and token encoding.

Would you like me to write similar **Hour 3 Notes format** after you complete Hour 3 (Protecting Routes with JWT)?

Perfect 👍 Here are your **Hour 3 Notes** — written in **simple language** so you can easily revise later.

**🕒 Day 2 — Hour 3: Protect Routes using JWT (Get Current User)**

**🎯 Goal**

To protect routes so that **only logged-in users** with a valid token can access them.

**🔹 1. What We Did**

* We created a route /users/me to get the **current user** based on the **JWT token**.
* We protected the route using **dependency injection**.
* We verified the token, handled errors, and returned the user’s info from the database.

**🔹 2. Token Dependency**

from fastapi.security import APIKeyHeader

jwt\_token = APIKeyHeader(name="Authorization")

* This line tells FastAPI that the token will come from the **Authorization header**.
* When calling the API, we must send the token like this:
* Bearer <your\_token\_here>

**🔹 3. Protected Route (/users/me)**

@app.get("/users/me")

def get\_me(token=Depends(jwt\_token), db: Session = Depends(get\_db)):

token = token.replace("Bearer ", "") # Remove Bearer prefix

try:

token\_data = jwt.decode(token, secret\_key, algorithms=[Algorithm])

token\_email = token\_data.get("sub")

if token\_email is None:

return {"error": "Invalid Token"}

except ExpiredSignatureError:

return {"error": "Token Expired"}

except InvalidTokenError:

return {"error": "Invalid Token"}

user\_detail = db.query(Users).filter(Users.email == token\_email).first()

return user\_detail

**🔹 4. What Happens in This Code**

1. **Extract Token** → Taken from request header using Depends(jwt\_token).
2. **Remove “Bearer ”** → Because token usually comes like Bearer <token>.
3. **Decode Token** → Using your secret\_key and algorithm.
4. **Get User Email** → From the token (sub claim).
5. **Handle Errors:**
   * ExpiredSignatureError → Token expired.
   * InvalidTokenError → Token is invalid or corrupted.
6. **Get User Info** → Using email from database.
7. **Return User Data**.

**🔹 5. Dependency Injection Concept**

* Depends() tells FastAPI to **automatically provide** that value.
* Example:
  + token=Depends(jwt\_token) → FastAPI extracts token from request headers.
  + db=Depends(get\_db) → FastAPI creates and gives a database session.
* So, you don’t have to manually pass them every time.

**🔹 6. Swagger UI Testing**

* Click **Authorize** button.
* Enter your token like:
* Bearer <token>
* Call /users/me:
  + ✅ Valid token → returns user info.
  + ❌ Expired token → shows “Token Expired”.
  + ❌ Invalid token → shows “Invalid Token”.

**🔹 7. Main Concepts Learned**

* How to **protect routes** using JWT.
* How to **decode and validate** tokens.
* How to **fetch current user info** using token data.
* How to **use dependency injection** for automatic header and DB handling.

Would you like me to now write the **Hour 4 plan** (next topic after this — e.g., refresh tokens or role-based authorization)?