CRUD REST API DOCUMENTATION

Overview :

This project is a RESTful API for managing a product catalog, allowing users to create, read, update, and delete product information. It is built using FastAPI, SQLAlchemy, and SQLite, providing a modern and efficient approach to API development.

Tools and Technologies Used :

- Programming Language: Python
- Framework: FastAPI (for building the API)
- Database: SQLite (for data storage)
- ORM: SQLAlchemy (for database interactions)
- Migrations: Alembic (for handling database schema changes)
- Data Validation: Pydantic (for defining data models and validation)
- Development Environment: Visual Studio Code (VSCode)
- Testing: Pytest (for unit testing)

Setup Instructions :

First, clone this repository in your preferred code editor using the following command, then navigate to the respective folder named Final_API

git clone https://github.com/19Rithvik/Final_API.git

cd Final_API

Now install required packages mentioned in the requirements file:

- Fastapi(A modern, fast web framework for building APIs with Python.)
- Uvicorn(ASGI server used to serve the FastAPI application.)
- sqlalchemy(ORM (Object Relational Mapper) for handling interactions with the SQLite database.)
- alembic(A tool for handling database migrations, managing changes to your database schema over time.)
- pydantic(For data validation and defining API request/response models bash) pytest(Testing framework for writing unit tests for your API.)

Install the required packages from the requirements file using the following command:

pip install -r requirements.txt

Note that the __init__.py file must be present in any folder being created, as it allows the folder to be recognized as a package in other files

Code Explanation :

1) MAIN.PY :

```
from fastapi import FastAPI, HTTPException, Query, Body, Depends

from pydantic import BaseModel, Fieldfrom sqlalchemy import Column, Integer,
String, Float, create_engine, UniqueConstraint

from sqlalchemy.orm import sessionmaker, Session, declarative_base
from sqlalchemy.exc import IntegrityError
import logging
from typing import Optional, List
```

Import the packages mentioned above and this code sets up the foundation for building a web API using FastAPI and SQLAlchemy. It starts by importing necessary modules from FastAPI, such as `FastAPI` for creating the API, `HTTPException` for handling errors, and utilities like `Query`, `Body`, and `Depends` for managing user input. It uses Pydantic's `BaseModel` and `Field` to define and validate the structure of data sent or received by the API. SQLAlchemy is used for interacting with the database, importing `Column`, data types like `Integer`, `String`, and `Float`, as well as `create_engine` to connect to the database and `UniqueConstraint` to ensure unique data entries. It also sets up tools like `sessionmaker` and `Session` to manage database interactions, and `declarative_base` for defining database models. Additionally, it imports `IntegrityError` to handle database rule violations, and the `logging` module is configured to log information about the API's operations, helping track system behavior and errors. The final line configures logging to capture informational messages, ensuring smooth debugging and monitoring of the API.

```
SQLALCHEMY_DATABASE_URL = "sqlite:///./test.db"

engine = create_engine(SQLALCHEMY_DATABASE_URL,
connect_args={"check_same_thread": False})
SessionLocal = sessionmaker(autocommit=False, autoflush=False, bind=engine)
Base = declarative_base()
```

This code sets up the connection to a SQLite database for the API. The `SQLALCHEMY_DATABASE_URL` defines the path to the database file, which in this case is called "test.db" and is stored locally. The `create_engine` function is then used to create the database connection, and the `connect_args={"check_same_thread": False}` argument ensures SQLite allows

multiple threads to interact with the database. The `SessionLocal` variable creates a session factory using `sessionmaker`, which controls how the API communicates

with the database, ensuring that changes are not automatically committed (`autocommit=False`) or automatically flushed to the database (`autoflush=False`). Finally, `Base = declarative_base()` sets up a base class for defining the structure of the database tables. This base class is used to map Python objects to database tables.

```
Base.metadata.create_all(bind=engine)


# FastAPI app
app = FastAPI()
```

The line `Base.metadata.create_all(bind=engine)` creates any missing tables in the database using the defined models. The `app = FastAPI()` initializes the main FastAPI application to handle incoming API requests and responses.

```
class Product(Base):
    __tablename__ = "products"
    id = Column(Integer, primary_key=True, index=True)
    name = Column(String, nullable=False)
    price = Column(Float, nullable=False)
    quantity = Column(Integer, nullable=False)
    description = Column(String, nullable=True)
    category = Column(String, nullable=True)

    __table_args__ = (UniqueConstraint('name', name='uq_product_name'),)  #
Unique constraint
```

The `Product` class defines a database model for a table named "products" using SQLAlchemy. It has several columns: `id`, which is an integer serving as the primary key with indexing for faster searches; `name`, a required text field that cannot be null; `price`, a required floating-point number representing the product's cost; and `quantity`, a required integer representing the available stock. The `description` and `category` columns are optional text fields, allowing for

additional product details and categorization. Additionally, the model includes a unique constraint on the `name` field, ensuring that no two products in the table have the same name, with the constraint being named `uq_product_name` for reference in the database.

```
class ProductCreate(BaseModel):
    name: str
    price: float = Field(..., gt=0, description="Price must be greater than 0")
    quantity: int = Field(..., ge=0, description="Quantity must be greater than
or equal to 0")
    description: Optional[str] = None
    category: Optional[str] = None

# Pydantic Model for Product Schema (Response with ID)
class ProductResponse(ProductCreate):
    id: int

# Pydantic Model for Product Schema (Update)
class ProductUpdate(BaseModel):
    name: Optional[str] = None
    price: Optional[float] = Field(None, gt=0)
    quantity: Optional[int] = Field(None, ge=0)
    description: Optional[str] = None
    category: Optional[str] = None
```

the ProductCreate class is designed for validating data when creating new products, while the ProductResponse class extends this to include an id field for API responses. The ProductUpdate class is tailored for updating existing products, allowing optional fields to enable partial updates without requiring all attributes to be provided.

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

the get_db function creates a new database session, yields it for use in other parts of the application (such as in dependency injection for API endpoints), and ensures that the session is properly closed afterward to prevent resource leaks. This function is typically used in FastAPI to manage database connections in a clean and efficient way

```
@app.post("/products/", response_model=ProductResponse)
def create_product(product: ProductCreate = Body(...), db: Session =
Depends(get_db)):
    try:
        db_product = Product(**product.model_dump())
        db.add(db_product)
        db.commit()
        db.refresh(db_product)
        return db_product
    except IntegrityError:
        db.rollback()
        raise HTTPException(status_code=400, detail="Product creation failed due
to duplicate name.")
```

this function defines a FastAPI endpoint for creating new products. It processes incoming product data, interacts with the database to store this data, and handles potential errors related to database constraints, ensuring that the API responds appropriately to both successful and failed creation attempts

```
@app.get("/products/{product_id}", response_model=ProductResponse)
def get_product(product_id: int, db: Session = Depends(get_db)):
    product = db.query(Product).filter(Product.id == product_id).first()
    if not product:
        raise HTTPException(status_code=404, detail="Product not found")
    return product
```

this function defines an API endpoint for retrieving a product by its ID. It queries the database for a product with the specified ID, and if found, returns it. If the

product is not found, it raises a 404 error to inform the client that the product doesn't exist.

```
@app.put("/products/{product_id}", response_model=ProductResponse)
def update_product(product_id: int, product: ProductUpdate, db: Session =
Depends(get_db)):
    db_product = db.query(Product).filter(Product.id == product_id).first()
    if not db_product:
        raise HTTPException(status_code=404, detail="Product not found")

    update_data = product.dict(exclude_unset=True)

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

    db.commit()
    db.refresh(db_product)
    return db_product
```

this function defines an API endpoint for updating an existing product by its ID. It retrieves the product from the database, checks if it exists, applies the updates provided in the request, commits the changes, and returns the updated product. If the product is not found, it raises a 404 error.

```
@app.delete("/products/{product_id}", response_model=dict)
def delete_product(product_id: int, db: Session = Depends(get_db)):
    db_product = db.query(Product).filter(Product.id == product_id).first()
    if not db_product:
        raise HTTPException(status_code=404, detail="Product not found")

    db.delete(db_product)
    db.commit()
    return {"message": "Product deleted successfully"}
```

this function defines an API endpoint for deleting a product by its ID. It retrieves the product from the database, checks if it exists, deletes it, commits the changes, and returns a success message. If the product is not found, it raises a 404 error.

```python
@app.get("/products/", response_model=List[ProductResponse])
def list_products(price_gte: float = Query(None, ge=0), db: Session =
Depends(get_db)):
    query = db.query(Product)
    if price_gte is not None:
        query = query.filter(Product.price >= price_gte)
    products = query.all()
    return products if products else []
```

this function defines an API endpoint for retrieving a list of products. It accepts an optional query parameter to filter products by a minimum price. It retrieves matching products from the database and returns them as a list. If no products are found, it returns an empty list.

2) Alembic Migration:

Now, after completing the main file, run the following commands in the terminal to create the Alembic migration:

alembic init alembic

This will create an Alembic folder with a version folder, an env.py file, and a separate alembic.ini file. Before proceeding with coding, we need to change the database URL to SQLite in the .ini file:

sqlalchemy.url = sqlite:///./test.db

Now, open the env.py file inside the Alembic folder and update the target_metadata from None to Base.metadata. Also, import Base from main.py. This is necessary for Alembic to use the base model from main.py and create migrations for the database

Now, run the following command, which will create the initial migration where we won't see the category in the database:

alembic revision --autogenerate -m "Initial migration"

This will create the initial migration with a random number as the file name (e.g., www_initial_migration.py) inside the version folder of the Alembic folder, with the schema automatically defined by the Alembic migration based on the schema provided in main.py. Note that the category column should not be declared in main.py while creating the initial migration. After creating the initial migration, check whether the SQLite database has been created according to the given Pydantic model using Alembic by running the following command:

```
      python -c "import sqlite3; conn = sqlite3.connect('your_databse_path');
cursor = conn.cursor(); cursor.execute('PRAGMA table_info(table_name);');
print(cursor.fetchall()); conn.close()"
```

This output shows the column index, name, type, not null constraint, default value, and whether it's a primary key,

```
[(0, 'id', 'INTEGER', 1, None, 1), (1, 'name', 'VARCHAR', 1, None, 0), (2,
'price', 'FLOAT', 1, None, 0), (3, 'quantity', 'INTEGER', 1, None, 0), (4,
'description', 'VARCHAR', 0, None, 0)]
```

Now, create another migration that adds the category column by running:

alembic revision --autogenerate -m "Add category column to products"

This will create a new migration for the category column with a random number as the file name (e.g., www_Add_category_column_to_products.py) inside the version folder of the Alembic folder, and it will update the schema in both main.py and the database.

Make sure that there is no manual writing of code for those migration files, as they are automatically generated. Ensure it looks like this (it may vary based on the names given to the attributes by the user).

After creating the second migration, check whether the SQLite database has been updated according to the given Pydantic model using Alembic by running the same command as before. The difference in the output should be that the category column has been added to the SQLite database

```
[(0, 'id', 'INTEGER', 1, None, 1), (1, 'name', 'VARCHAR', 1, None, 0), (2,
'price', 'FLOAT', 1, None, 0), (3, 'quantity', 'INTEGER', 1, None, 0), (4,
'description', 'VARCHAR', 0, None, 0), (5, 'category', 'VARCHAR', 0, None, 0)]
```

The .py.mako file is created inside the Alembic folder to provide a flexible way to ensure that migration scripts are dynamically generated and consistent across different environments or schema changes. Typically, you won't need to modify these files manually

Initial migration:

```
"""Initial migration

Revision ID: c0f828767d02
Revises:
Create Date: 2024-09-23 23:00:19.352089

"""
from typing import Sequence, Union

from alembic import op
import sqlalchemy as sa


# revision identifiers, used by Alembic.
revision: str = 'c0f828767d02'
down_revision: Union[str, None] = None
branch_labels: Union[str, Sequence[str], None] = None
depends_on: Union[str, Sequence[str], None] = None



def upgrade() -> None:
    # ### commands auto generated by Alembic - please adjust! ###
    op.create_table('products',
```

```
    sa.Column('id', sa.Integer(), nullable=False),
    sa.Column('name', sa.String(), nullable=False),
    sa.Column('price', sa.Float(), nullable=False),
    sa.Column('quantity', sa.Integer(), nullable=False),
    sa.Column('description', sa.String(), nullable=True),
    sa.PrimaryKeyConstraint('id'),
    sa.UniqueConstraint('name', name='uq_product_name')
    )
    op.create_index(op.f('ix_products_id'), 'products', ['id'], unique=False)
    # ### end Alembic commands ###


def downgrade() -> None:
    # ### commands auto generated by Alembic - please adjust! ###
    op.drop_index(op.f('ix_products_id'), table_name='products')
    op.drop_table('products')
    # ### end Alembic commands ###
```

Add column migration:

```
"""Add category column to products

Revision ID: e5699365c8ef
Revises: c0f828767d02
Create Date: 2024-09-23 23:00:52.333193

"""
from typing import Sequence, Union

from alembic import op
import sqlalchemy as sa


# revision identifiers, used by Alembic.
revision: str = 'e5699365c8ef'
down_revision: Union[str, None] = 'c0f828767d02'
branch_labels: Union[str, Sequence[str], None] = None
depends_on: Union[str, Sequence[str], None] = None



def upgrade() -> None:
    # ### commands auto generated by Alembic - please adjust! ###
    op.add_column('products', sa.Column('category', sa.String(), nullable=True))
    # ### end Alembic commands ###
```

```python
def downgrade() -> None:
    # ### commands auto generated by Alembic - please adjust! ###
    op.drop_column('products', 'category')
    # ### end Alembic commands ###
```

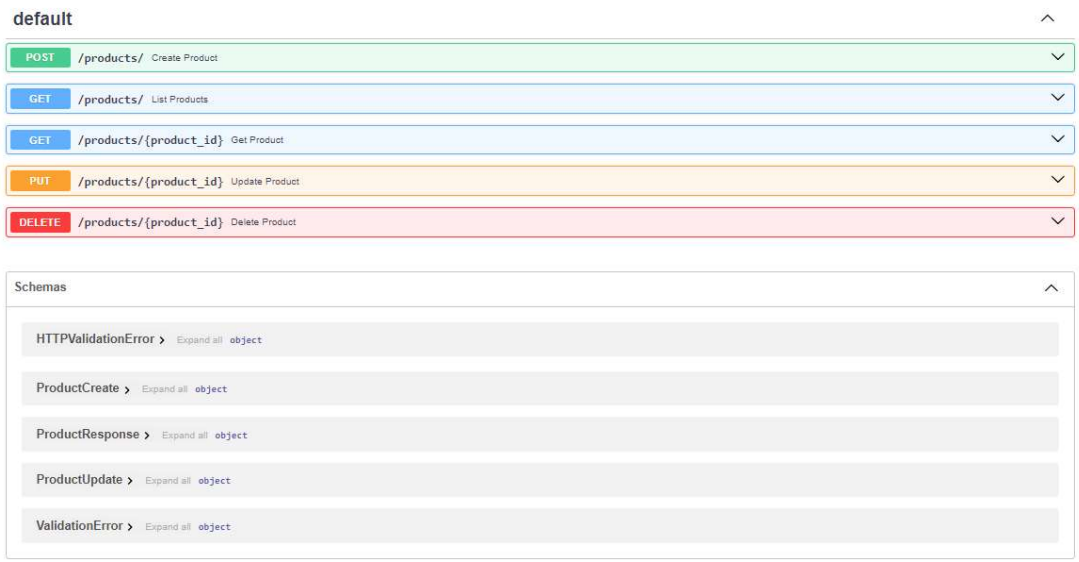
TESTING :

1) Test the file created using FastApi-swagger:

After creating the API, run the following command to test it:

uvicorn main:app –reload

This will load the file and create a local environment on port 8000. Now, navigate to the local host's docs to test the API at (http://127.0.0.1:8000/docs). After that, you will see a page like this:



Now, feel free to test the model created from the options above

2) Test.py File

Another way to test the API is by creating a test file and then running the pytest command to check whether it's working or not. Here is a sample file for the test

```python
from fastapi.testclient import TestClient
from .main import app

client = TestClient(app)

# Function to clean up the database before tests
def cleanup_products():
    response = client.get("/products/")
    for product in response.json():
        client.delete(f"/products/{product['id']}")

# Positive Test Case 1: Create a product successfully
def test_create_product_success():
    cleanup_products()  # Clean up before running the test

    response = client.post("/products/", json={
        "name": "Valid Product",
        "price": 29.99,
        "quantity": 150,
        "description": "A valid product for testing.",
        "category": "Valid Category"
    })

    if response.status_code == 200:
        product = response.json()
        print(f"Success: Product created successfully - ID: {product['id']},
Name: {product['name']}, Price: {product['price']}")
    else:
        print(f"Failure: Product creation failed with status
{response.status_code} - Detail: {response.json()}")

# Negative Test Case 1: Create a product with a duplicate name
def test_create_product_duplicate_name():
    cleanup_products()  # Clean up before running the test

    client.post("/products/", json={
        "name": "Duplicate Product",
        "price": 20.99,
        "quantity": 200
    })

    response = client.post("/products/", json={
        "name": "Duplicate Product",
        "price": 25.99,
        "quantity": 150
    })

    if response.status_code == 400:
```

```
        assert response.json()["detail"] == "Product creation failed due to
duplicate name."
        print(f"Success: Duplicate product creation handled correctly - Status:
{response.status_code}, Detail: {response.json()}")
    else:
        print(f"Failure: Duplicate product creation failed with status
{response.status_code} - Detail: {response.json()}")

# Positive Test Case 2: Get a product successfully
def test_get_product_success():
    cleanup_products()  # Clean up before running the test

    create_response = client.post("/products/", json={
        "name": "Product for Get",
        "price": 15.99,
        "quantity": 50
    })

    if create_response.status_code == 200:
        product_id = create_response.json()["id"]
        response = client.get(f"/products/{product_id}")

        if response.status_code == 200:
            product = response.json()
            print(f"Success: Product retrieved successfully - ID:
{product['id']}, Name: {product['name']}")
        else:
            print(f"Failure: Retrieval of product {product_id} failed with status
{response.status_code} - Detail: {response.json()}")
    else:
        print(f"Failure: Product creation failed with status
{create_response.status_code} - Detail: {create_response.json()}")

# Negative Test Case 2: Get a non-existent product
def test_get_non_existent_product():
    cleanup_products()  # Clean up before running the test

    response = client.get("/products/99999")  # Assuming this ID does not exist
    if response.status_code == 404:
        print(f"Success: Non-existent product retrieval handled correctly -
Status: {response.status_code}, Detail: {response.json()}")
    else:
        print(f"Failure: Attempt to get non-existent product failed with status
{response.status_code} - Detail: {response.json()}")

# Positive Test Case 3: Update a product successfully
def test_update_product_success():
    cleanup_products()  # Clean up before running the test
```

```
    create_response = client.post("/products/", json={
        "name": "Product to Update",
        "price": 15.99,
        "quantity": 60
    })

    if create_response.status_code == 200:
        product_id = create_response.json()["id"]
        response = client.put(f"/products/{product_id}", json={
            "price": 20.99,
            "quantity": 80
        })

        if response.status_code == 200:
            updated_product = response.json()
            print(f"Success: Product updated successfully - ID:
{updated_product['id']}, New Price: {updated_product['price']}, New Quantity:
{updated_product['quantity']}")
        else:
            print(f"Failure: Product update failed with status
{response.status_code} - Detail: {response.json()}")
    else:
        print(f"Failure: Product creation failed with status
{create_response.status_code} - Detail: {create_response.json()}")

# Negative Test Case 3: Update a non-existent product
def test_update_non_existent_product():
    cleanup_products()  # Clean up before running the test

    response = client.put("/products/99999", json={"price": 30.00})  # Assuming
this ID does not exist
    if response.status_code == 404:
        print(f"Success: Attempt to update non-existent product handled correctly
- Status: {response.status_code}, Detail: {response.json()}")
    else:
        print(f"Failure: Attempt to update non-existent product failed with
status {response.status_code} - Detail: {response.json()}")

# Negative Test Case 4: Delete a non-existent product
def test_delete_non_existent_product():
    cleanup_products()  # Clean up before running the test

    response = client.delete("/products/99999")  # Assuming this ID does not
exist
    if response.status_code == 404:
        print(f"Success: Attempt to delete non-existent product handled correctly
- Status: {response.status_code}, Detail: {response.json()}")
```

```
    else:
        print(f"Failure: Attempt to delete non-existent product failed with
status {response.status_code} - Detail: {response.json()}")

# Positive Test Case 5: List products with a price filter
def test_list_products_with_price_filter():
    cleanup_products()  # Clean up before running the test

    client.post("/products/", json={
        "name": "Product A",
        "price": 30.99,
        "quantity": 10
    })
    client.post("/products/", json={
        "name": "Product B",
        "price": 40.99,
        "quantity": 20
    })

    response = client.get("/products/?price_gte=35")
    if response.status_code == 200:
        products = response.json()
        if len(products) == 1 and products[0]["name"] == "Product B":
            print(f"Success: Product list retrieved with price filter
successfully - Filtered product name: {products[0]['name']}")
        else:
            print(f"Failure: Price filter did not return expected results -
Status: {response.status_code}, Detail: {products}")
    else:
        print(f"Failure: Attempt to list products with price filter failed with
status {response.status_code} - Detail: {response.json()}")

# Run these test cases
if __name__ == "__main__":
    import pytest
    pytest.main()
```

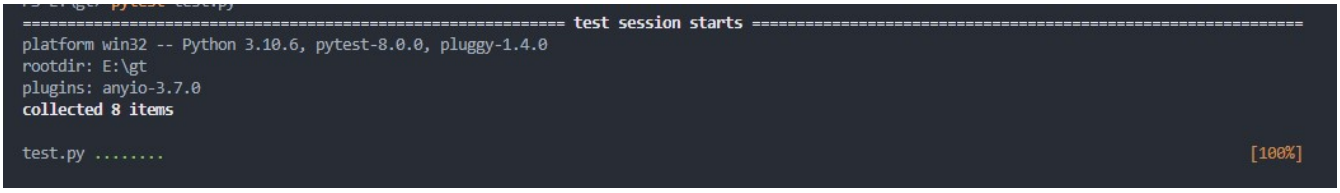
This is testing for the FastAPI application is designed to validate various functionalities related to product management. It begins by setting up a test client and includes a cleanup function that removes any existing products from the database before each test. The suite consists of several test cases: it successfully creates a product, handles duplicate product names appropriately,

retrieves products by ID, and manages attempts to access non-existent products. It also tests the updating and deletion of products, ensuring that operations on non-existent products return the correct error status. Finally, it validates the listing of products with a price filter, confirming that only products meeting the specified criteria are returned. Each test case prints success or failure messages based on the outcomes, providing clear feedback on the API's behavior during testing.

Now run the following command for testing it

pytest test.py

You will see the following output:



If you encounter any other result than dots after test.py ad mentioned in screenshot (like test.py ..F.F…)then there may be some failed cases.

Conclusion:

This project demonstrates the development of a RESTful API using FastAPI and SQLite, focusing on structured data management through CRUD operations. The implementation ensures data integrity and usability, catering to both developers and non-developers. Using Alembic migrations, database schema updates are handled efficiently. Testing is performed using Pytest and Swagger UI, ensuring robust and reliable API functionality