okay it's saying that im not authenticated when i try to do the things on swagger but it's not giving me an area to authenticate? how do i authenticate?

Ah, this is a common thing when using **Swagger / FastAPI docs** with `OAuth2PasswordBearer`. FastAPI doesn't automatically show a login box—you need to **use the "Authorize" button** in the Swagger UI to provide a token. Here's how you can do it:

Step 1: Get a token

1. Go to `/docs`.
2. Use the `/token` endpoint.
3. Click "**Try it out**" and enter your `username` and `password`.
4. Click "**Execute**".
5. You'll get a JSON response like:

```json
{
  "access_token": "eyJhbGciOiJIUzI1NiIsInR5cCI6...",
  "token_type": "bearer"
}
```

Copy the value of `"access_token"`.

Step 2: Authorize in Swagger

1. At the top right of `/docs`, click the "**Authorize**" button (it looks like a lock).
2. In the popup, you'll see a field for **Value**.
3. Paste your token like this:

```nginx
Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6...
```

4. Click **Authorize**, then close the popup.

Step 3: Call protected endpoints

Now any endpoint that uses `Depends(get_current_user)` will include your token automatically, and Swagger will treat you as authenticated.

If you want, I can also show a **way to make the token automatically prefilled in Swagger** so it's easier to test each time. That avoids manually pasting it. Do you want me to do that?

like this?

Exactly! You don't enter your username/password in the "Authorize" box itself—you first **call** `/token` to get the JWT, then paste it into the "Authorize" box **prefixed with** `Bearer` .

So if your `/token` response was:

```json
{
  "access_token": "abc123def456...",
  "token_type": "bearer"
}
```

Then in the Swagger "Authorize" popup, you put:

```nginx
Bearer abc123def456...
```

Once you click **Authorize**, all requests to endpoints that have `Depends(get_current_user)` will automatically include that token, and you'll no longer get the "not authenticated" error.

If you want, I can also show a **quicker way to test protected endpoints without copying the token manually every time**.

Available authorizations

Scopes are used to grant an application different levels of access to data on behalf of the end user. Each API may declare one or more scopes.

API requires the following scopes. Select which ones you want to grant to Swagger UI.

OAuth2PasswordBearer (OAuth2, password)
Token URL: token

Flow: password

username:
password:
Client credentials location:

Authorization header
client_id:
client_secret:

Ah, that's the built-in Swagger UI OAuth2 flow for FastAPI. For your setup, you're using `OAuth2PasswordBearer` with a simple username/password → JWT flow, so you **don't need client_id/client_secret**. Here's what to do:

1. Click the "**Authorize**" button in Swagger.
2. In the popup under **username**, enter the username you registered (e.g., `Geeks`).
3. Under **password**, enter the user's password.
4. Leave **client_id** and **client_secret** empty.
5. Click **Authorize**.

Swagger will then call `/token` behind the scenes, get the JWT, and automatically attach it to requests to protected endpoints (the ones that use `Depends(get_current_user)`).

After that, you can try your `/shelters/` POST or PUT requests—they should now work without giving you the "not authenticated" error.

If you want, I can give a **step-by-step screenshot-style guide for Swagger** to make it completely clear. Do you want me to do that?

okay i logged in but it looks weird now, when i was logged out the drop downs had a unlocked but when i authorized it's locked now

i tried to do a post and i got this error

Server response
Code Details
500
Undocumented
Error: Internal Server Error

Response body
Download
Internal Server Error
Response headers
content-length: 21
content-type: text/plain; charset=utf-8
date: Mon,17 Nov 2025 19:44:09 GMT
server: uvicorn
Responses
Code Description Links
200
Successful Response

Media type

application/json
Controls Accept header.
Example Value
Schema
"string"
No links
422
Validation Error

Media type

application/json
Example Value
Schema


```
{
  "detail": [
    {
      "loc": [
        "string",
        0
      ],
      "msg": "string",
      "type": "string"
    }
  ]
}
```


Ah, that 500 Internal Server Error usually means something went wrong **server-side**—often a bug in your endpoint logic or a mismatch between what Swagger is sending and what your FastAPI endpoint expects.

A few common culprits for your `/shelters/` POST endpoint:

1. **Missing fields / incorrect body format**
   Your `Shelter` model probably expects fields like `name`, `location`, `capacity`. In Swagger, make sure your POST body is **JSON** and matches exactly, e.g.:

   json Copy code

   ```
   {
     "name": "Central Shelter",
     "location": "Downtown",
     "capacity": 50
   }
   ```

2. **Authentication dependency causing issues**
   Since you added `Depends(get_current_user)`, the server will fail if your JWT is missing or invalid. But you said you just authorized, so this is less likely.
3. **Session / DB issue**
   If `get_session` or `session.add()` fails (e.g., DB connection issue), it will throw 500.

How to debug:

- Look at the terminal where you ran `uvicorn main:app --reload`. FastAPI usually prints the full traceback for a 500 error. That will tell you exactly which line failed.
- Make sure your request body matches the `Shelter` model fields exactly.

If you want, I can rewrite your `/shelters/` POST endpoint **with proper error handling** so it never returns 500 blindly, but instead gives a clear message if something is wrong.

Do you want me to do that?

where would a bug be in this?

```
from fastapi import FastAPI, Depends, HTTPException
from sqlmodel import SQLModel, Session, select
from models import Shelter
from database import engine, get_session
from contextlib import asynccontextmanager # added
from auth import verify_password, create_access_token
from fastapi.security import OAuth2PasswordBearer,
OAuth2PasswordRequestForm
from models import User
from jose import jwt, JWTError
from auth import SECRET_KEY, ALGORITHM
from auth import get_password_hash
from pydantic import BaseModel, Field

@asynccontextmanager
async def lifespan(app: FastAPI):
    SQLModel.metadata.create_all(engine)
    yield

#also added
app = FastAPI(lifespan=lifespan)

class UserCreate(BaseModel):
    username: str
    password: str = Field(..., max_length=72)
```

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

@app.post("/token")
def login(form_data: OAuth2PasswordRequestForm =
Depends(), session: Session =
Depends(get_session)):
    user = session.exec(select(User).where(User.username
== form_data.username)).first()
    if not user or not verify_password(form_data.password,
user.hashed_password):
        raise HTTPException(status_code=401, detail="Invalid
credentials")
    token = create_access_token(data={"sub":
user.username})
    return {"access_token": token, "token_type": "bearer"}

def get_current_user(token: str =
Depends(oauth2_scheme), session: Session =
Depends(get_session)):
    try:
        payload = jwt.decode(token, SECRET_KEY,
algorithms=[ALGORITHM])
        username = payload.get("sub")
        user = session.exec(select(User).where(User.username
== username)).first()
        if not user:
            raise HTTPException(status_code=401, detail="User
not found")
        return user
    except JWTError:
        raise HTTPException(status_code=403, detail="Token
invalid")

from auth import get_password_hash

# @app.post("/register")
# def register_user(user: User, session: Session =
Depends(get_session)):
#     # Hash the plain password before saving
#     user.hashed_password =
get_password_hash(user.hashed_password)
```

```
#     session.add(user)
#     session.commit()
#     session.refresh(user)
#     return {"detail": f"User {user.username} registered
successfully"}


"""the newer ver"""
# @app.post("/register")
# def register_user(user: UserCreate, session: Session =
Depends(get_session)):
#     password_to_hash = user.password[:72]  # safety
truncate for bcrypt
#     hashed_password =
get_password_hash(password_to_hash)
#     db_user = User(username=user.username,
hashed_password=hashed_password)
#     session.add(db_user)
#     session.commit()
#     session.refresh(db_user)
#     return {"detail": f"User {db_user.username} registered
successfully"}


@app.post("/register")
def register_user(user: UserCreate, session: Session =
Depends(get_session)):
    # Encode to bytes first, then truncate to 72 bytes
    raw_password_bytes = user.password.encode("utf-8")
[:72]
    hashed_password =
get_password_hash(raw_password_bytes)
    db_user = User(username=user.username,
hashed_password=hashed_password)
    session.add(db_user)
    session.commit()
    session.refresh(db_user)
    return {"detail": f"User {db_user.username} registered
successfully"}



#https://fastapi.tiangolo.com/advanced/events/#lifespan
```

```
# @app.on_event("startup")
# def on_startup():
#     SQLModel.metadata.create_all(engine)

#changed as previous is being depreceated


@app.get("/")
def read_root():
    return {"message": "It is working!"}

@app.post("/shelters/")
def create_shelter(shelter: Shelter, session: Session =
Depends(get_session), user: User =
                Depends(get_current_user)):
    session.add(shelter)
    session.commit()
    session.refresh(shelter)
    return shelter

@app.get("/shelter/{shelter_id}")
def read_shelter(shelter_id: int, session: Session =
Depends(get_session), user: User =
                Depends(get_current_user)):
    shelter = session.get(Shelter, shelter_id)
    if not shelter:
        raise HTTPException(status_code=404, detail="Shelter
not found")
    return shelter

@app.put("/shelters/{shelter_id}")
def update_shelter(shelter_id: int, updated: Shelter, session:
Session =
                   Depends(get_session), user: User =
                Depends(get_current_user)):

    shelter = session.get(Shelter, shelter_id)
    if not shelter:
        raise HTTPException(status_code=404, detail="Shelter
not found")
    shelter.name = updated.name
    shelter.location = updated.location
```

```
    shelter.capacity = updated.capacity
    session.commit()
    session.refresh(shelter)
    return shelter

@app.delete("/shelters/{shelter_id}")
def delete_shelter(shelter_id: int, session: Session =
Depends(get_session), user: User =
            Depends(get_current_user)):
    shelter = session.get(Shelter, shelter_id)
    if not shelter:
        raise HTTPException(status_code=404, detail="Shelter
not found")
    session.delete(shelter)
    session.commit()
    return {"detail": "Shelter deleted"}



#Run: uvicorn main:app --reload
```

Looking at your code, the likely places for a 500 Internal Server Error when trying to do something like a POST to `/shelters/` are:

1. `Shelter` model validation

The endpoint:

python Copy code

```python
@app.post("/shelters/")
def create_shelter(shelter: Shelter, session: Session = Depends(get_session), use
    session.add(shelter)
    session.commit()
    session.refresh(shelter)
    return shelter
```

- FastAPI will **try to validate your request body against the `Shelter` model**.
- If your JSON body is missing required fields, or has the wrong type, FastAPI can throw a 422 **Validation Error**, or your database may throw an error if it cannot store it,

resulting in a 500.

Check: The request body in Swagger/Postman must match exactly the `Shelter` model fields and types.

2. Database session / engine issues

You're using:

```python
session: Session = Depends(get_session)
```

- If `get_session` fails (e.g., engine isn't connected to Postgres), any `session.add()` or `session.commit()` will throw an error.
- Also, if `engine` didn't create tables properly (e.g., `Shelter` table missing), `session.add(shelter)` will fail.

Check: Make sure `SQLModel.metadata.create_all(engine)` is running properly on startup, and your tables exist.

3. `get_current_user` dependency

Since all `/shelters/` endpoints depend on `get_current_user`:

```python
user: User = Depends(get_current_user)
```

- If JWT decoding fails, or the user is not found, `get_current_user` raises a `HTTPException`.
- That usually returns a proper 401 or 403, not a 500. But if `session.exec(...)` somehow fails (DB issue), it could propagate as a 500.

4. Password hashing / JWT

This is mostly only used at `/register` or `/token`, so less likely to affect `/shelters/` directly.

✅ How to pinpoint

1. Run Uvicorn in terminal:

bash Copy code

```
uvicorn main:app --reload
```

2. **Do the POST request** from Swagger or Postman.
3. **Check the terminal traceback** — it will show exactly which line threw the exception. That's the exact bug to fix.

Quick safety checks

- Make sure your JSON matches `Shelter`:

json Copy code

```
{
  "name": "Central Shelter",
  "location": "Downtown",
  "capacity": 50
}
```

- Make sure the database is running and tables are created.
- If you want, I can rewrite your `/shelters/` POST with **try/except** that catches DB errors and returns a readable 400/500 instead of just crashing.

Do you want me to do that?

yes

Perfect — we can wrap your endpoint in a `try/except` so that any database or validation error returns a proper JSON response instead of crashing as a 500. Here's an example for your `create_shelter` endpoint:

python Copy code

```
from sqlalchemy.exc import SQLAlchemyError  # add this import
```

```python
@app.post("/shelters/")
def create_shelter(
    shelter: Shelter,
    session: Session = Depends(get_session),
    user: User = Depends(get_current_user)
):
    try:
        session.add(shelter)
        session.commit()
        session.refresh(shelter)
        return shelter
    except SQLAlchemyError as e:
        session.rollback()  # undo any partial changes
        raise HTTPException(
            status_code=400,
            detail=f"Database error: {str(e.__dict__.get('orig'))}"
        )
    except Exception as e:
        raise HTTPException(
            status_code=500,
            detail=f"Unexpected error: {str(e)}"
        )
```

Explanation:

- `SQLAlchemyError` catches database-related errors (like missing tables, constraints, or type issues).
- `session.rollback()` ensures the session is clean for the next request.
- The generic `Exception` catches anything else unexpected.

You can wrap your other endpoints (`update` , `delete`) in a similar way.

If you want, I can rewrite **all your** `/shelters/` **endpoints with these error handlers** so you can test them safely in Swagger without crashing. Do you want me to do that?

okay so where exactly would i ever use the token?

Ah, now we're getting to the core of OAuth2 with FastAPI! Here's the flow and where the token comes in:

1. Obtain the token

You use the `/token` endpoint to log in with your username and password.

- In Swagger, this is where you enter your credentials.
- The endpoint returns something like:

json Copy code

```
{
  "access_token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9...",
  "token_type": "bearer"
}
```

This `access_token` is your **JWT token**, which proves you are authenticated.

2. Use the token to access protected endpoints

Any endpoint that has:

python Copy code

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

or any `Depends(oauth2_scheme)` is **protected**.

The token goes in the **Authorization header** of your HTTP request:

makefile Copy code

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

For example, if you got the token `abc123`:

makefile Copy code

```
Authorization: Bearer abc123
```

- FastAPI's `get_current_user` reads the token from the header, verifies it, and returns the user.
- If the token is missing or invalid, you'll get a 401 or 403.

3. Swagger UI

Once you click **Authorize** in Swagger:

- You paste the token (or let Swagger request it from `/token`)
- Swagger automatically adds the `Authorization: Bearer ...` header to requests for protected endpoints.
- That's why endpoints appear "locked" after you authorize — Swagger is now sending the token.

✅ **Key point:** You only use the token after logging in; you **don't send your password** to `/shelters/` or other protected endpoints.

If you want, I can **show an example of testing a protected endpoint in Swagger** with your current setup, including what to type where. This usually clears up the "how do I use the token?" confusion. Do you want me to do that?

okay i mean in my code currently, i get the token but where do i put it?

Ah, got it! In your **Python/FastAPI code**, you don't explicitly "put the token" anywhere yourself for your own server endpoints — FastAPI handles it for you through **dependencies**. Let me break it down clearly.

1. Token generation

Your `/token` endpoint returns the JWT token:

python Copy code

```
    token = create_access_token(data={"sub": user.username})
    return {"access_token": token, "token_type": "bearer"}
```

You store this token **client-side** (in a frontend app, Postman, or Swagger).

2. Token usage in protected routes

Look at your `create_shelter` route:

python Copy code

```
@app.post("/shelters/")
def create_shelter(shelter: Shelter, session: Session = Depends(get_session), use
    ...
```

Here's what happens:

1. `Depends(get_current_user)` tells FastAPI:

   "Before running this route, call `get_current_user` to get the current authenticated user."

2. `get_current_user` expects a **token in the Authorization header**:

python Copy code

```
def get_current_user(token: str = Depends(oauth2_scheme), session: Session = Depe
    payload = jwt.decode(token, SECRET_KEY, algorithms=[ALGORITHM])
    username = payload.get("sub")
    ...
```

3. So the token is **not hardcoded in your server**. It's sent **with the request from the client**.

3. How it looks in practice

- **Client** logs in → gets token `abc123`.
- **Client** sends POST to `/shelters/` with:

pgsql Copy code

```
Authorization: Bearer abc123
Content-Type: application/json
{
  "name": "Shelter A",
  "location": "Downtown",
  "capacity": 50
}
```

- FastAPI decodes the token automatically using `get_current_user`.

✅ Summary

- In **your FastAPI code**, the token is never manually inserted — `Depends(oauth2_scheme)` reads it automatically from request headers.
- You only **use the token outside the server**: Swagger, Postman, or a frontend app.

If you want, I can write a **little Python example showing how to call your `/shelters/` route using the token** — so you can test it programmatically without Swagger.

Do you want me to do that?

why isnt the token manually inserted by the user anywhere?

like i want it so they put in the token to access somewhere

Ah! I see exactly what you mean now. You want your **users themselves to manually input a token** in order to access a route, like a "key" they have to type in, rather than the token automatically being handled in the Authorization header.

Right now, FastAPI + `OAuth2PasswordBearer` **expects the token in the request header** (the "Authorization: Bearer <token>"), so the user never sees or types it manually in the code or form — it's automatically read from the request.

If you want the **user to type it somewhere**, you have a few options:

1. Accept token as a query parameter

Instead of using `OAuth2PasswordBearer`, you can require the token as a query param:

python Copy code

```python
from fastapi import Query

def get_current_user(token: str = Query(...), session: Session = Depends(get_sess
    try:
        payload = jwt.decode(token, SECRET_KEY, algorithms=[ALGORITHM])
        username = payload.get("sub")
        user = session.exec(select(User).where(User.username == username)).first(
        if not user:
            raise HTTPException(status_code=401, detail="User not found")
        return user
    except JWTError:
        raise HTTPException(status_code=403, detail="Token invalid")
```

Then the user would go to, for example:

bash Copy code

```bash
/shelters/?token=abc123
```

2. Accept token in the body (POST request)

If it's a POST route, you could add a field in the request body:

python Copy code

```python
from pydantic import BaseModel

class ShelterWithToken(BaseModel):
    name: str
    location: str
    capacity: int
    token: str  # user types it in
```

```
@app.post("/shelters/")
def create_shelter(shelter: ShelterWithToken, session: Session = Depends(get_sess
    # validate token
    try:
        payload = jwt.decode(shelter.token, SECRET_KEY, algorithms=[ALGORITHM])
    except JWTError:
        raise HTTPException(status_code=403, detail="Token invalid")

    db_shelter = Shelter(name=shelter.name, location=shelter.location, capacity=s
    session.add(db_shelter)
    session.commit()
    session.refresh(db_shelter)
    return db_shelter
```

Now the user **manually enters the token in the request body**.

3. Custom header for manual token input

Instead of the standard `Authorization` header, you could have a custom header like `X-API-TOKEN` and ask users to input that. Swagger allows you to manually type headers when testing:

python Copy code

```
from fastapi import Header

def get_current_user(x_api_token: str = Header(...), session: Session = Depends(g
    try:
        payload = jwt.decode(x_api_token, SECRET_KEY, algorithms=[ALGORITHM])
        ...
```

Then users manually input `X-API-TOKEN: abc123` in Swagger or Postman.

So, the reason users **aren't currently typing the token** is because `OAuth2PasswordBearer` automatically expects the token in a header, which Swagger fills in for you after you "Authorize" — it's never meant to be manually typed per request.

If you want, I can rewrite your `create_shelter` route to **require the token manually** so you see exactly how it would work in Swagger.

Do you want me to do that?

okay ignoring that firstly,

Got it — we'll put the manual token stuff aside for now. What's the first thing you want to focus on?

is the username and password not supposed to save to the database or what?

Ah, now we're talking! In your current code, it **should** save the username and hashed password to the database when you hit `/register`, assuming a few things are correct. Here's what to check carefully:

1️⃣ Database table exists

Make sure your `User` model is defined properly and the table is created in the database.

Example:

python Copy code

```python
from sqlmodel import SQLModel, Field

class User(SQLModel, table=True):
    id: int | None = Field(default=None, primary_key=True)
    username: str = Field(index=True, unique=True)
    hashed_password: str
```

And your `lifespan` correctly creates tables:

python Copy code

```
@asynccontextmanager
async def lifespan(app: FastAPI):
    SQLModel.metadata.create_all(engine)
    yield
```

Without this, nothing will be saved.

2️⃣ Password hashing works

Your registration route encodes the password and truncates to 72 bytes (because of bcrypt limits), then hashes it. That part looks fine:

python Copy code

```
raw_password_bytes = user.password.encode("utf-8")[:72]
hashed_password = get_password_hash(raw_password_bytes)
db_user = User(username=user.username, hashed_password=hashed_password)
```

3️⃣ Database session is committing

You are doing:

python Copy code

```
session.add(db_user)
session.commit()
session.refresh(db_user)
```

✅ That should write the row to the database.

If the table exists and there are no errors, the user **should be in the database**.

4️⃣ Possible reasons it doesn't appear

- You are connecting to a **different database** than you think (check your `engine` URL).
- The table `User` wasn't created (maybe `SQLModel.metadata.create_all(engine)` didn't run before `/register`).
- There's an **error silently preventing the commit**. Try wrapping `session.commit()` in a try/except and print/log any exceptions.

If you want, I can give you a **quick snippet to test your `/register` route and confirm the user is actually saved**, using either Swagger UI or a Python script. That usually helps debug this kind of issue fast.

Do you want me to do that?

this is my sql currently in my models.py

```
from sqlmodel import SQLModel, Field
from typing import Optional

class Shelter(SQLModel, table=True):
    id: Optional[int] = Field(default=None, primary_key=True)
    name: str
    location: str
    capacity: int

class User(SQLModel, table=True):
    id: Optional[int] = Field(default=None, primary_key=True)
    username: str
    hashed_password: str
```

Perfect — your `User` model looks correct. That means the table **can** store the username and hashed password.

A few things to check to figure out why it might not be saving:

1️⃣ Table actually exists

Even if the model is correct, the table must exist in your database. With your `lifespan` function:

python Copy code

```python
@asynccontextmanager
async def lifespan(app: FastAPI):
    SQLModel.metadata.create_all(engine)
    yield
```

Make sure this is **actually running** when your app starts. You can temporarily add a print:

python Copy code

```python
@asynccontextmanager
async def lifespan(app: FastAPI):
    SQLModel.metadata.create_all(engine)
    print("Tables created!")
    yield
```

If you don't see `"Tables created!"` in your console, then the tables weren't created.

2️⃣ Session is committing

You already have:

python Copy code

```python
session.add(db_user)
session.commit()
session.refresh(db_user)
```

✅ That is correct. But just to be sure, you can add a try/except around commit to catch errors:

python Copy code

```python
try:
    session.add(db_user)
    session.commit()
    session.refresh(db_user)
except Exception as e:
    print("Error saving user:", e)
```

3️⃣ Database URL

Double-check your `engine` in `database.py`. If you're running multiple SQLite files or Postgres instances, you might be writing to a different database than you're checking.

Example for SQLite:

python Copy code

```python
from sqlmodel import create_engine, Session

engine = create_engine("sqlite:///./database.db", echo=True)
```

- `echo=True` will print SQL statements, so you can see the INSERT happening.

4️⃣ Quick test

After running `/register`, you can check your database manually:

python Copy code

```python
with Session(engine) as session:
    users = session.exec(select(User)).all()
    print(users)
```

You should see the user you just registered.

If you want, I can write a **minimal FastAPI snippet** to fully test registration, login, and verify the user actually gets saved — that usually clears up these bugs immediately.

Do you want me to do that?

so can you put that all in the code?

Absolutely! Here's a revised version of your `main.py` that:

- Ensures tables are created at startup.
- Prints out debug info for database commits.

- Adds a quick manual check at the end so you can see saved users.
- Keeps your authentication logic intact.

python Copy code

```python
from fastapi import FastAPI, Depends, HTTPException
from sqlmodel import SQLModel, Session, select
from models import Shelter, User
from database import engine, get_session
from contextlib import asynccontextmanager
from auth import verify_password, create_access_token, get_password_hash, SECRET_
from fastapi.security import OAuth2PasswordBearer, OAuth2PasswordRequestForm
from jose import jwt, JWTError
from pydantic import BaseModel, Field

# ---------------- Lifespan / Startup -----------------
@asynccontextmanager
async def lifespan(app: FastAPI):
    print("Creating tables if they don't exist...")
    SQLModel.metadata.create_all(engine)
    yield
    print("App shutdown")

app = FastAPI(lifespan=lifespan)

# ---------------- Schemas -----------------
class UserCreate(BaseModel):
    username: str
    password: str = Field(..., max_length=72)

oauth2_scheme = OAuth2PasswordBearer(tokenUrl="token")

# ---------------- Authentication -----------------
@app.post("/token")
def login(form_data: OAuth2PasswordRequestForm = Depends(), session: Session = De
    user = session.exec(select(User).where(User.username == form_data.username)).
    if not user or not verify_password(form_data.password, user.hashed_password):
        raise HTTPException(status_code=401, detail="Invalid credentials")
    token = create_access_token(data={"sub": user.username})
    return {"access_token": token, "token_type": "bearer"}

def get_current_user(token: str = Depends(oauth2_scheme), session: Session = Depe
    try:
```

```
        payload = jwt.decode(token, SECRET_KEY, algorithms=[ALGORITHM])
        username = payload.get("sub")
        user = session.exec(select(User).where(User.username == username)).first(
        if not user:
            raise HTTPException(status_code=401, detail="User not found")
        return user
    except JWTError:
        raise HTTPException(status_code=403, detail="Token invalid")

# ---------------- User Registration -----------------
@app.post("/register")
def register_user(user: UserCreate, session: Session = Depends(get_session)):
    try:
        raw_password_bytes = user.password.encode("utf-8")[:72]
        hashed_password = get_password_hash(raw_password_bytes)
        db_user = User(username=user.username, hashed_password=hashed_password)
        session.add(db_user)
        session.commit()
        session.refresh(db_user)
        print(f"User {db_user.username} saved successfully")  # Debug
        return {"detail": f"User {db_user.username} registered successfully"}
    except Exception as e:
        print("Error saving user:", e)
        raise HTTPException(status_code=500, detail="Error saving user")

# ---------------- Shelter Endpoints -----------------
@app.get("/")
def read_root():
    return {"message": "It is working!"}

@app.post("/shelters/")
def create_shelter(shelter: Shelter, session: Session = Depends(get_session), use
    session.add(shelter)
    session.commit()
    session.refresh(shelter)
    return shelter

@app.get("/shelter/{shelter_id}")
def read_shelter(shelter_id: int, session: Session = Depends(get_session), user:
    shelter = session.get(Shelter, shelter_id)
    if not shelter:
        raise HTTPException(status_code=404, detail="Shelter not found")
    return shelter
```

```python
@app.put("/shelters/{shelter_id}")
def update_shelter(shelter_id: int, updated: Shelter, session: Session = Depends(
    shelter = session.get(Shelter, shelter_id)
    if not shelter:
        raise HTTPException(status_code=404, detail="Shelter not found")
    shelter.name = updated.name
    shelter.location = updated.location
    shelter.capacity = updated.capacity
    session.commit()
    session.refresh(shelter)
    return shelter

@app.delete("/shelters/{shelter_id}")
def delete_shelter(shelter_id: int, session: Session = Depends(get_session), user
    shelter = session.get(Shelter, shelter_id)
    if not shelter:
        raise HTTPException(status_code=404, detail="Shelter not found")
    session.delete(shelter)
    session.commit()
    return {"detail": "Shelter deleted"}

# ---------------- Debug / Check Users -----------------
@app.get("/users/")
def list_users(session: Session = Depends(get_session)):
    users = session.exec(select(User)).all()
    return users
```

✅ What this does differently

1. Prints debug info when tables are created and users are saved.
2. `list_users` endpoint lets you see saved users in the database.
3. Exception handling for registration to catch database commit errors.
4. Keeps your authentication and shelter endpoints intact.

If you run this and register a user, then go to `/users/`, you should see the user you just added.

Do you want me to also **make Swagger UI automatically show the login token** so you can test protected endpoints easily?

i added through register and it's still not showing on the db

Ah, okay — if you added a user through `/register` but it's not showing up in the database, there are a few common reasons in your setup that could explain this. Let's go step by step.

1️⃣ Check which database your app is actually connecting to

Your `engine` in `database.py` is what SQLModel uses to connect. Make sure it points to a **persistent database file or server** and not an in-memory one. For example:

python Copy code

```python
# database.py
from sqlmodel import create_engine, Session

# SQLite persistent file
engine = create_engine("sqlite:///./app.db", echo=True)

def get_session():
    with Session(engine) as session:
        yield session
```

If you had `sqlite:///:memory:` before, the database exists only in RAM while the app is running. That would explain why `/users/` or your database client shows nothing.

2️⃣ Make sure the tables actually exist

When the app starts, your `lifespan` creates tables:

python Copy code

```
SQLModel.metadata.create_all(engine)
```

- With a **new database file**, this is fine.
- With **Dockerized Postgres**, ensure your tables are being created in the correct container database, not in a local file.

3️⃣ Verify the commit

In `/register`, you have:

python Copy code

```
session.add(db_user)
session.commit()
session.refresh(db_user)
```

- This should save the user.
- Add a `print(session.query(User).all())` right after commit to see what the session sees.

4️⃣ SQLite vs Postgres mismatch

- SQLite will store in a file (`.db`) locally.
- Postgres stores in the container database.
- If you switched between them or your engine URL points to Postgres but the container isn't running, nothing will save.

5️⃣ Debugging step

Add this temporarily in `/register`:

python Copy code

```
@app.post("/register")
def register_user(user: UserCreate, session: Session = Depends(get_session)):
    raw_password_bytes = user.password.encode("utf-8")[:72]
    hashed_password = get_password_hash(raw_password_bytes)
    db_user = User(username=user.username, hashed_password=hashed_password)
    session.add(db_user)
    session.commit()
```

```
    session.refresh(db_user)

    # Debug print
    users = session.exec(select(User)).all()
    print("Users in DB after commit:", users)

    return {"detail": f"User {db_user.username} registered successfully"}
```

- This will confirm whether the user is actually in the database **according to SQLModel**.
- If you see it in the print but not in your external DB viewer, it's likely a **database connection mismatch**.

💡 **Most likely problem:** your `engine` URL isn't pointing to the database you're checking. Check your `database.py` and verify whether it's SQLite file vs Postgres container.

If you want, I can **rewrite your database setup + main.py** so `/register` definitely saves users to a visible, persistent database and works with Postgres Docker. That way you can see them in Beekeeper or pgAdmin.

Do you want me to do that?

the table does exist, i checked

my other post for the shelter also works so why isnt this working

Ah, okay — that helps narrow it down. If your `Shelter` posts are working but the `User` entries are not showing up, the issue is likely **how you're sending the data** in `/register` vs how `Shelter` is created.

Let's compare:

1 Shelter works

python Copy code

```python
@app.post("/shelters/")
def create_shelter(shelter: Shelter, session: Session = Depends(get_session), use
    session.add(shelter)
    session.commit()
    session.refresh(shelter)
    return shelter
```

- `shelter: Shelter` is **the exact SQLModel table**, so FastAPI can automatically parse the JSON body into a `Shelter` object.
- Example POST body:

json Copy code

```json
{
  "name": "Safe Haven",
  "location": "Kingston",
  "capacity": 50
}
```

2 User register currently

python Copy code

```python
@app.post("/register")
def register_user(user: UserCreate, session: Session = Depends(get_session)):
    raw_password_bytes = user.password.encode("utf-8")[:72]
    hashed_password = get_password_hash(raw_password_bytes)
    db_user = User(username=user.username, hashed_password=hashed_password)
    session.add(db_user)
    session.commit()
    session.refresh(db_user)
    return {"detail": f"User {db_user.username} registered successfully"}
```

- `user: UserCreate` is a **Pydantic model**, not the SQLModel table.
- That's fine, but the JSON you send must match `UserCreate`:

json Copy code

```
{
  "username": "testuser",
  "password": "mypassword"
}
```

- If you accidentally sent the field as `hashed_password` instead of `password`, `db_user` may get empty or invalid values.

3️⃣ Common pitfalls

1. **JSON field names don't match `UserCreate`**
   - Make sure it's `username` and `password`, not `user` or `hashed_password`.
2. **Session is rolled back due to exception**
   - If `db_user` violates a constraint (like unique username), SQLModel may rollback silently. Check the console logs for exceptions.
3. **Database connection mismatch**
   - `Shelter` is saving because it might be using a different table in a working schema. Double-check the `User` table is in the same DB as the `Shelter` table.

4️⃣ Quick debug

Add this at the end of `/register`:

python Copy code

```python
all_users = session.exec(select(User)).all()
print("All users in DB:", all_users)
```

- If this prints the user but you don't see it in Beekeeper, it's a **schema/connection issue**.
- If it prints empty, the issue is **JSON input or commit failure**.

💡 **Most likely reason:** the JSON you're sending to `/register` doesn't match `UserCreate`. Swagger may show `"password"` as required, so make sure you're not sending `"hashed_password"` or `"user"` instead.

I can write an example **tested curl / Swagger POST** for `/register` that will definitely save the user if you want. Do you want me to do that?

like this

```
@app.post("/register")
def register_user(user: UserCreate, session: Session = Depends(get_session)):
    try:
        raw_password_bytes = user.password.encode("utf-8")[:72]
        hashed_password = get_password_hash(raw_password_bytes)
        db_user = User(username=user.username, hashed_password=hashed_password)
        session.add(db_user)
        session.commit()
        session.refresh(db_user)
        print(f"User {db_user.username} saved successfully")  # Debug
        return {"detail": f"User {db_user.username} registered successfully"}
        all_users = session.exec(select(User)).all()
        print("All users in DB:", all_users)
    except Exception as e:
        print("Error saving user:", e)
        raise HTTPException(status_code=500, detail="Error saving user")
```

Almost — the issue here is **anything after a `return` statement will never run**.

In your code:

python Copy code

```python
return {"detail": f"User {db_user.username} registered successfully"}
all_users = session.exec(select(User)).all()  # ← this will never execute
print("All users in DB:", all_users)
```

Those two lines are unreachable. You should move them **before the** `return` . Here's a fixed version:

python Copy code

```python
@app.post("/register")
def register_user(user: UserCreate, session: Session = Depends(get_session)):
    try:
        raw_password_bytes = user.password.encode("utf-8")[:72]
        hashed_password = get_password_hash(raw_password_bytes)
        db_user = User(username=user.username, hashed_password=hashed_password)
        session.add(db_user)
        session.commit()
        session.refresh(db_user)
        print(f"User {db_user.username} saved successfully")  # Debug

        # Debug: list all users in DB
        all_users = session.exec(select(User)).all()
        print("All users in DB:", all_users)

        return {"detail": f"User {db_user.username} registered successfully"}
    except Exception as e:
        print("Error saving user:", e)
        raise HTTPException(status_code=500, detail="Error saving user")
```

✅ Now the debug prints will run, and you can see in the console whether the user actually got saved.

After this, if the user still doesn't appear in Beekeeper, it's likely a **database connection / schema mismatch**, not a code bug.

If you want, I can give a **fully working minimal version** of `/register` + `/token` + `/shelters` that will save users and protect endpoints correctly. Do you want me to do that?

okay nvm

Got it! We can pause on that.

how to remove rows in sql

In SQL, removing rows from a table is done with the `DELETE` statement. Here's a breakdown:

1. Delete specific rows

Use a `WHERE` clause to target specific rows:

sql Copy code

```sql
DELETE FROM table_name
WHERE condition;
```

Example:

sql Copy code

```sql
DELETE FROM Shelter
WHERE id = 3;
```

This deletes the row in `Shelter` where the `id` is 3.

2. Delete all rows

If you want to clear the table completely:

sql Copy code

```sql
DELETE FROM table_name;
```

Example:

sql Copy code

```sql
DELETE FROM Shelter;
```

Warning: This removes **all rows**, but keeps the table structure.

3. Delete and reset auto-increment (optional)

If your table has an auto-increment ID and you want to reset it (PostgreSQL example):

sql Copy code

```sql
TRUNCATE TABLE Shelter RESTART IDENTITY;
```

- `TRUNCATE TABLE` is faster than `DELETE FROM` for large tables.
- It also removes all rows but resets auto-increment counters.

If you want, I can also show the **SQLModel / Python way** to delete rows using your current FastAPI + SQLModel setup so you don't have to write raw SQL.

Do you want me to show that?

is there a way to make it so i when im trying to update something with put it only updates part of it?

Yes 👍 — this is exactly what **partial updates** are for.

There are **two clean ways** to do this in FastAPI + SQLModel.

✅ Best practice: use `PATCH` + optional fields

This lets you update **only the fields the user sends**.

1️⃣ Create an *update schema* with optional fields

python

```python
from typing import Optional
from pydantic import BaseModel

class ShelterUpdate(BaseModel):
    name: Optional[str] = None
    location: Optional[str] = None
    capacity: Optional[int] = None
```

Copy code

2 Use PATCH instead of PUT

python

```python
@app.patch("/shelters/{shelter_id}")
def update_shelter_partial(
    shelter_id: int,
    updates: ShelterUpdate,
    session: Session = Depends(get_session),
    user: User = Depends(get_current_user),
):
    shelter = session.get(Shelter, shelter_id)
    if not shelter:
        raise HTTPException(status_code=404, detail="Shelter not found")

    update_data = updates.model_dump(exclude_unset=True)

    for key, value in update_data.items():
        setattr(shelter, key, value)

    session.commit()
    session.refresh(shelter)
    return shelter
```

Copy code

3 Example request (Swagger / JSON)

You can send **only what you want to change**:

json

```json
{
  "capacity": 120
```

```
}
```

Copy code

or

json

```
{
  "name": "New Shelter Name",
  "location": "Kingston"
}
```

Copy code

Everything else stays the same.

⚠️ If you must keep `PUT`

Technically you *can* do partial updates with `PUT`, but it's **not REST-correct**. `PUT` implies replacing the entire resource.

If you still want it:

python

```python
@app.put("/shelters/{shelter_id}")
def update_shelter(
    shelter_id: int,
    updates: ShelterUpdate,
    session: Session = Depends(get_session),
    user: User = Depends(get_current_user),
):
    shelter = session.get(Shelter, shelter_id)
    if not shelter:
        raise HTTPException(status_code=404, detail="Shelter not found")

    for field, value in updates.model_dump(exclude_unset=True).items():
        setattr(shelter, field, value)

    session.commit()
    session.refresh(shelter)
    return shelter
```

Copy code

🧠 Rule of thumb

Method	Use when
`POST`	Create new
`PUT`	Replace entire object
`PATCH`	Update **part** of an object ✅

If you want, I can:

- Convert your **current PUT endpoint** to PATCH
- Add **field-level validation** (e.g., capacity must be > 0)
- Make Swagger show cleaner optional inputs

Just tell me 👍

db_product.is_archived = True

instead of doing something like this is there a way to flip it each time? so that if i need to unarchive the same endpoint will do so?

Yes — this is a **toggle**, and there's a clean, safe way to do it ✅

If `is_archived` is a boolean, you can flip it every time the endpoint is called.

✅ The simple toggle (recommended)

python

```python
db_product.is_archived = not db_product.is_archived
```

Copy code

That's it.

- `True → False`

- `False → True`

🔁 Example endpoint (archive / unarchive with one call)

python

```python
@app.patch("/products/{product_id}/archive")
def toggle_archive(
    product_id: int,
    session: Session = Depends(get_session),
):
    db_product = session.get(Product, product_id)
    if not db_product:
        raise HTTPException(status_code=404, detail="Product not found")

    db_product.is_archived = not db_product.is_archived

    session.commit()
    session.refresh(db_product)

    return {
        "id": db_product.id,
        "is_archived": db_product.is_archived,
        "message": "Product archived" if db_product.is_archived else "Product una
    }
```

Copy code

🧠 Why this is better than two endpoints

❌ `/archive` and `/unarchive`
❌ Passing `true/false` manually
✅ One endpoint, predictable behavior
✅ Less frontend logic
✅ Harder to misuse

⚠️ Optional safety check (if NULL is possible)

If `is_archived` might be `None` in older rows:

Copy code

python

```python
db_product.is_archived = not bool(db_product.is_archived)
```

Swagger behavior

Each time you hit the endpoint:

- First call → archives
- Second call → unarchives
- Third → archives again

Perfect for toggle buttons in a UI.

If you want, I can also show:

- How to **log who archived it**
- How to **soft-delete instead**
- How to **prevent archiving active items**

Just say the word 👍