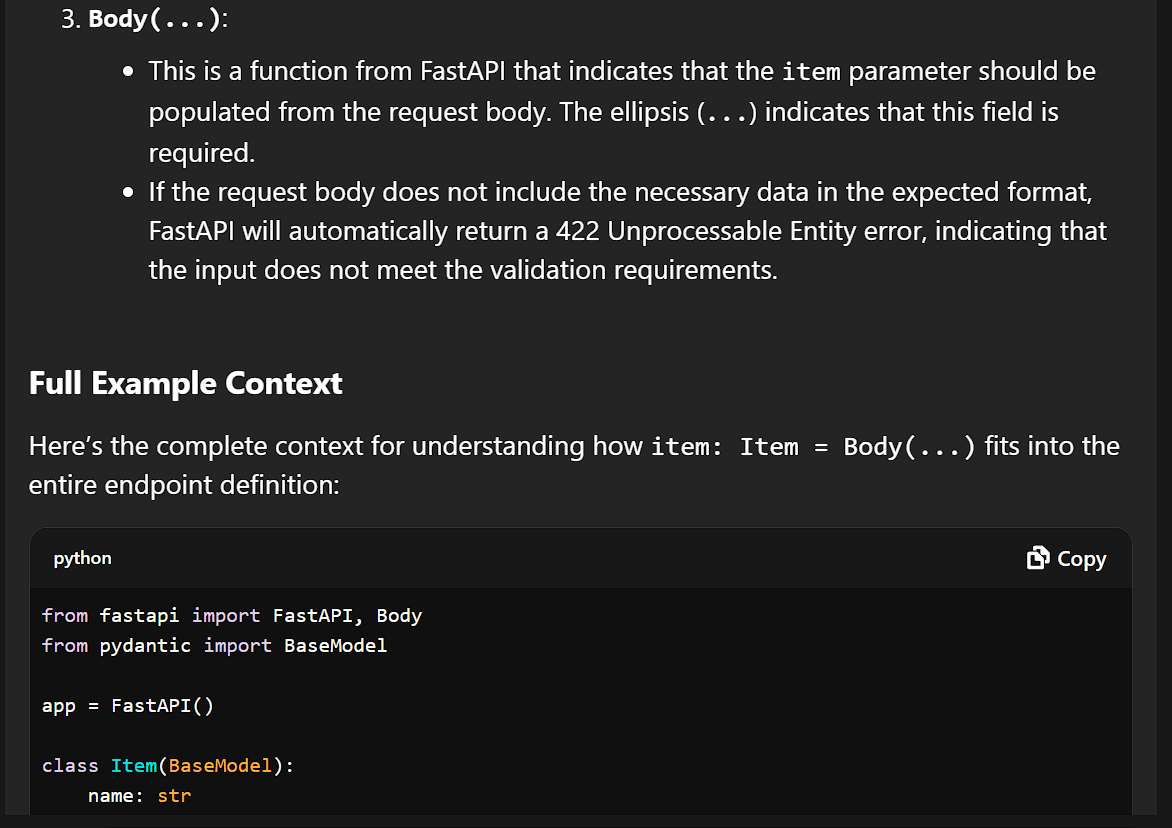
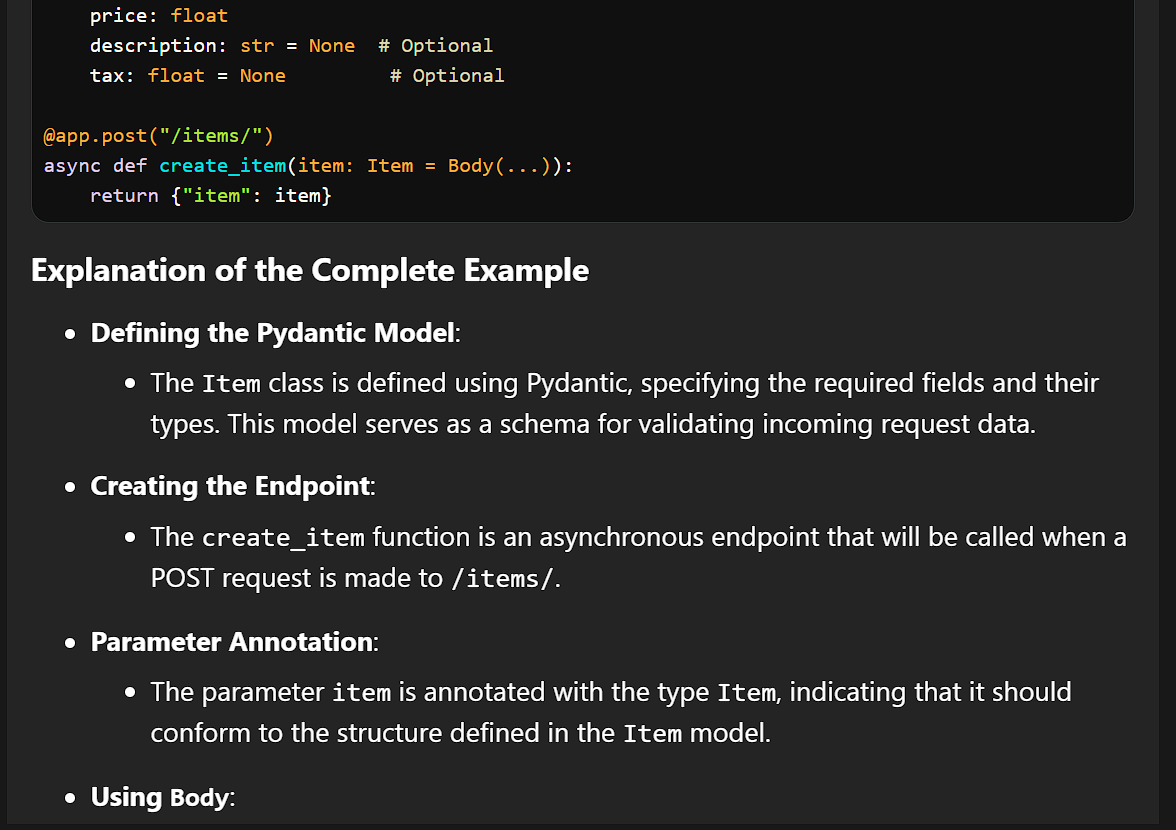
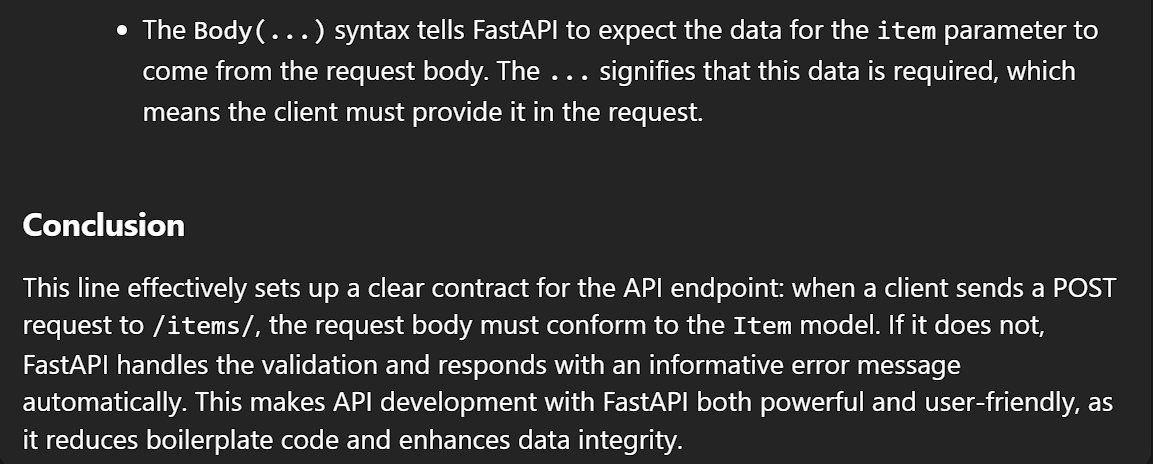
* Learn about decorators

All about the Body() function (type hinting)

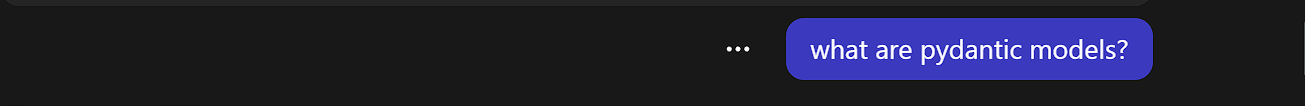


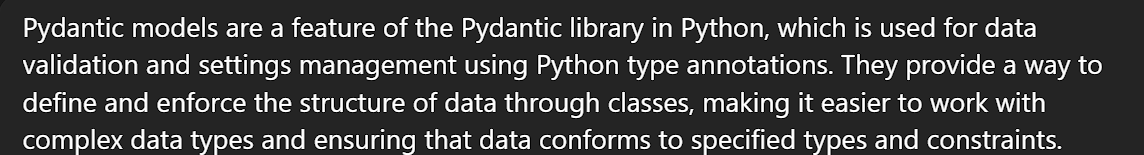


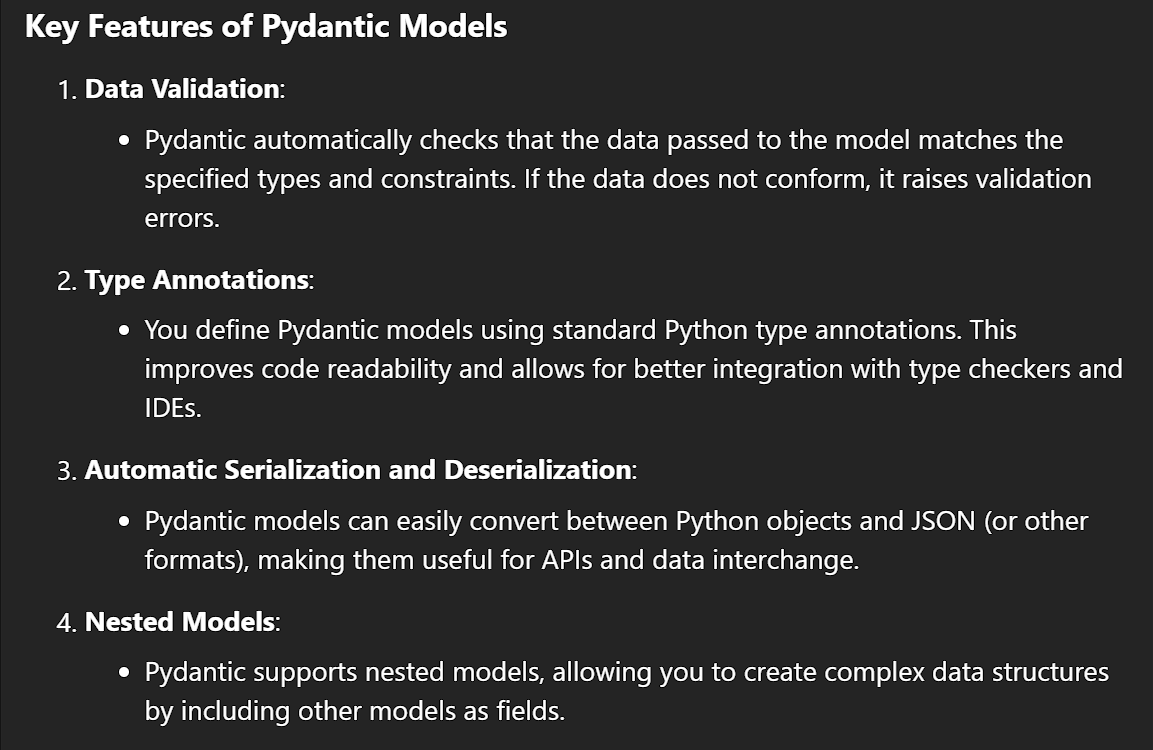


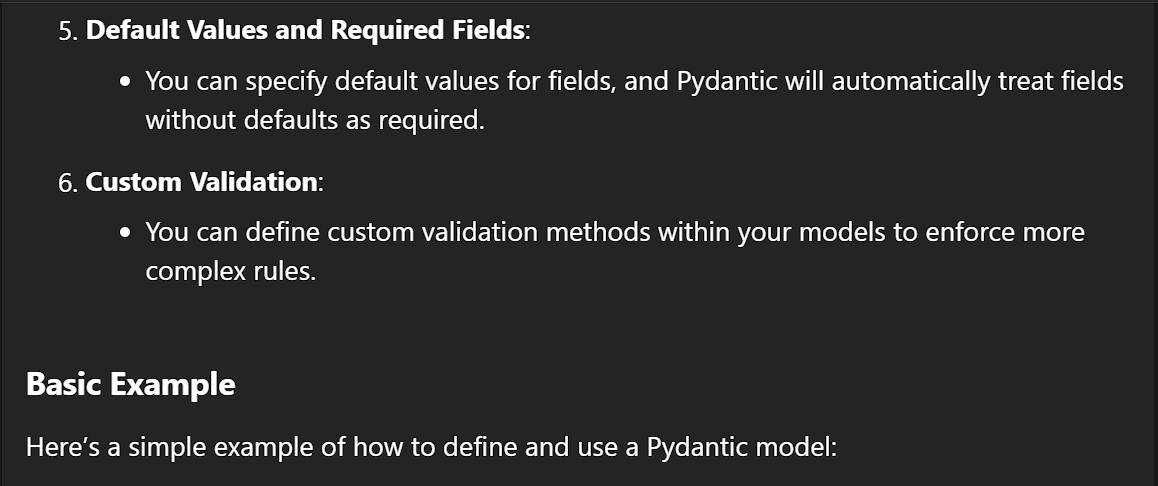


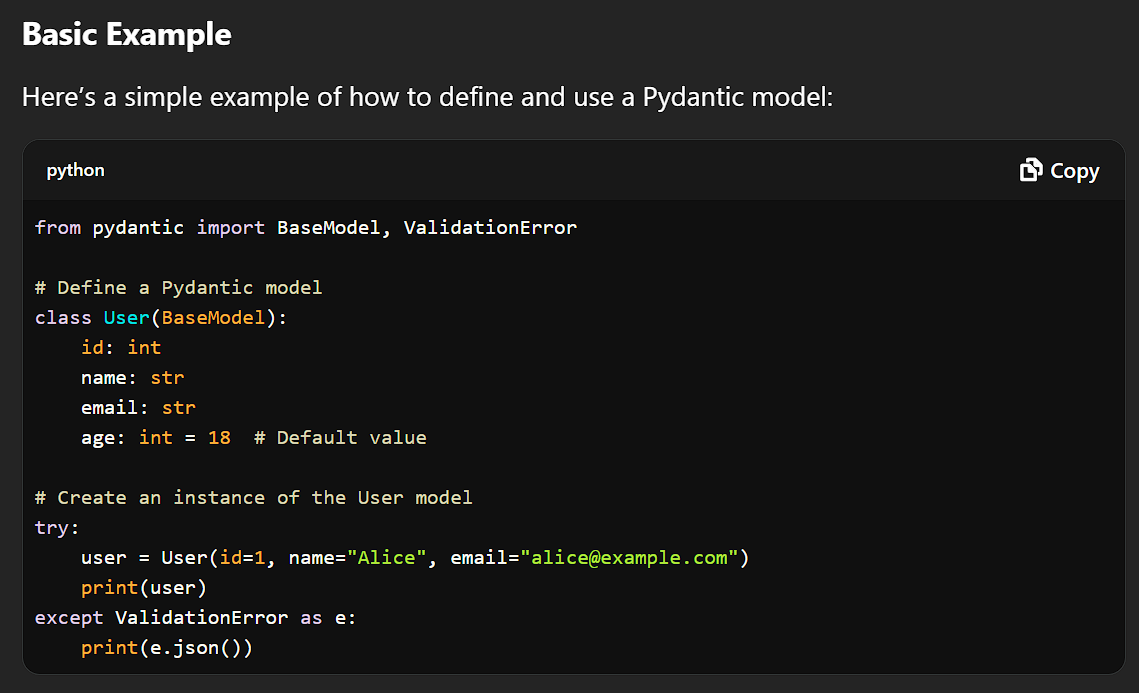
PYDANTIC MODELS

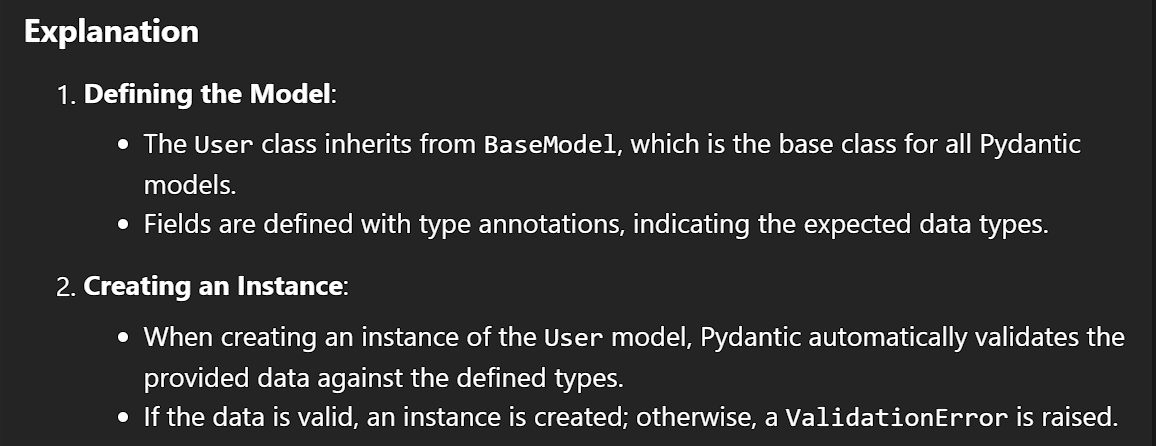


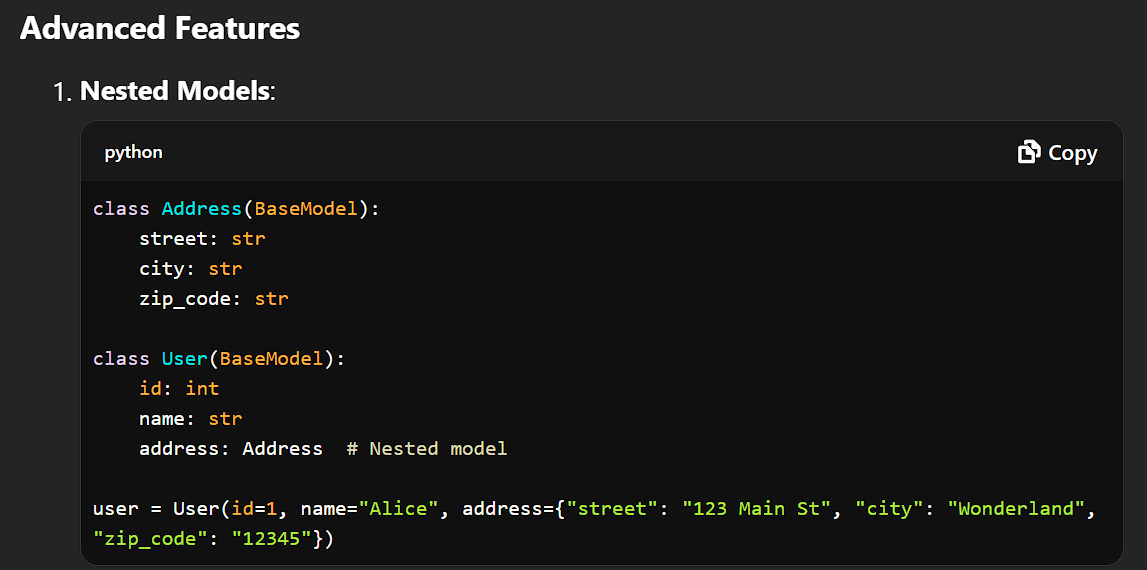




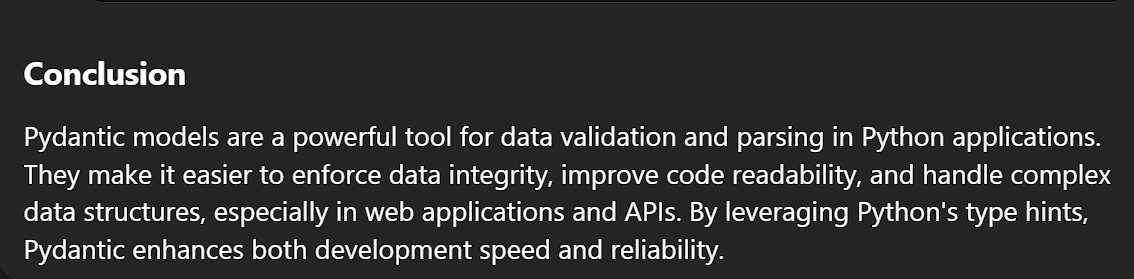




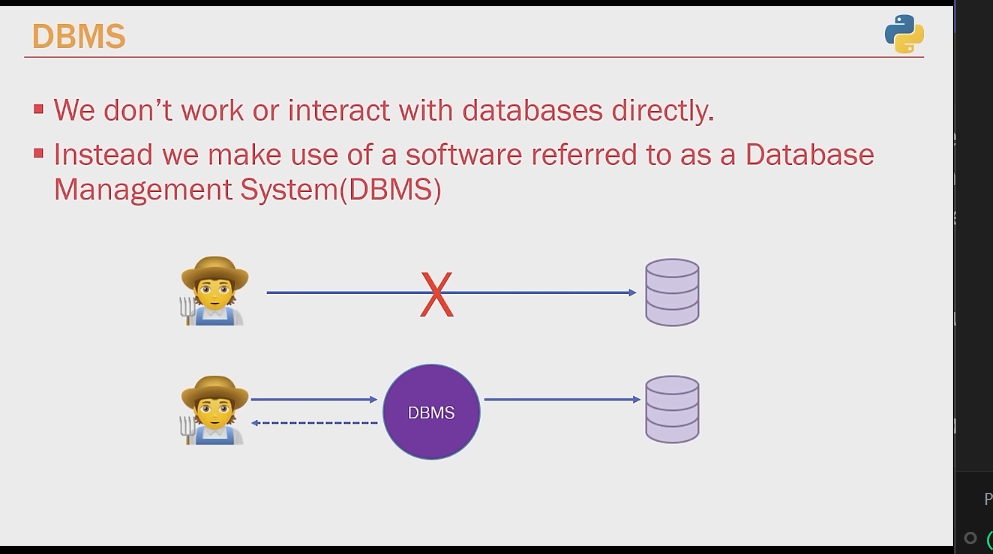


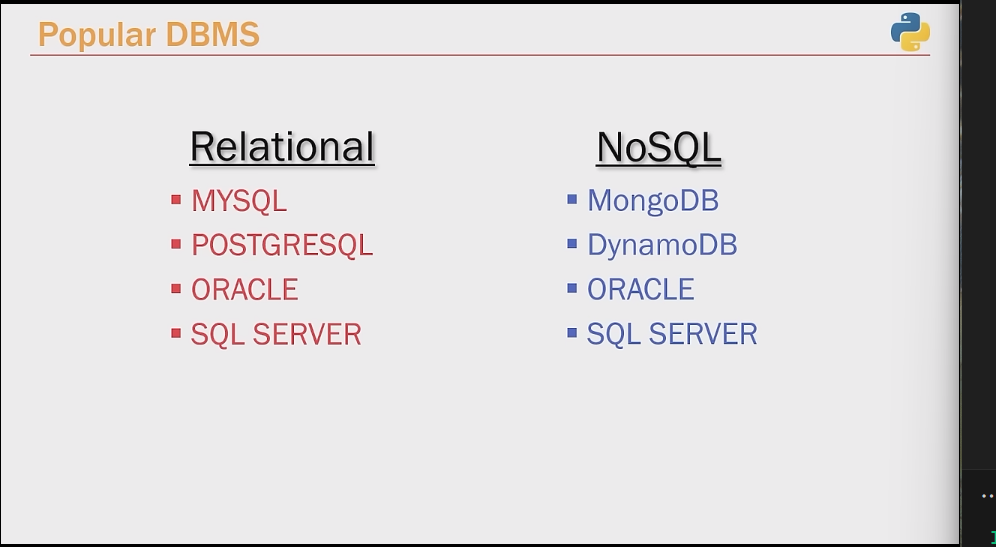


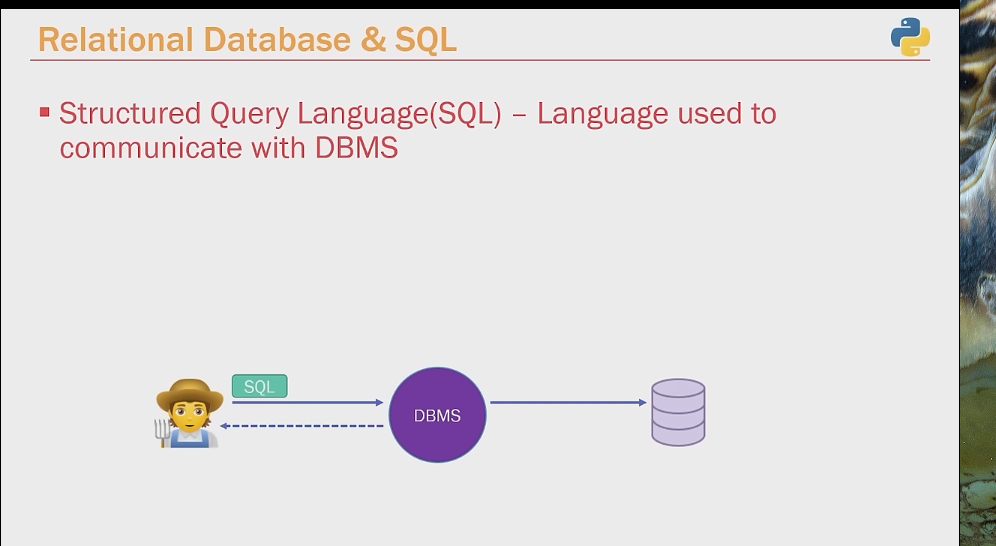


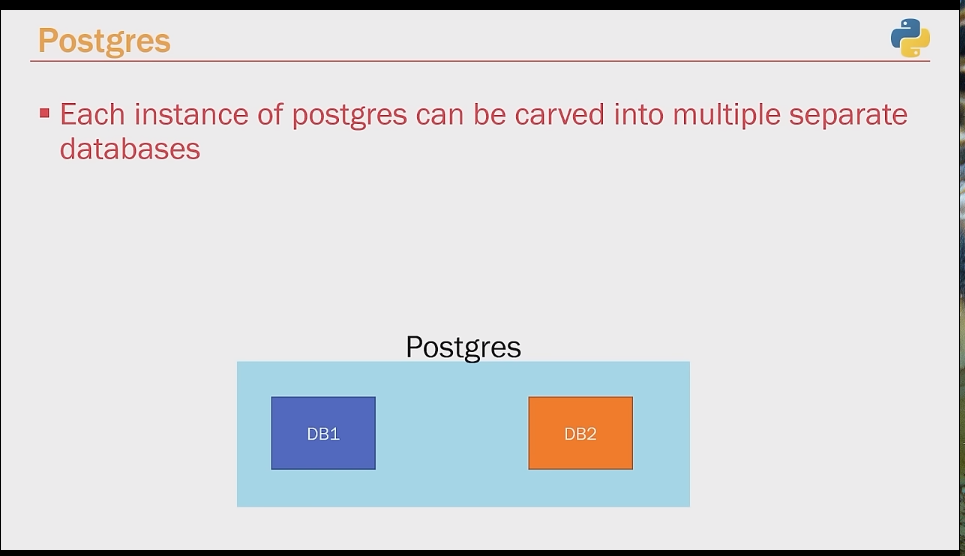


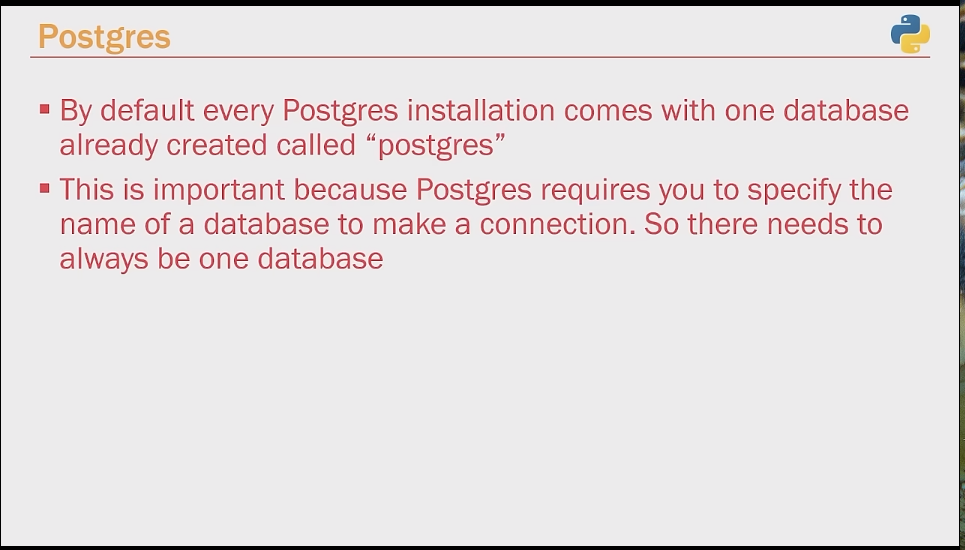
Databases.

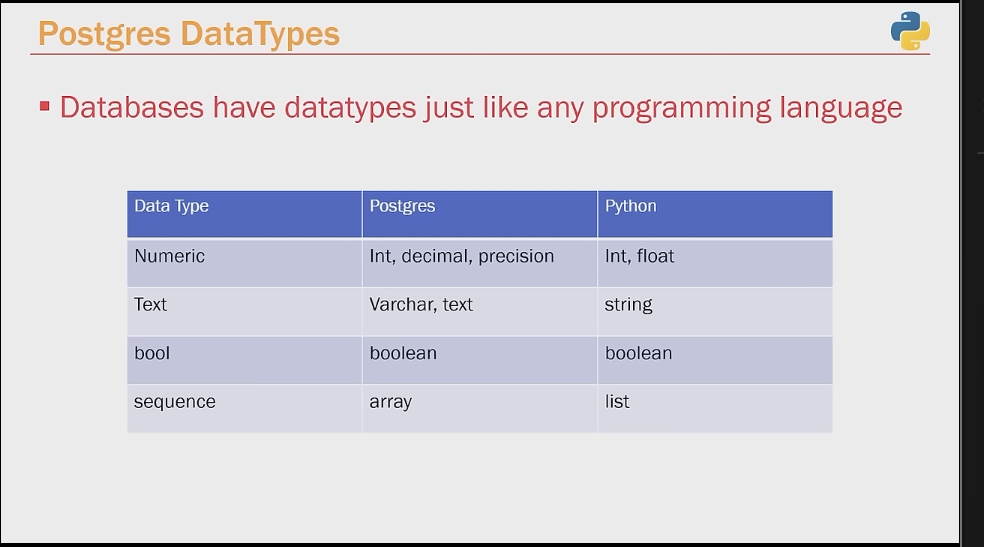


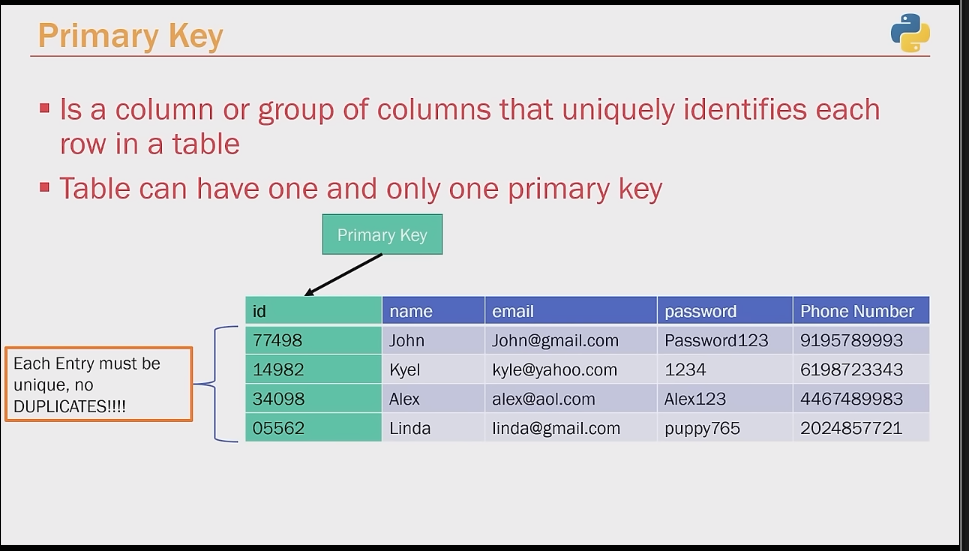


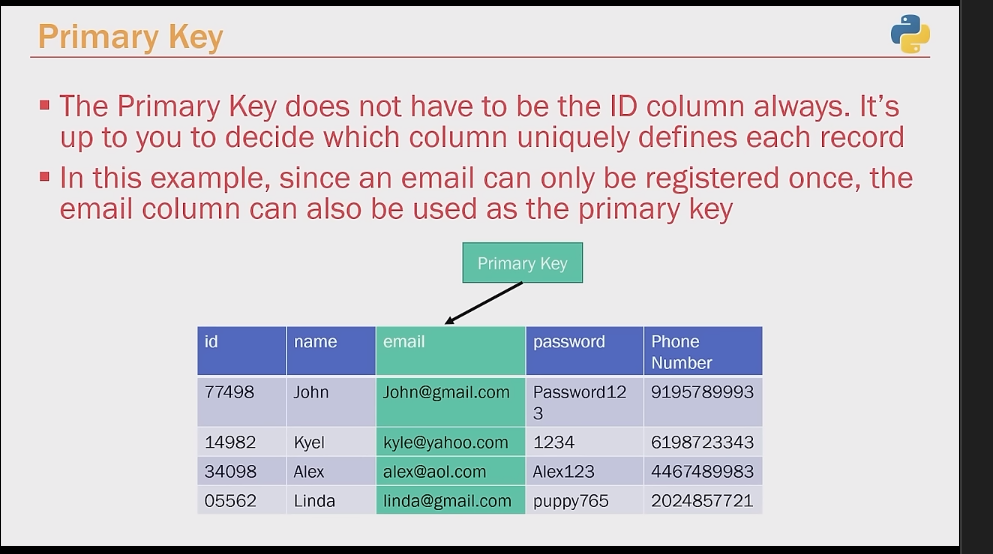




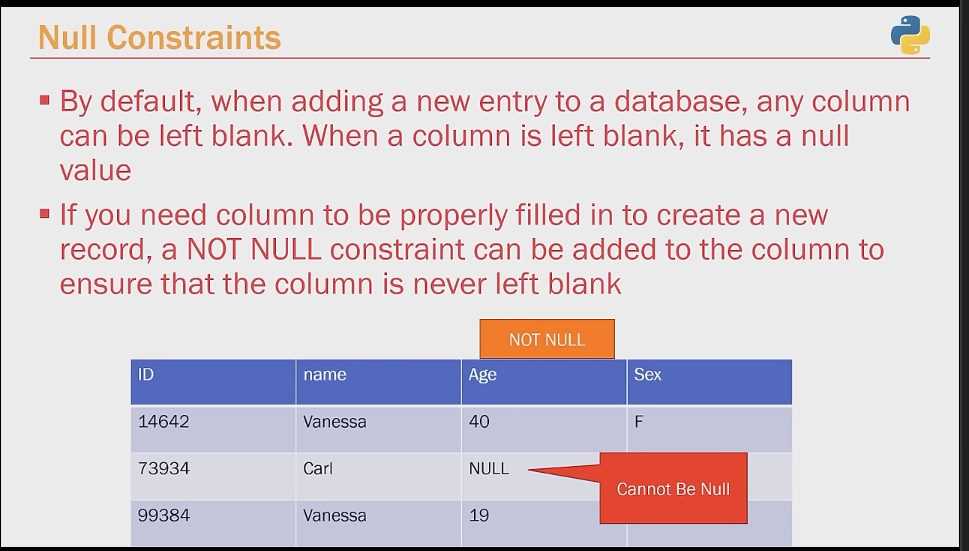






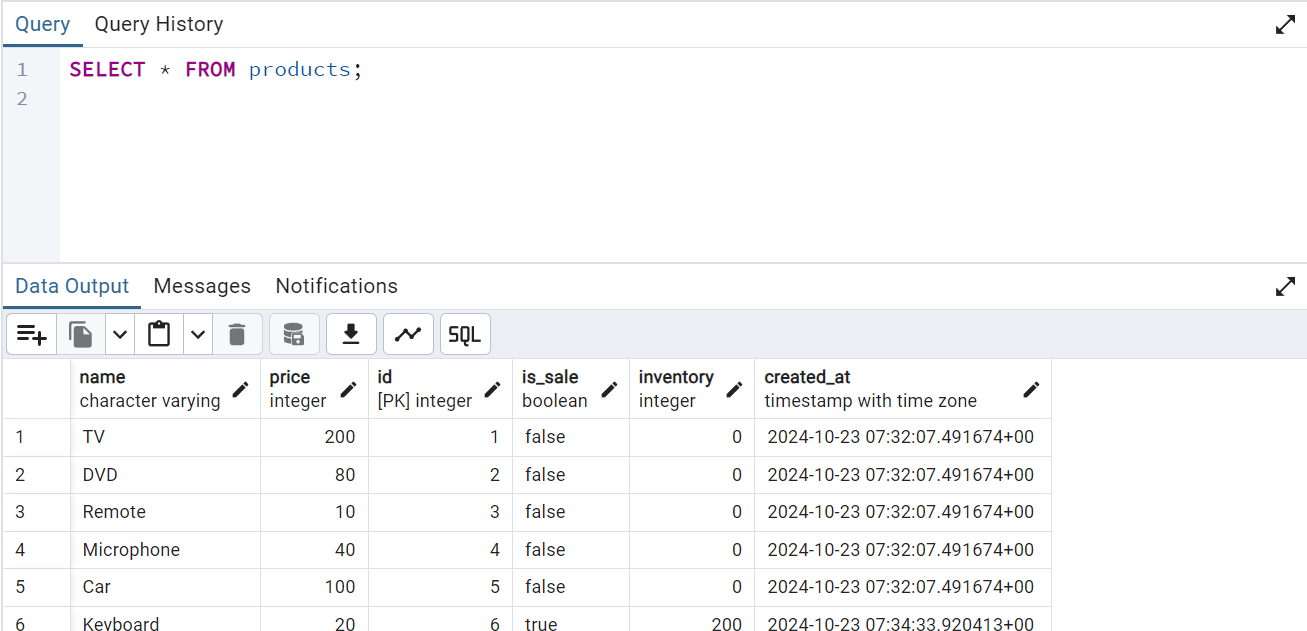




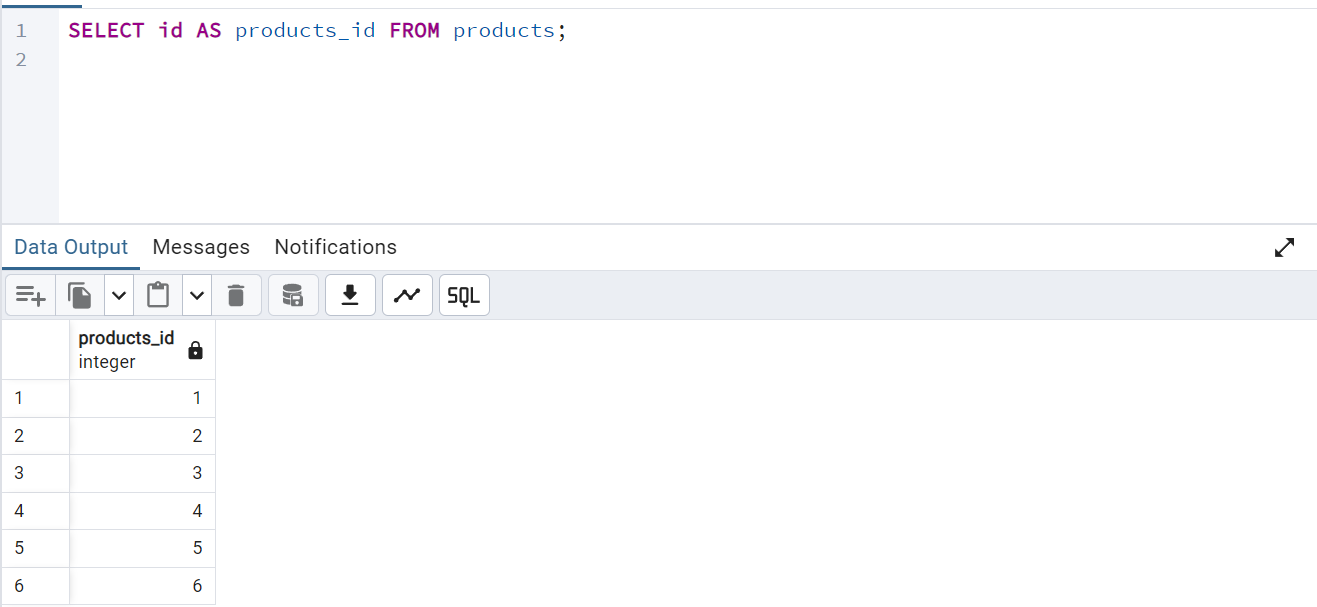


**SQL commands from the video**

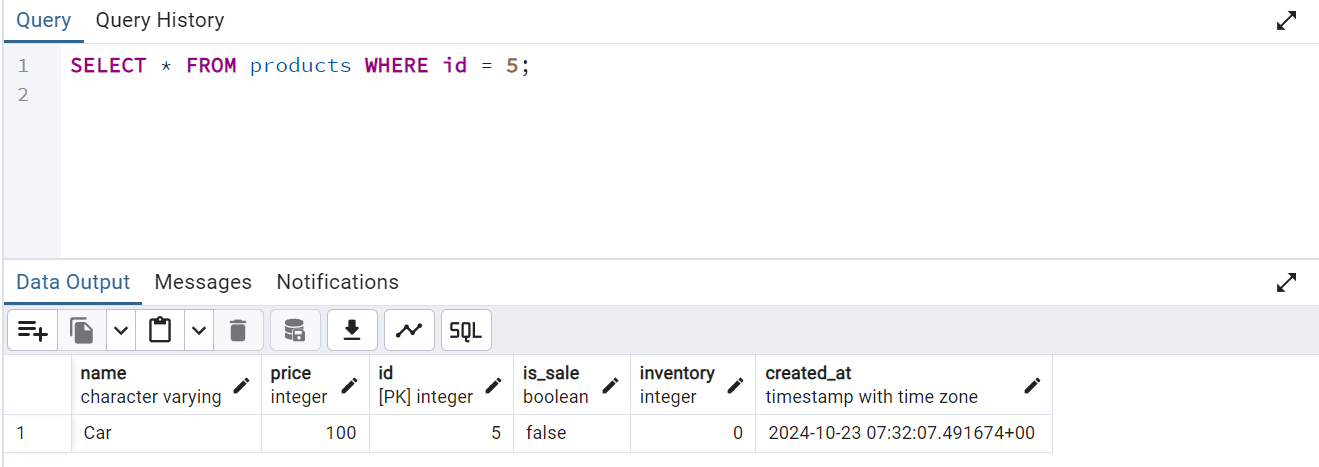
* Selecting all columns in a table:

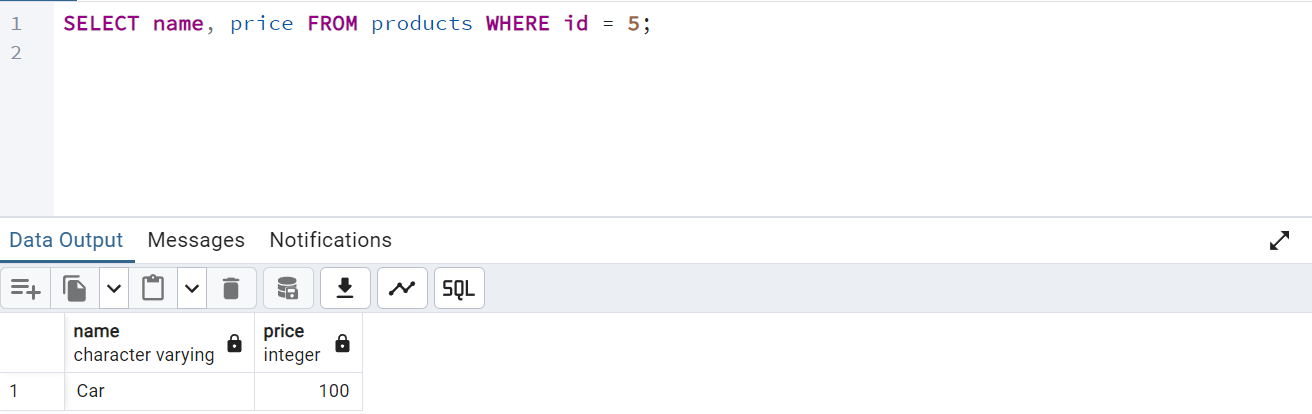


* Selecting a column with a different column title

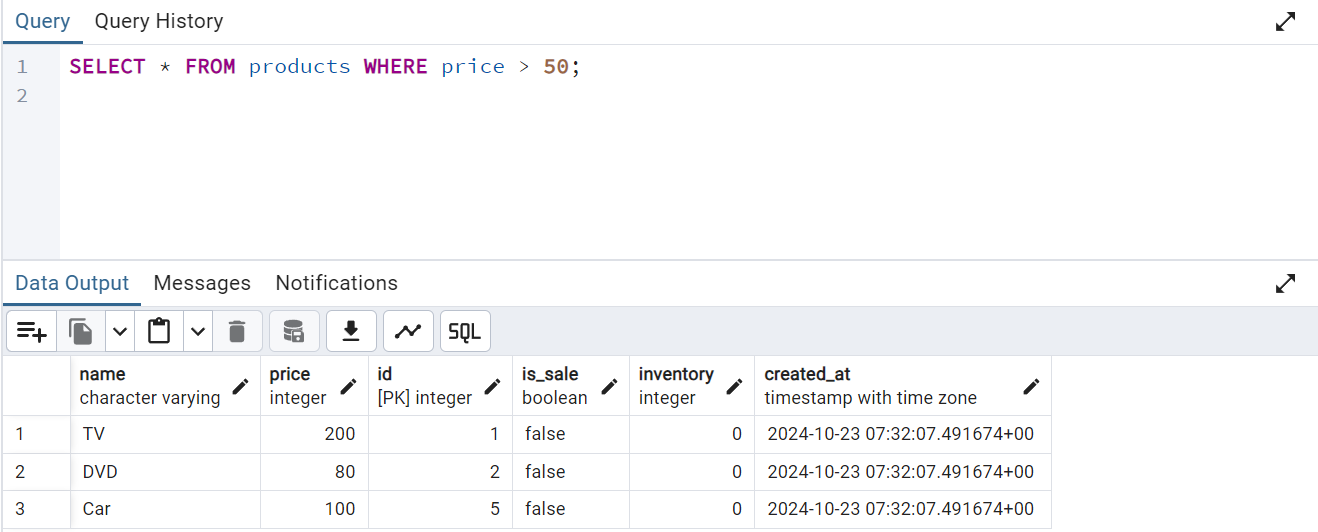


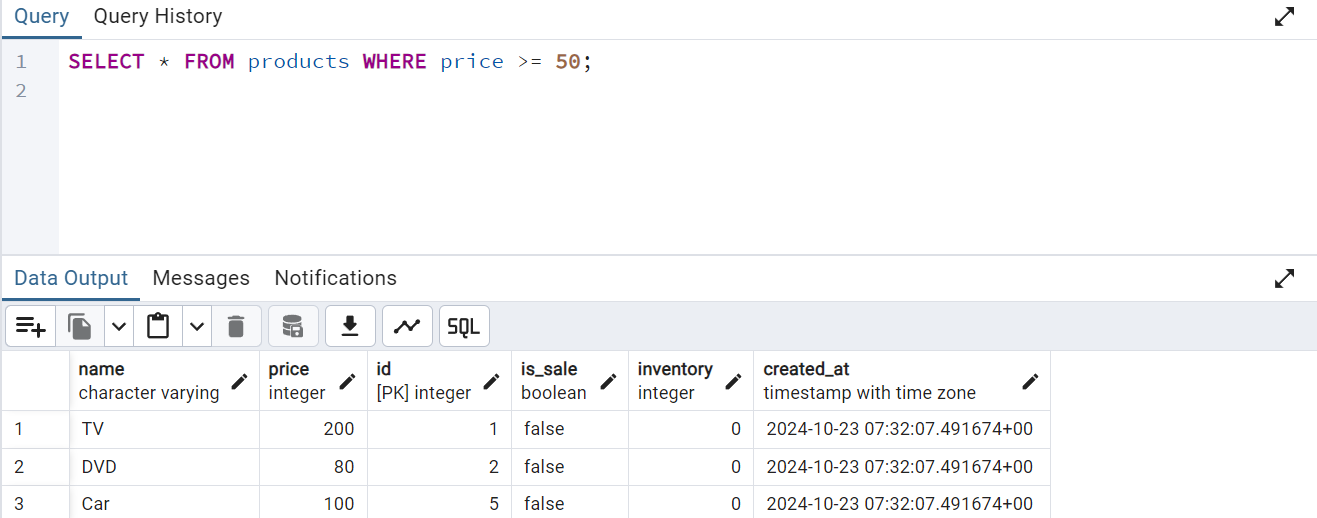
* Filtering queries based on attributes:



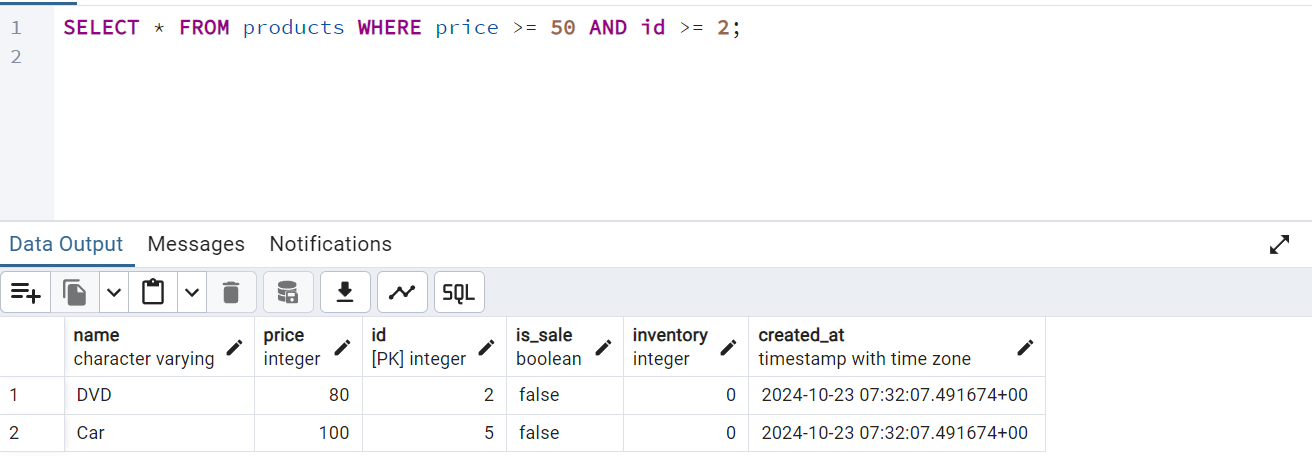


* Using operators other than the equal to sign for filtering:

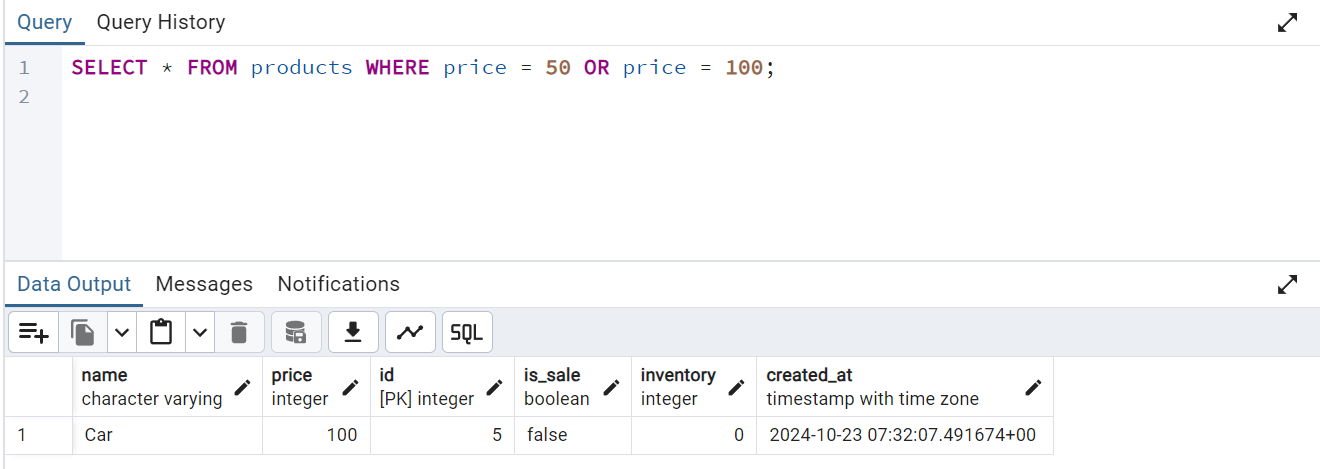




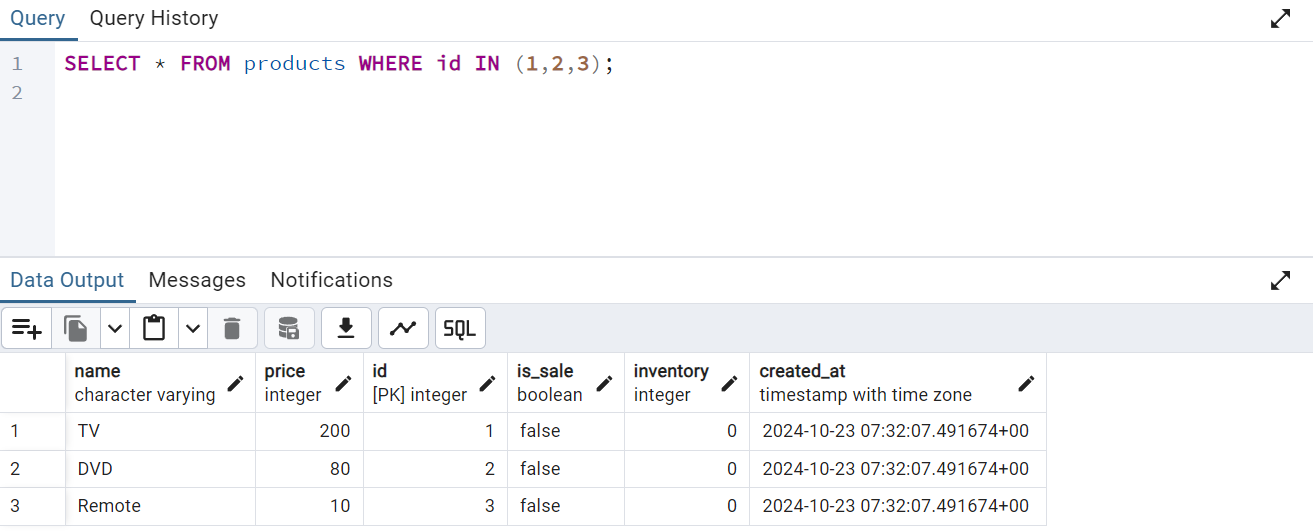
* Some more complex filtering queries:
* AND:



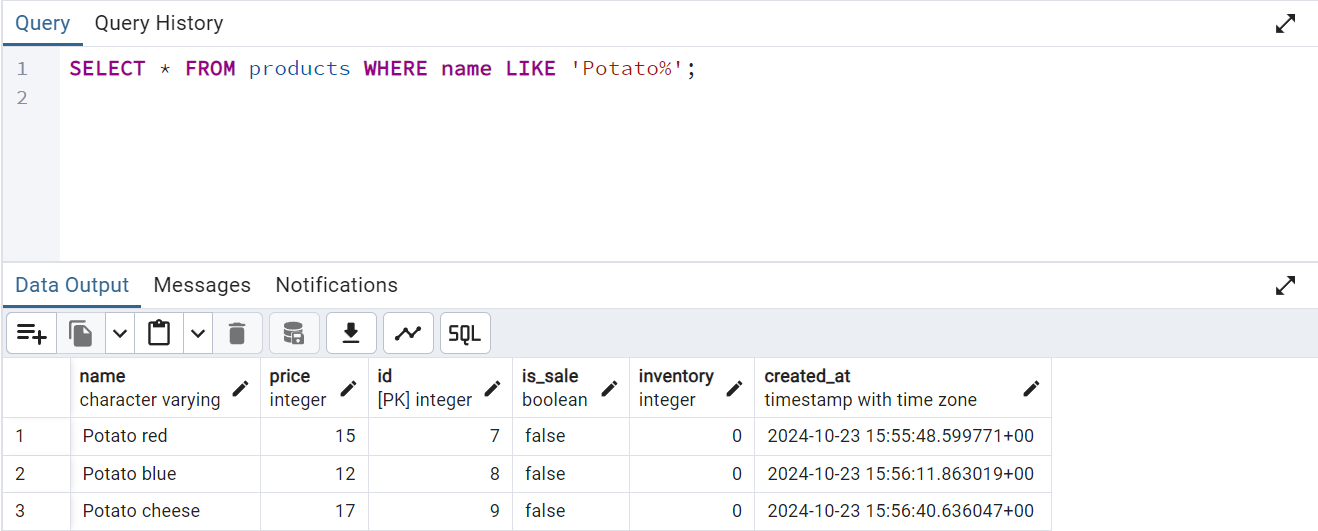
* OR:



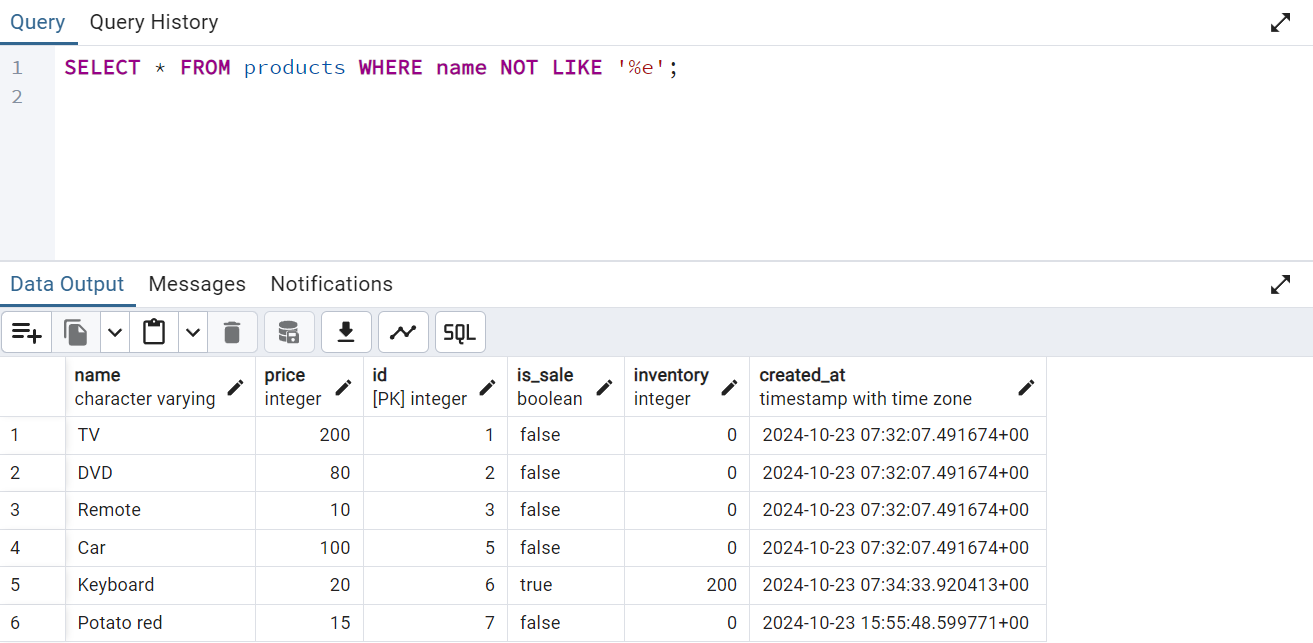
* IN:

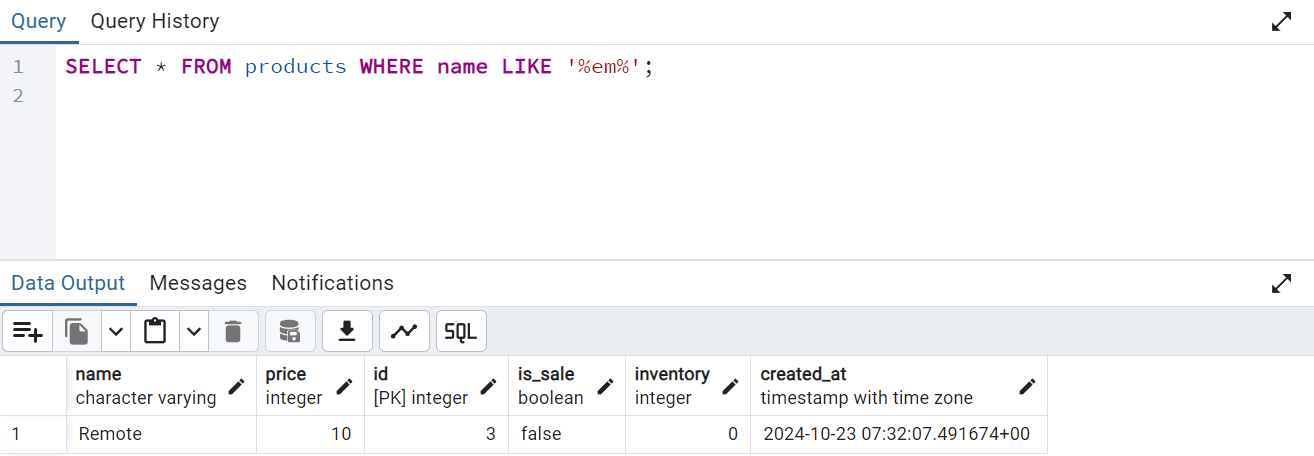


* LIKE (the percentage symbol in the query represents anything else after the word ‘Potato’):

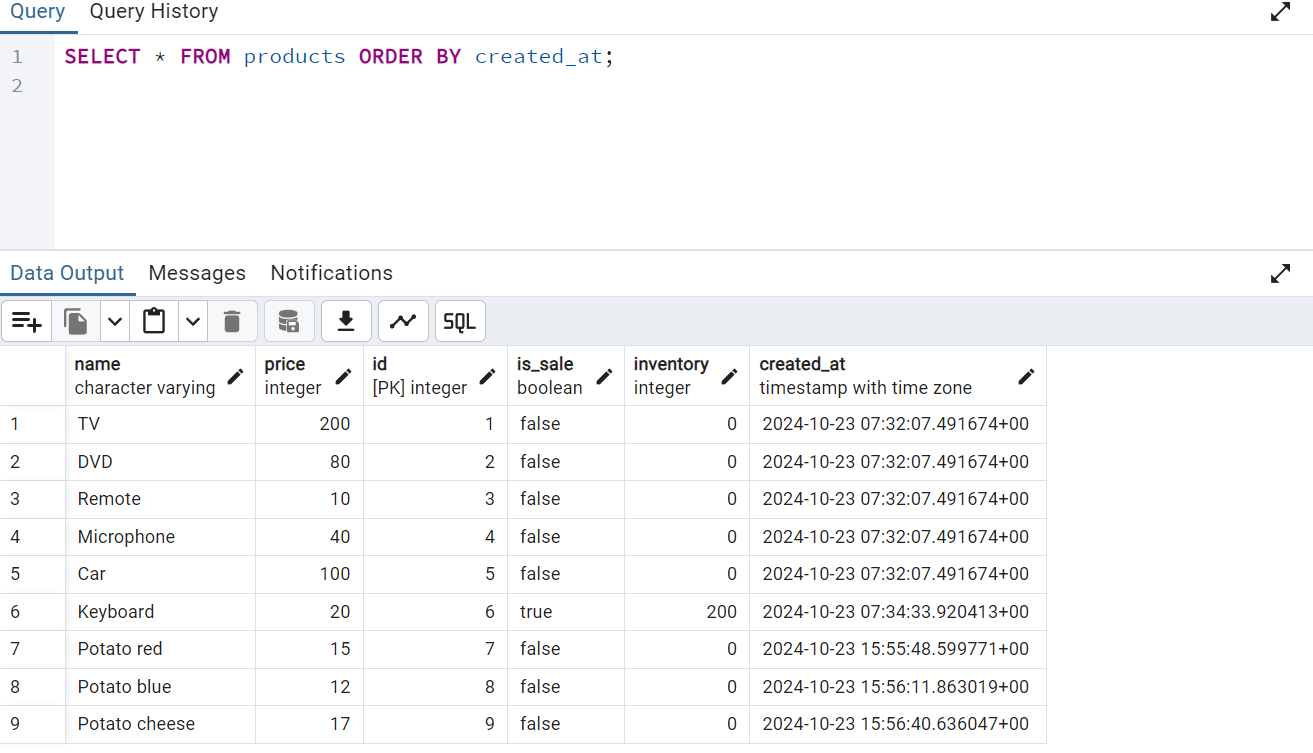


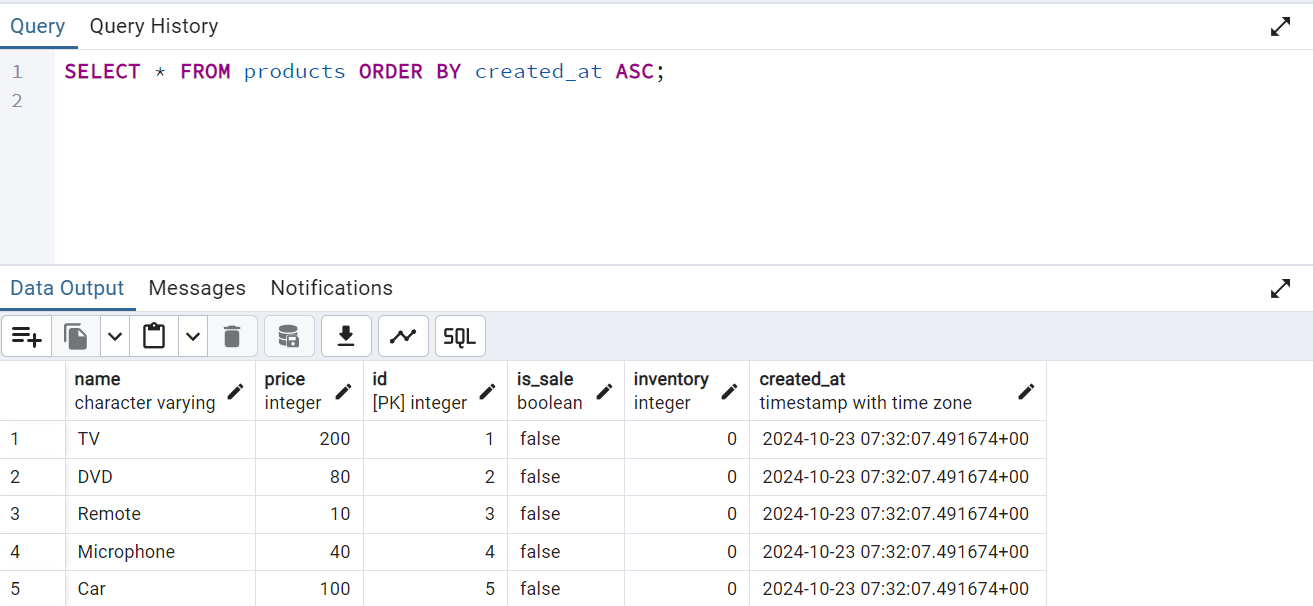


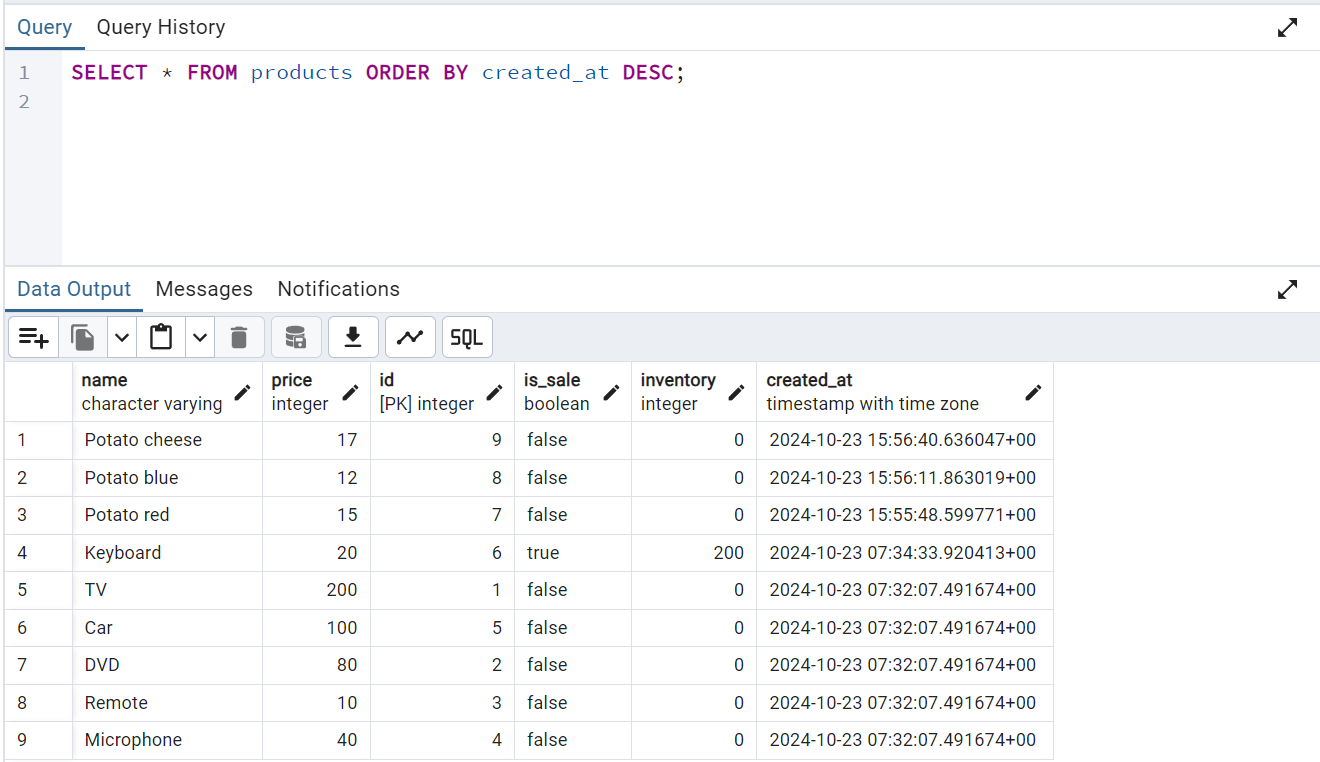


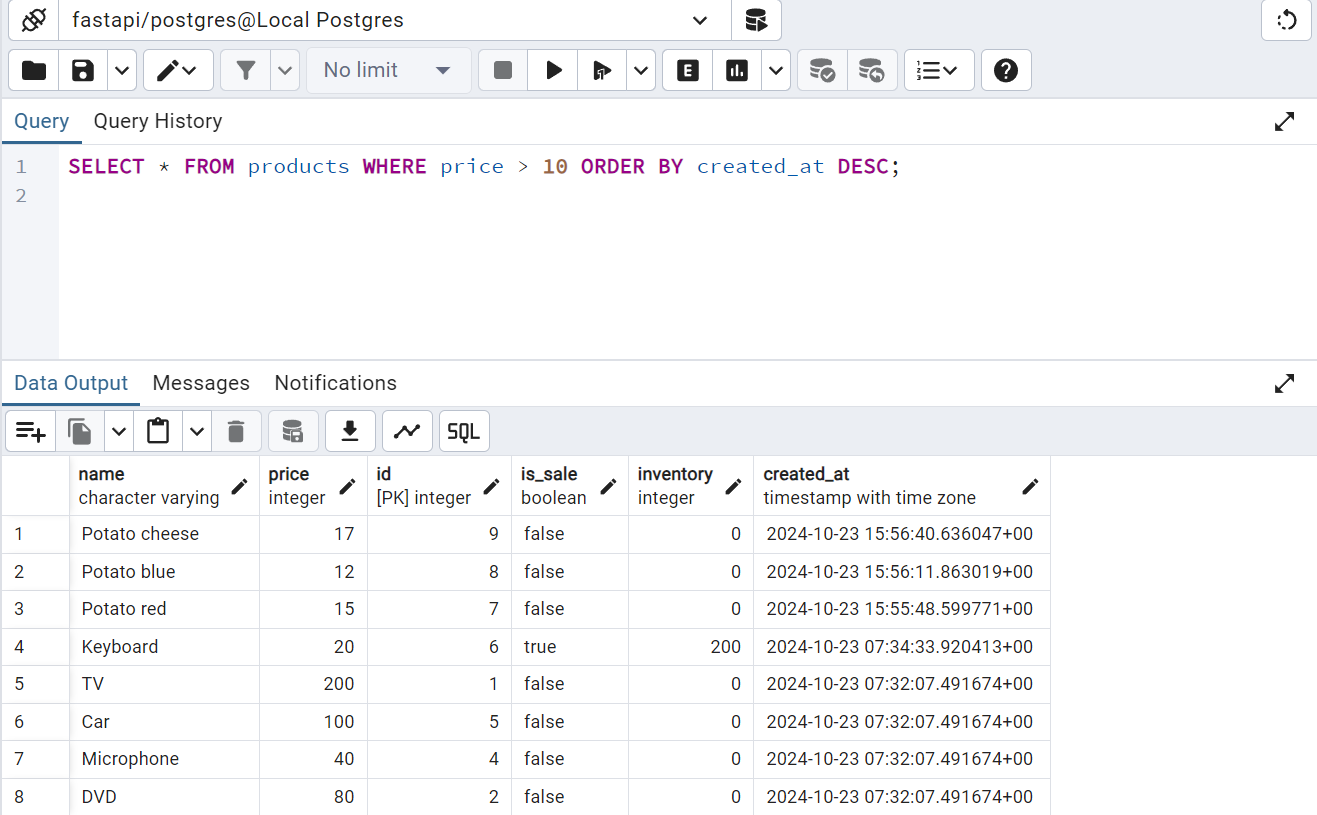


* ORDER BY:

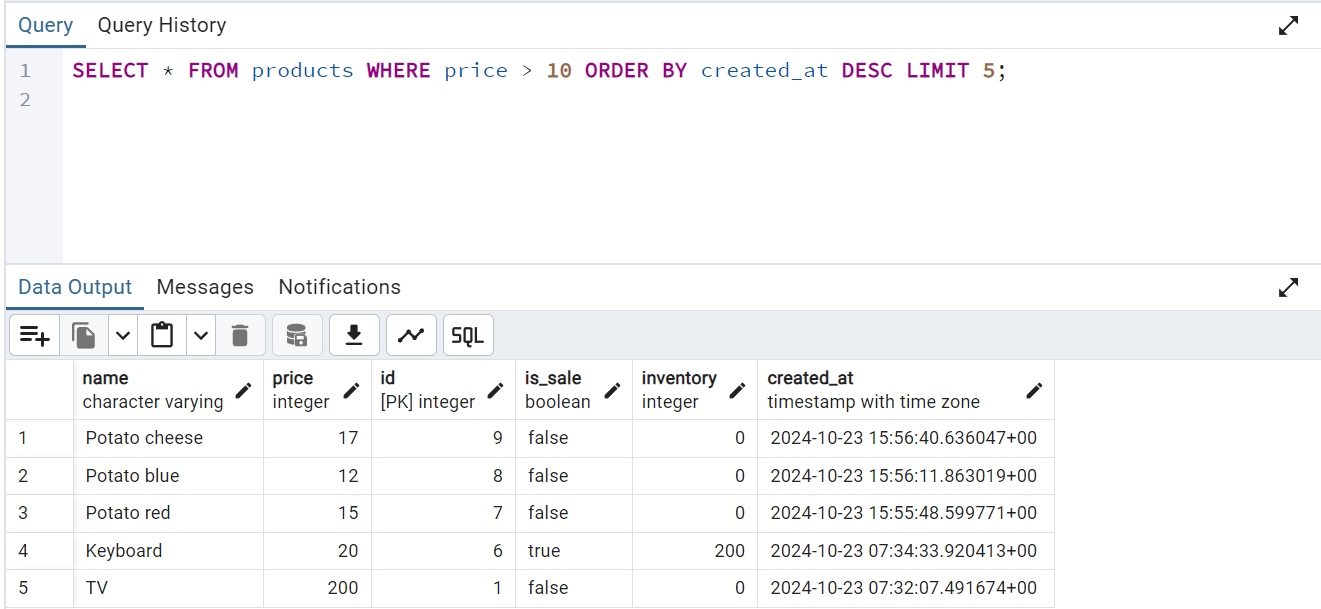




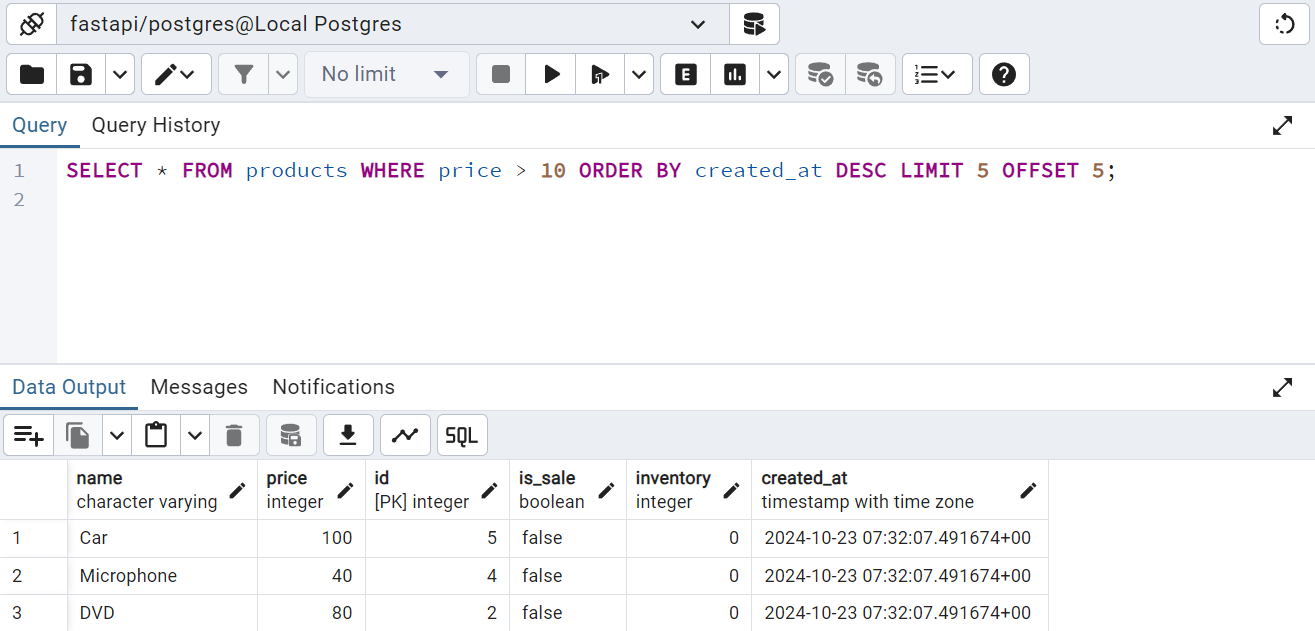




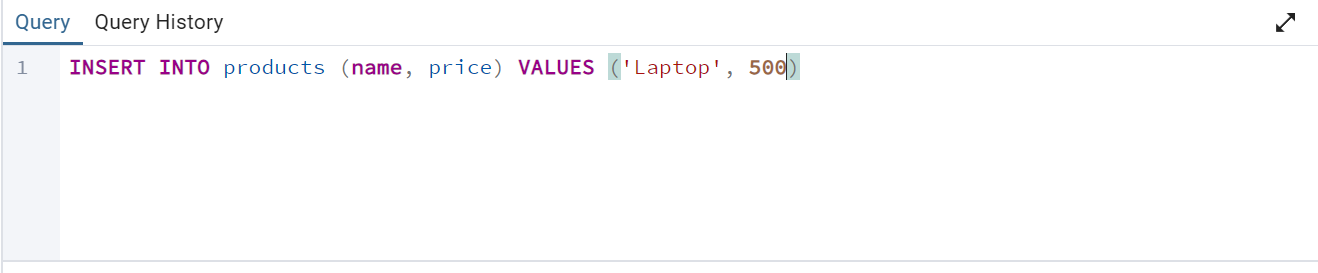
* LIMIT (to limit the number of rows after a query):



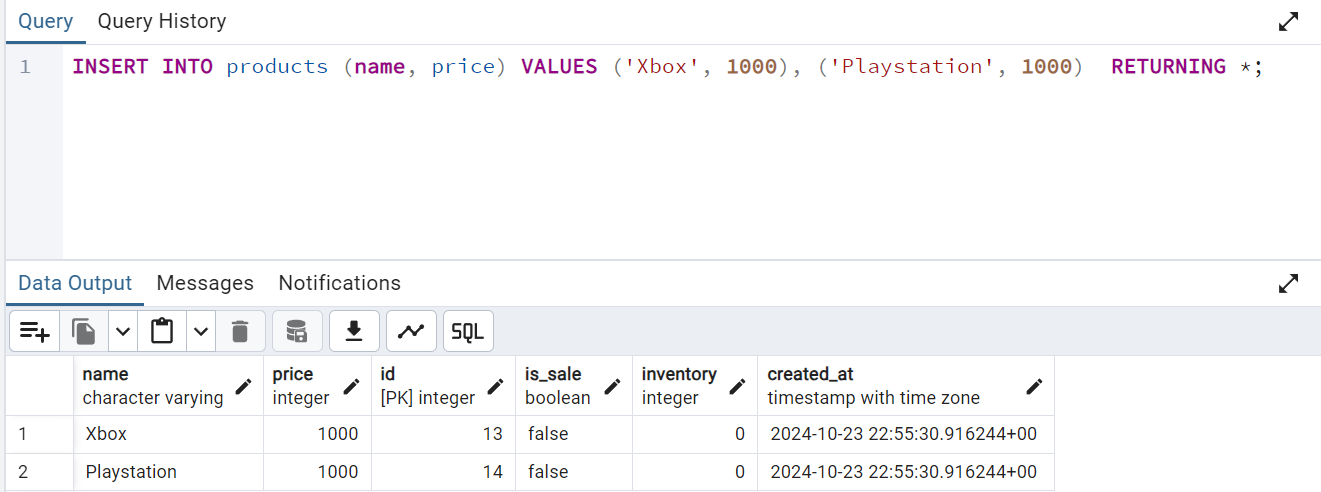
* OFFSET (to skip a number of rows):



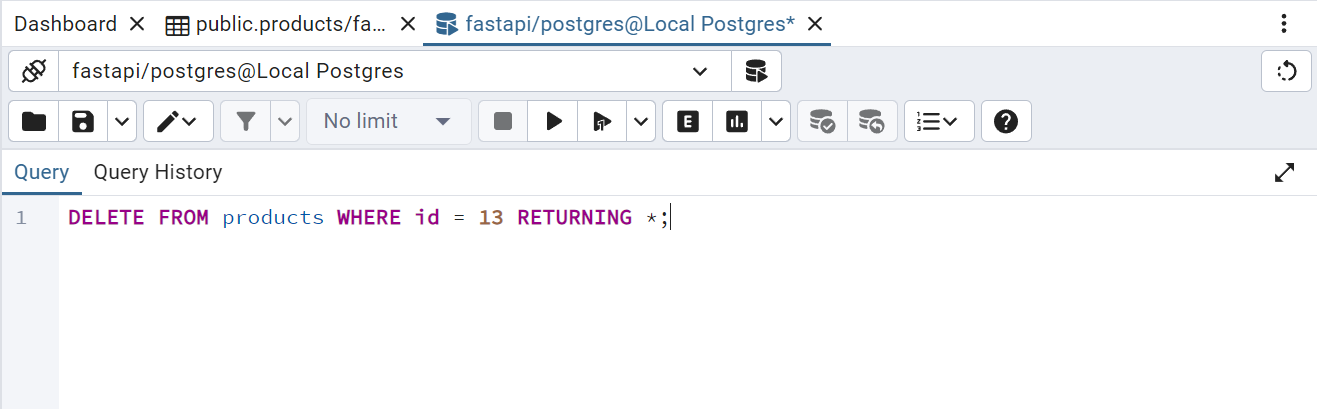
* Inserting Rows into a Table



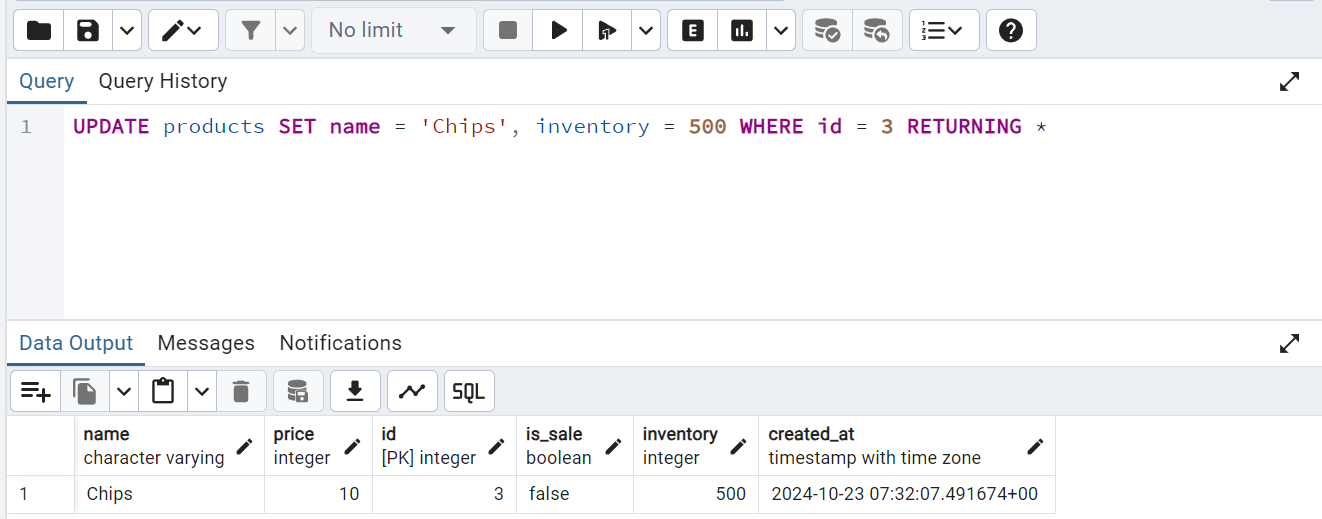
* Adding Multiple Rows:



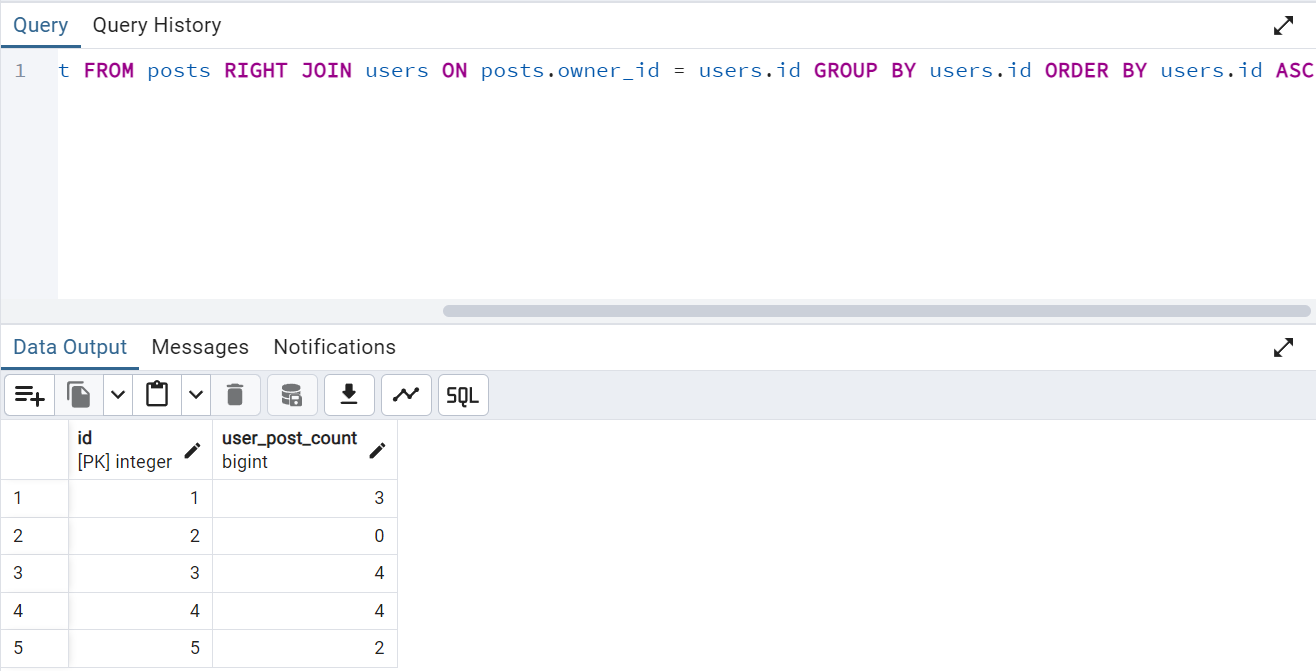
* Deleting Entries or Rows:

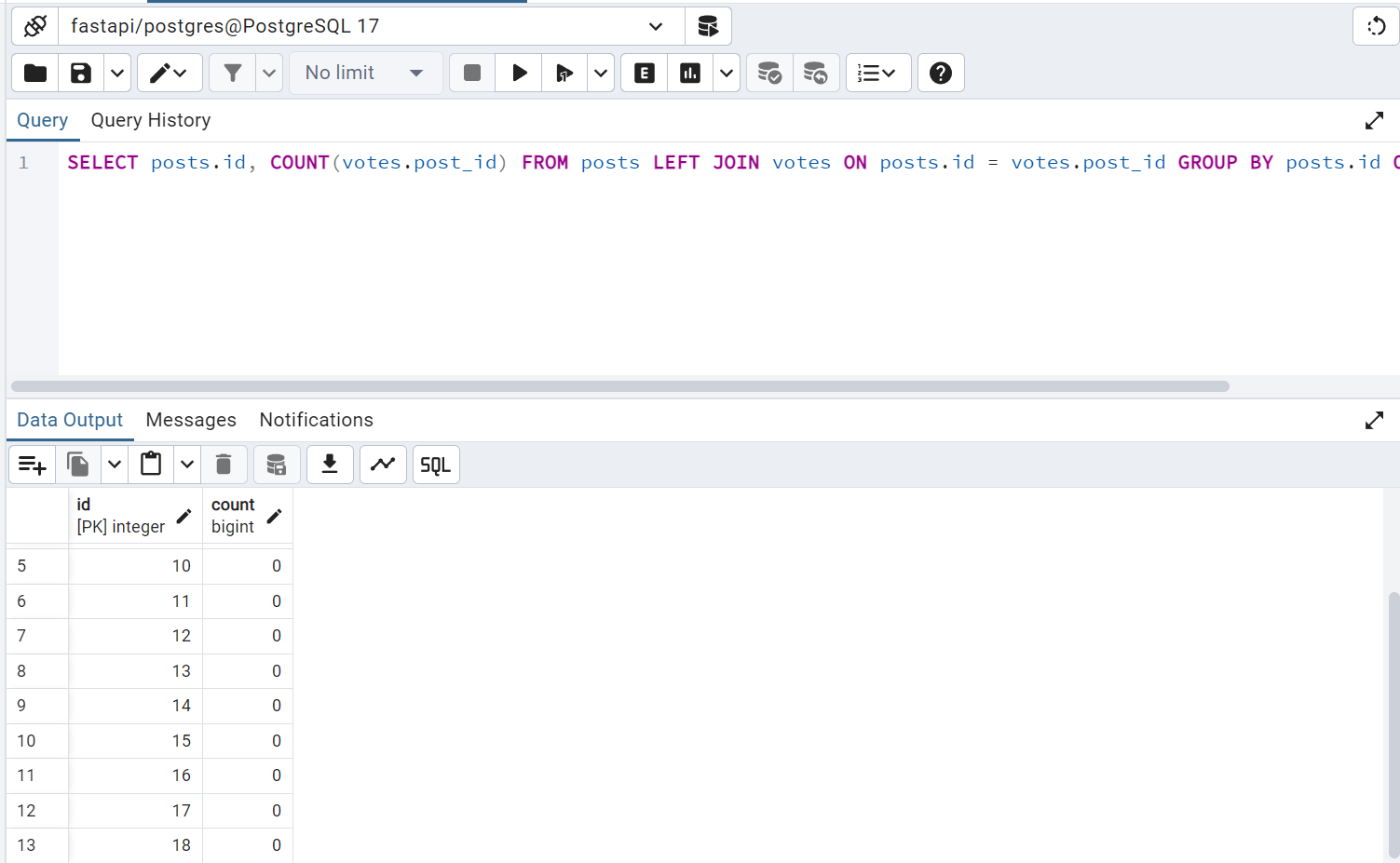


* Updating an Entry in the Table:

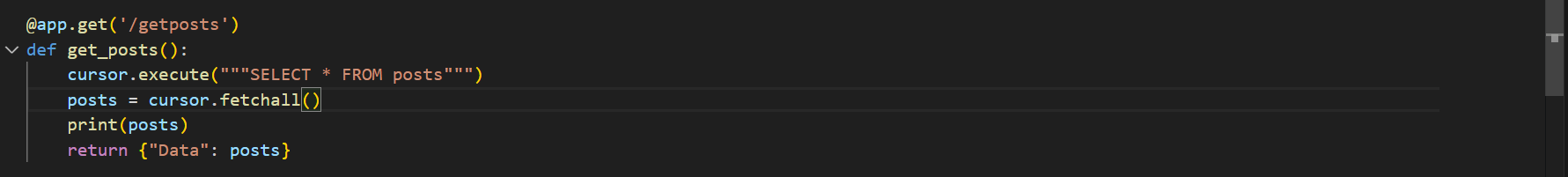


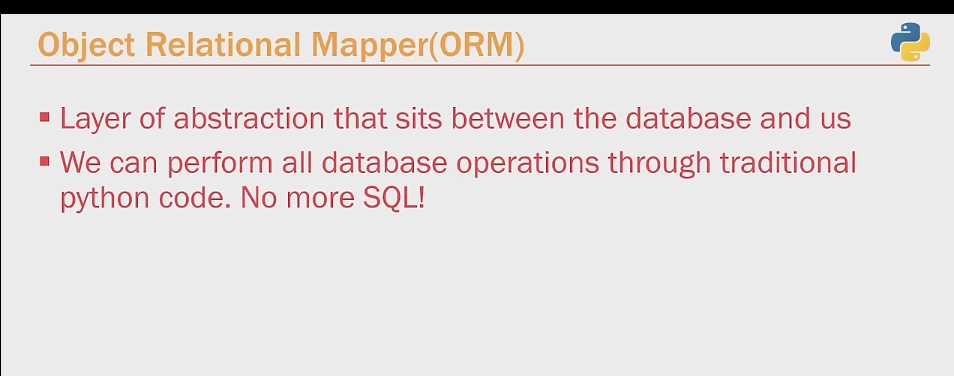
- The JOIN statement: used to join multiple tables in a database together.

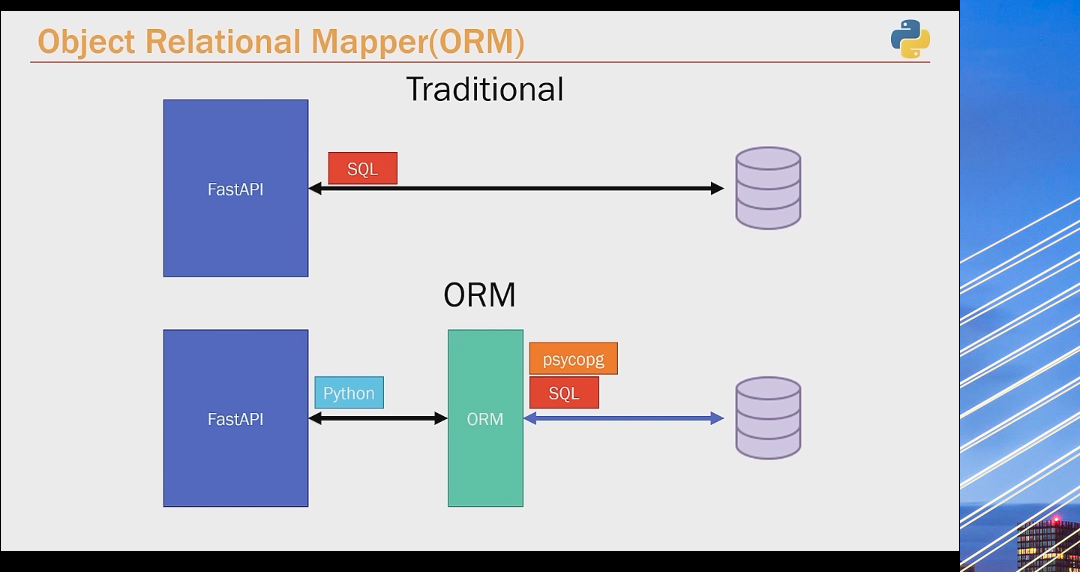




* Using SQL in the python environment (enabled by psycopg2):







ChatGPT

To use SQLAlchemy 2.0 with FastAPI to create a REST API, follow these steps to set up the FastAPI app, connect it to PostgreSQL, and define API routes for interacting with the database.

**1. Install Required Packages**

Make sure you have the necessary packages:

bash

Copy code

pip install fastapi sqlalchemy psycopg2-binary uvicorn

**2. Set Up the Database Connection and Models**

Define your database models using SQLAlchemy and set up a session with FastAPI dependency injection.

**database.py**

python

Copy code

from sqlalchemy import create\_engine

from sqlalchemy.orm import DeclarativeBase, sessionmaker

# Database URL - replace with your PostgreSQL details

DATABASE\_URL = "postgresql+psycopg2://username:password@localhost:5432/mydatabase"

# Create the engine and sessionmaker

engine = create\_engine(DATABASE\_URL, echo=True)

SessionLocal = sessionmaker(bind=engine)

class Base(DeclarativeBase):

pass

**models.py**

Define the User model.

python

Copy code

from sqlalchemy.orm import Mapped, mapped\_column

from sqlalchemy import Integer, String

from .database import Base

class User(Base):

\_\_tablename\_\_ = "users"

id: Mapped[int] = mapped\_column(Integer, primary\_key=True, autoincrement=True)

name: Mapped[str] = mapped\_column(String(50))

email: Mapped[str] = mapped\_column(String(100), unique=True)

**Create the Tables**

Run the following Python code once to create the tables:

python

Copy code

from .database import engine, Base

Base.metadata.create\_all(bind=engine)

**3. Set Up Dependency Injection for the Database Session**

Using FastAPI’s dependency injection, you can create a session for each request.

**dependencies.py**

python

Copy code

from .database import SessionLocal

from sqlalchemy.orm import Session

from fastapi import Depends

# Dependency function

def get\_db():

db = SessionLocal()

try:

yield db

finally:

db.close()

**4. Define Pydantic Schemas**

Define request and response schemas with Pydantic, which FastAPI will use to validate and serialize data.

**schemas.py**

python

Copy code

from pydantic import BaseModel

class UserBase(BaseModel):

name: str

email: str

class UserCreate(UserBase):

pass

class UserResponse(UserBase):

id: int

class Config:

orm\_mode = True

**5. Create CRUD Functions**

These functions will handle database interactions.

**crud.py**

python

Copy code

from sqlalchemy.orm import Session

from .models import User

from .schemas import UserCreate

def create\_user(db: Session, user: UserCreate):

db\_user = User(name=user.name, email=user.email)

db.add(db\_user)

db.commit()

db.refresh(db\_user)

return db\_user

def get\_user(db: Session, user\_id: int):

return db.query(User).filter(User.id == user\_id).first()

def get\_users(db: Session, skip: int = 0, limit: int = 10):

return db.query(User).offset(skip).limit(limit).all()

**6. Create FastAPI Endpoints**

Set up API endpoints in main.py.

**main.py**

python

Copy code

from fastapi import FastAPI, Depends, HTTPException

from sqlalchemy.orm import Session

from . import crud, models, schemas, dependencies

app = FastAPI()

@app.post("/users/", response\_model=schemas.UserResponse)

def create\_user(user: schemas.UserCreate, db: Session = Depends(dependencies.get\_db)):

return crud.create\_user(db=db, user=user)

@app.get("/users/{user\_id}", response\_model=schemas.UserResponse)

def read\_user(user\_id: int, db: Session = Depends(dependencies.get\_db)):

db\_user = crud.get\_user(db=db, user\_id=user\_id)

if db\_user is None:

raise HTTPException(status\_code=404, detail="User not found")

return db\_user

@app.get("/users/", response\_model=list[schemas.UserResponse])

def read\_users(skip: int = 0, limit: int = 10, db: Session = Depends(dependencies.get\_db)):

return crud.get\_users(db=db, skip=skip, limit=limit)

**7. Run the FastAPI App**

Use Uvicorn to run the FastAPI application:

bash

Copy code

uvicorn main:app --reload

**Summary of API Endpoints**

1. **Create User** - POST /users/ with JSON body {"name": "John Doe", "email": "john@example.com"}
2. **Get User by ID** - GET /users/{user\_id}
3. **List Users** - GET /users/ with optional query parameters skip and limit

This setup provides a fully functional API with SQLAlchemy 2.0 and PostgreSQL, wrapped in FastAPI.

QUESITION ABOUT THE dependencies.py FILE.

from .database import SessionLocal

from sqlalchemy.orm import Session

from fastapi import Depends # Dependency function

def get\_db():

db = SessionLocal()

try:

yield db

finally:

db.close()

What the use of this block of code?

**ChatGPT said:**

ChatGPT

This block of code defines a **dependency function** get\_db() for managing the lifecycle of a database session in a FastAPI application. Here’s how it works and why it’s important:

**Explanation of Each Part**

1. **SessionLocal()**: This line creates a new database session. SessionLocal is an instance of SQLAlchemy’s sessionmaker, configured to use the database engine. Each time SessionLocal() is called, a new Session object is created, allowing for interactions with the database.
2. **Dependency Injection with yield**:
   * The yield keyword is used in dependency functions in FastAPI to create an asynchronous "context manager" for handling resources, like a database session.
   * When yield db is called, FastAPI will provide db (the database session) to any endpoint that depends on get\_db().
3. **Closing the Session with finally**:
   * After the endpoint completes, the finally block ensures that db.close() is called, properly closing the session and releasing any resources associated with it.
   * Closing the session is critical because an open session can keep connections open in the database, leading to memory leaks or connection pool exhaustion.

**Purpose in FastAPI**

Using get\_db as a dependency ensures:

* **Scoped Session Lifecycle**: Each API request gets its own independent database session, which is closed after the request completes.
* **Automatic Session Management**: You don’t have to manually open or close sessions in each endpoint, reducing boilerplate code and preventing resource leaks.

**Usage in FastAPI Endpoints**

You can inject get\_db() as a dependency in your API route functions by using Depends(get\_db).

For example:

python

Copy code

from fastapi import FastAPI, Depends, HTTPException

from sqlalchemy.orm import Session

from . import crud, models, schemas, dependencies

app = FastAPI()

@app.get("/users/{user\_id}", response\_model=schemas.UserResponse)

def read\_user(user\_id: int, db: Session = Depends(dependencies.get\_db)):

db\_user = crud.get\_user(db=db, user\_id=user\_id)

if db\_user is None:

raise HTTPException(status\_code=404, detail="User not found")

return db\_user

In this example:

* db: Session = Depends(dependencies.get\_db) injects a session into the endpoint.
* FastAPI ensures the session db is available for the duration of the request, then closes it automatically after the request is processed.

This setup makes your code cleaner and improves database connection management.

4o

**FastAPI App with SQLAlchemy Queries**

Below are examples of different queries integrated with FastAPI routes, using the SQLAlchemy ORM in 2.0 style.

python

Copy code

# main.py

from fastapi import FastAPI, Depends, HTTPException

from sqlalchemy.orm import Session

from . import models, schemas, dependencies

from .database import engine

app = FastAPI()

# Create the tables (for demo purposes)

models.Base.metadata.create\_all(bind=engine)

@app.get("/users/", response\_model=list[schemas.UserResponse])

def read\_users(skip: int = 0, limit: int = 10, db: Session = Depends(dependencies.get\_db)):

"""Get a paginated list of users."""

stmt = select(models.User).offset(skip).limit(limit)

users = db.execute(stmt).scalars().all()

return users

@app.get("/users/{user\_id}", response\_model=schemas.UserResponse)

def read\_user(user\_id: int, db: Session = Depends(dependencies.get\_db)):

"""Get a single user by ID."""

stmt = select(models.User).where(models.User.id == user\_id)

user = db.execute(stmt).scalar\_one\_or\_none()

if user is None:

raise HTTPException(status\_code=404, detail="User not found")

return user

@app.post("/users/", response\_model=schemas.UserResponse)

def create\_user(user: schemas.UserCreate, db: Session = Depends(dependencies.get\_db)):

"""Create a new user."""

db\_user = models.User(name=user.name, email=user.email, age=user.age)

db.add(db\_user)

db.commit()

db.refresh(db\_user)

return db\_user

@app.put("/users/{user\_id}", response\_model=schemas.UserResponse)

def update\_user(user\_id: int, user: schemas.UserCreate, db: Session = Depends(dependencies.get\_db)):

"""Update an existing user by ID."""

stmt = select(models.User).where(models.User.id == user\_id)

db\_user = db.execute(stmt).scalar\_one\_or\_none()

if db\_user is None:

raise HTTPException(status\_code=404, detail="User not found")

# Update user details

db\_user.name = user.name

db\_user.email = user.email

db\_user.age = user.age

db.commit()

db.refresh(db\_user)

return db\_user

@app.delete("/users/{user\_id}")

def delete\_user(user\_id: int, db: Session = Depends(dependencies.get\_db)):

"""Delete a user by ID."""

stmt = select(models.User).where(models.User.id == user\_id)

db\_user = db.execute(stmt).scalar\_one\_or\_none()

if db\_user is None:

raise HTTPException(status\_code=404, detail="User not found")

db.delete(db\_user)

db.commit()

return {"message": "User deleted successfully"}

**Explanation of Each Route**

1. **read\_users**:
   * Retrieves a list of users with pagination (using skip and limit).
   * Executes a select query and returns a list of users.
2. **read\_user**:
   * Retrieves a single user by ID.
   * Uses scalar\_one\_or\_none() to fetch exactly one or return None if the user is not found.
3. **create\_user**:
   * Inserts a new user using db.add() and commits the transaction.
   * db.refresh(db\_user) reloads the instance with the latest data, including auto-generated fields like id.
4. **update\_user**:
   * First checks if the user exists by querying with select.
   * Updates the fields of the db\_user instance if found, then commits and refreshes it.
5. **delete\_user**:
   * Checks if the user exists, then deletes the record and commits the transaction.

Each route uses SQLAlchemy 2.0 syntax with the select function and modern methods for session handling, making the code cleaner and more efficient within FastAPI’s dependency injection framework.

**SQLAlchemy itself doesn’t handle database migrations directly, but you can add migration capabilities to your API with Alembic, a lightweight database migration tool developed by the same creator as SQLAlchemy. Alembic integrates seamlessly with SQLAlchemy to handle** schema changes like creating tables, adding or removing columns, and other database schema migrations.

Here's how to set up Alembic in your FastAPI project:

**Step 1: Install Alembic**

bash

Copy code

pip install alembic

**Step 2: Initialize Alembic**

In your project root, initialize Alembic. This will create an alembic folder containing configuration and migration scripts.

bash

Copy code

alembic init alembic

**Step 3: Configure Alembic to Use SQLAlchemy Models**

1. Open the alembic.ini file and find the sqlalchemy.url setting. Replace it with your database URL, or set it to read from an environment variable:

ini

Copy code

sqlalchemy.url = postgresql+psycopg2://username:password@localhost:5432/database\_name

1. Alternatively, edit the env.py file in the alembic folder to read the database URL from your FastAPI project’s settings. Modify the env.py file to include:

python

Copy code

# env.py

from logging.config import fileConfig

from sqlalchemy import engine\_from\_config, pool

from alembic import context

from myapp.database import Base # Import Base from your project

from myapp.models import \* # Import all models

config = context.config

fileConfig(config.config\_file\_name)

# Replace with the path to your models' Base

target\_metadata = Base.metadata

def run\_migrations\_online():

connectable = engine\_from\_config(

config.get\_section(config.config\_ini\_section),

prefix="sqlalchemy.",

poolclass=pool.NullPool,

)

with connectable.connect() as connection:

context.configure(connection=connection, target\_metadata=target\_metadata)

with context.begin\_transaction():

context.run\_migrations()

if context.is\_offline\_mode():

context.configure(url=config.get\_main\_option("sqlalchemy.url"))

with context.begin\_transaction():

context.run\_migrations()

else:

run\_migrations\_online()

**Step 4: Generate a Migration Script**

After creating or modifying models in your SQLAlchemy models.py, you can autogenerate migration scripts with Alembic.

Run this command to create a new migration script based on changes in the models.py file:

alembic revision --autogenerate -m "Create User table"

This will create a new migration file in alembic/versions/ with your changes.

**Step 5: Apply the Migrations**

To apply your migrations to the database, run:

alembic upgrade head

This command applies all pending migrations up to the latest one.

**Example Workflow in FastAPI**

Now, whenever you make changes to your models (e.g., adding a new column or table), you can:

1. Generate a new migration script:

bash

Copy code

alembic revision --autogenerate -m "Add new field to User table"

1. Apply the migration:

bash

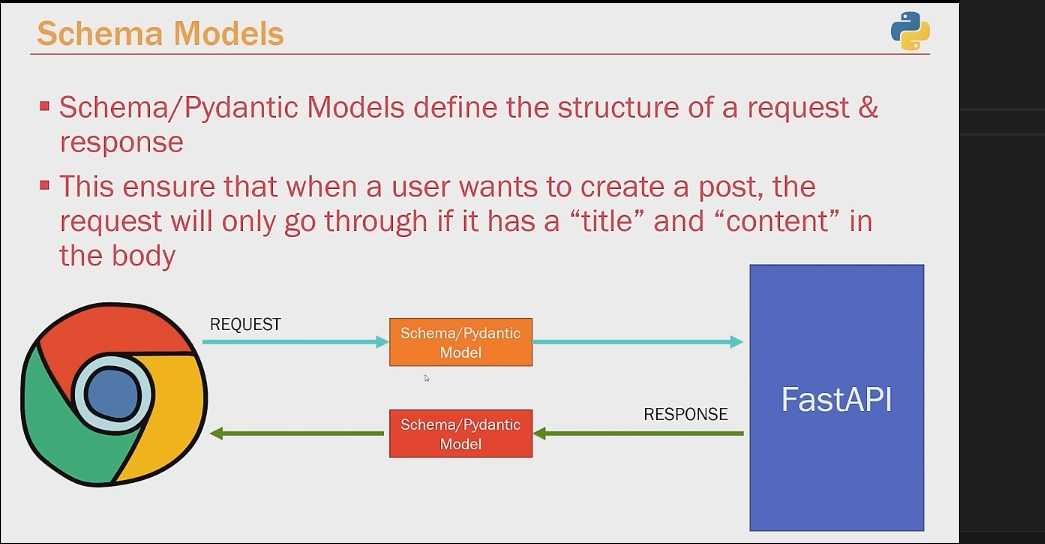
Copy code

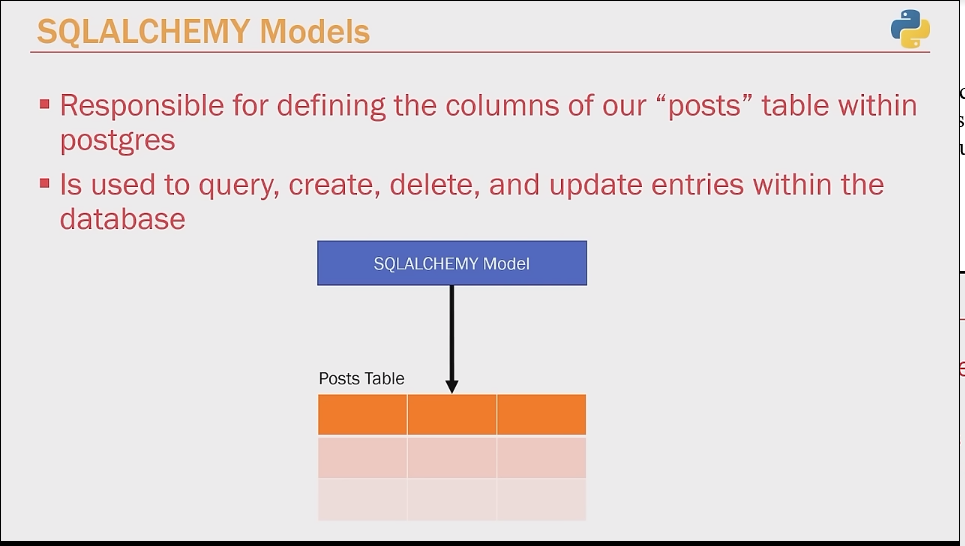
alembic upgrade head

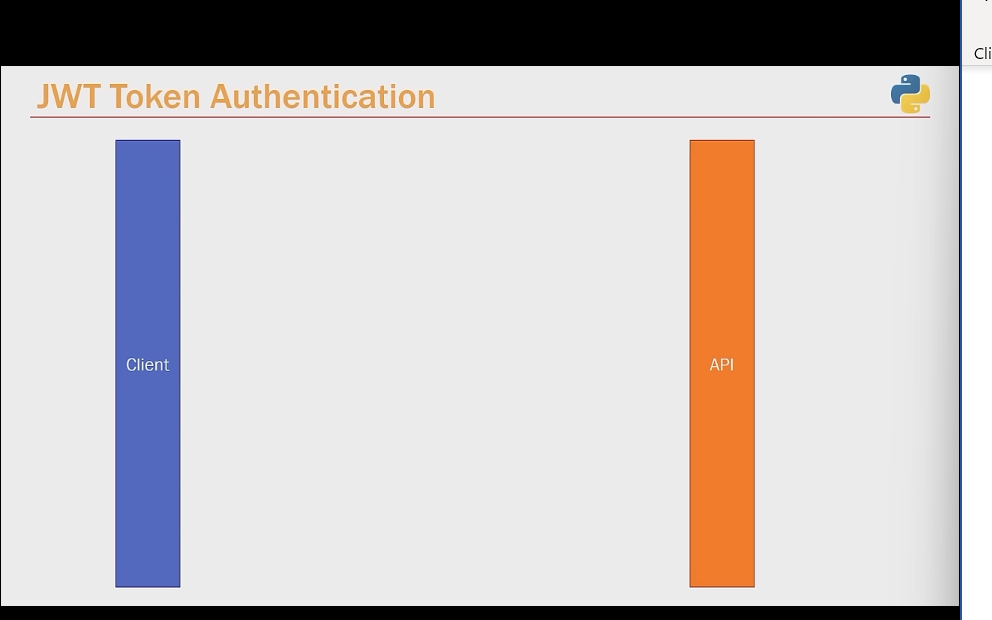
**Summary**

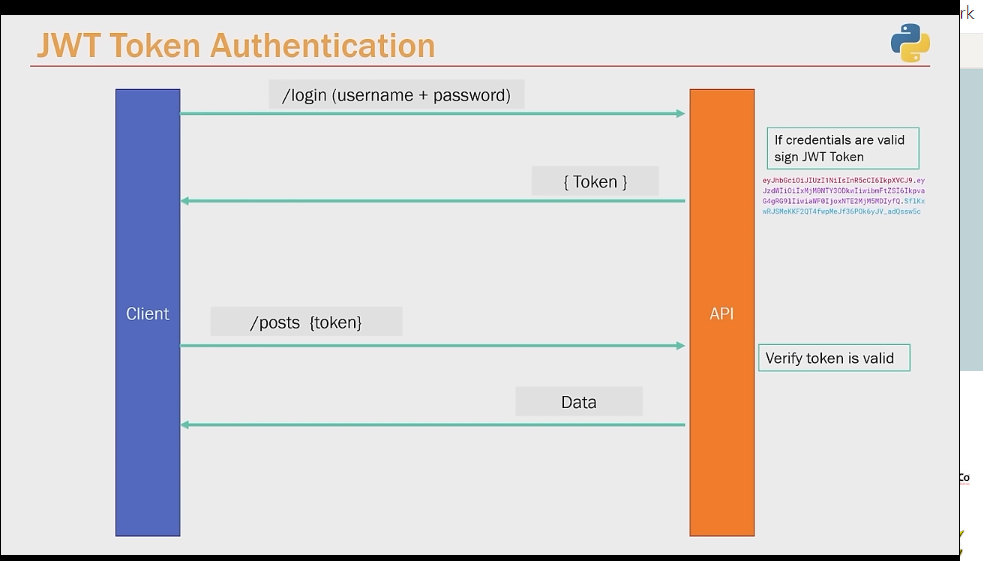
Using Alembic alongside SQLAlchemy:

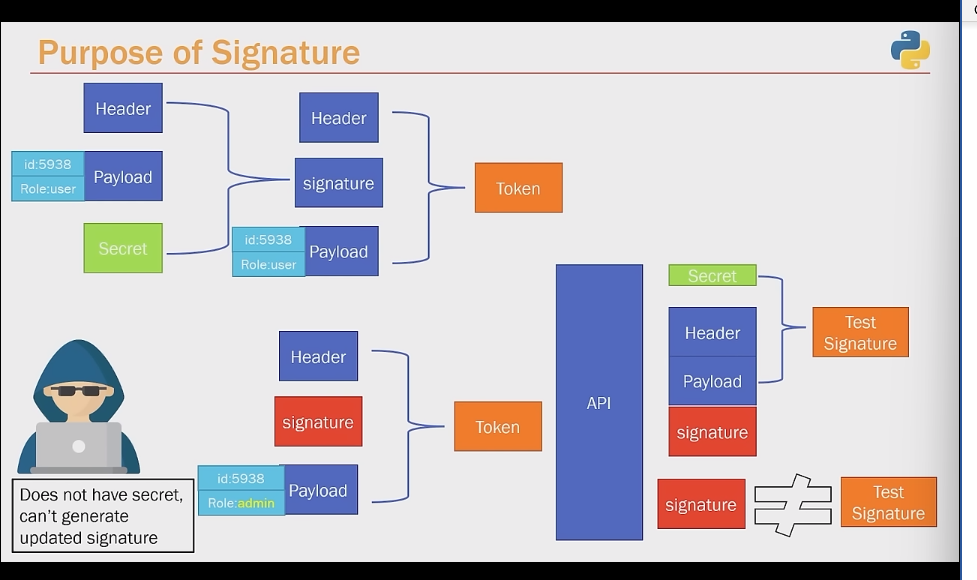
* Simplifies schema migrations without directly altering the database schema manually.
* Enables tracking of schema changes across versions.
* Integrates smoothly into a FastAPI app setup for efficient and versioned database schema management.

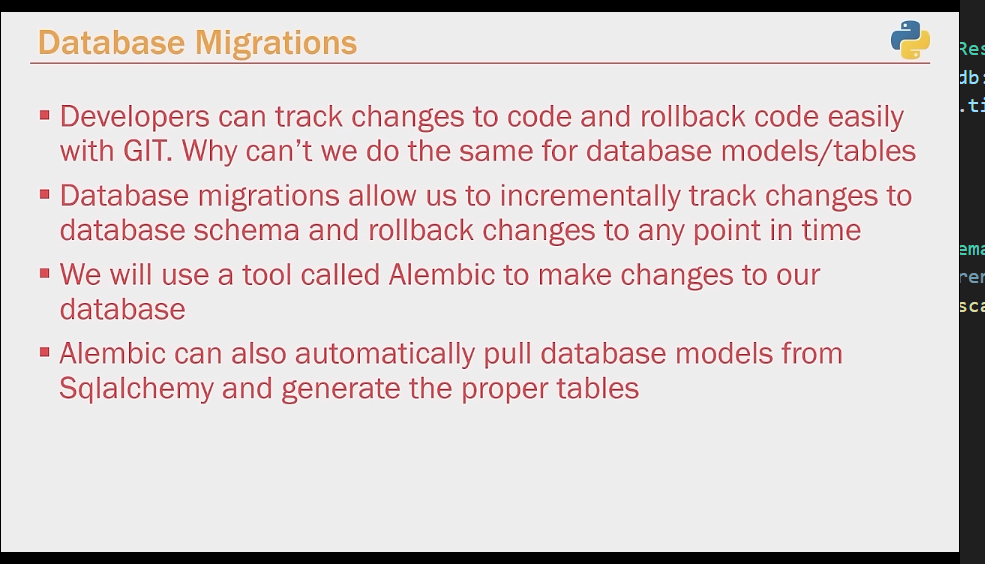




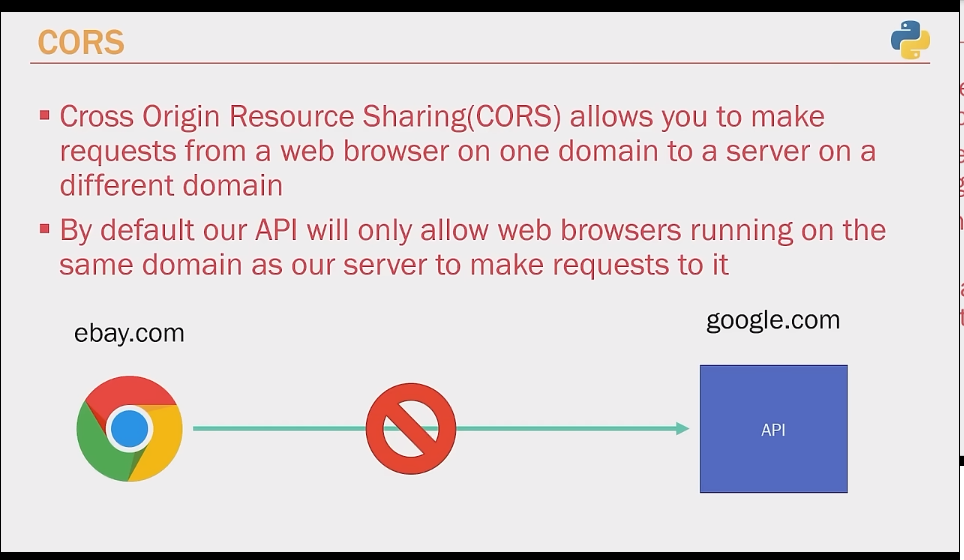








## CORS Policy



## Creating a requirements.txt file with pip

