FASTAPI CRASH COURSE

- **Navigate to the desired directory**: Open your terminal (or command prompt) and navigate to the directory where you want to create the virtual environment.

Example:

```
bash
Copy code
cd /path/to/your/directory
```

- **Create the virtual environment**: Run the following command to create a virtual environment in the directory. You can name the environment (e.g., `venv` or any other name you prefer):

```
bash
Copy code
python3 -m venv venv
```

This will create a virtual environment in a folder called `venv`.

- **Activate the virtual environment**: After the environment is created, you need to activate it.

  - On **Linux/macOS**:

    ```
    bash
    Copy code
    source venv/bin/activate
    ```

  - On **Windows**:

    ```
    bash
    Copy code
    venv\Scripts\activate
    ```

- **Verify the virtual environment is active**: Once activated, your terminal prompt should change, indicating that you're now working within the virtual environment. You can verify by running:

```
bash
Copy code
python --version
```

This should show the version of Python you installed in the virtual environment.

Now you're ready to install any Python packages in this isolated environment!

1. `pip3 install "uvicorn[standard]"`

This command installs Uvicorn, a lightning-fast ASGI (Asynchronous Server Gateway Interface) server, which is used to serve FastAPI applications.

- `uvicorn` is an ASGI web server implementation for Python.
- The `[standard]` part installs additional optional dependencies, such as support for WebSockets, HTTP/2, and other useful features for running production-ready applications.

Why do we need `uvicorn`?

FastAPI is an asynchronous web framework, and to handle asynchronous code properly, it needs an ASGI server like Uvicorn. Without it, you can't serve your FastAPI application in a performant and scalable way. Uvicorn allows FastAPI to handle large numbers of concurrent requests efficiently, making it ideal for modern, async-based applications.

2. Command: `uvicorn main:app --reload`

This command runs your FastAPI application using Uvicorn.

- `main` refers to the Python file that contains your FastAPI code. For example, if your code is in `main.py`, this refers to `main.py`.
- `app` refers to the FastAPI instance in your code. For example, if you have `app = FastAPI()` in `main.py`, it refers to this `app`.
- `--reload` enables auto-reloading during development. This means Uvicorn will automatically restart the server whenever you make changes to your code, making development easier and faster.

3. What is Pydantic, and why do we use it?

Pydantic is a Python library used for data validation and parsing. It's a core component of FastAPI for defining data models. Pydantic helps ensure that the data you send or receive via HTTP requests is valid, safe, and well-structured.

Why use Pydantic in FastAPI?

- **Data validation**: Pydantic validates incoming data based on the types defined in your models. For example, if your model expects an integer and the user sends a string, Pydantic will raise an error automatically.
- **Data serialization**: Pydantic can automatically convert Python objects (like your model classes) into formats like JSON, which is needed for API responses.
- **Type hints**: Pydantic uses Python type hints, making your code easier to understand and maintain.

4. FastAPI and Pydantic - Model Classes and Nested Models

In FastAPI, Pydantic models are used to define the structure of data that your API will accept and return.

Example of a Simple Pydantic Model:

```
python
Copy code
from pydantic import BaseModel

class Band(BaseModel):
    name: str
    genre: str
    has_albums: bool
```

In this example:

- `name`: a required string field.
- `genre`: a required string field.
- `has_albums`: a boolean that indicates if the band has albums.

When someone sends a request to your API, FastAPI uses Pydantic to validate that the incoming data conforms to this model.

Nested Models:

Pydantic also supports **nested models**, which are models inside other models. This is useful when your data is hierarchical.

Example of a Nested Model:

```
python
Copy code
from pydantic import BaseModel
from typing import List

class Album(BaseModel):
    title: str
    year: int

class Band(BaseModel):
    name: str
    genre: str
    has_albums: bool
    albums: List[Album]
```

Here, the `Band` model includes a list of `Album` objects. FastAPI will ensure that any incoming data is structured correctly, with albums containing valid `title` and `year` fields.

5. FastAPI and Pydantic - URL Query Parameters for Filtering

In FastAPI, URL query parameters can be used to filter or customize the data you receive from an API. Query parameters are defined in the URL, like in `http://127.0.0.1:8000/bands?genre=rock&has_albums=true`.

Example of Query Parameters:

```
python
Copy code
from fastapi import FastAPI
from typing import List, Optional

app = FastAPI()

bands_db = [
    {"name": "Nirvana", "genre": "rock", "has_albums": True},
    {"name": "The Beatles", "genre": "pop", "has_albums": True},
    {"name": "Linkin Park", "genre": "rock", "has_albums": True},
]

@app.get("/bands/")
def get_bands(genre: Optional[str] = None, has_albums: Optional[bool] =
None):
    results = bands_db
    if genre:
        results = [band for band in results if band["genre"] == genre]
    if has_albums is not None:
        results = [band for band in results if band["has_albums"] ==
has_albums]
    return results
```

In this example:

- `genre` and `has_albums` are optional query parameters.
- The API will filter the bands by genre and whether they have albums based on the query parameters passed in the URL.

URL Example:

```
bash
Copy code
http://127.0.0.1:8000/bands?genre=rock&has_albums=true
```

- This URL filters the list of bands to show only rock bands that have albums.

6. FastAPI - Request Body and POST Requests

FastAPI allows you to handle **POST requests** with request bodies, which are used to send data to the server. Pydantic models define the structure of the request body.

Example of Handling a POST Request:

```
python
Copy code
```

```
from fastapi import FastAPI
from pydantic import BaseModel

app = FastAPI()

class Band(BaseModel):
    name: str
    genre: str
    has_albums: bool

@app.post("/bands/")
def create_band(band: Band):
    return {"message": "Band created successfully!", "band": band}
```

Here:

- A POST request to `/bands/` expects a JSON object that matches the `Band` model (with fields `name`, `genre`, and `has_albums`).
- FastAPI will automatically validate the incoming request body and ensure it matches the Pydantic model.

To make a POST request, you might use a tool like `curl` or Postman:

```
bash
Copy code
curl -X POST "http://127.0.0.1:8000/bands/" -H "Content-Type:
application/json" -d '{"name": "Nirvana", "genre": "rock", "has_albums":
true}'
```

7. Pydantic Pre-Validators

Pydantic supports **pre-validators**, which are functions that can be used to manipulate or validate the input data before it is used. These are helpful for transforming or ensuring the integrity of incoming data.

Example of a Pre-validator:

```
python
Copy code
from pydantic import BaseModel, Field, validator

class Band(BaseModel):
    name: str
    genre: str
    has_albums: bool

    @validator('name')
    def name_must_not_be_empty(cls, value):
        if len(value.strip()) == 0:
            raise ValueError('Band name cannot be empty')
        return value
```

In this example:

- The `@validator` decorator ensures that the `name` field is not empty. If the name is an empty string, an error is raised.
- Pre-validators like this allow you to perform additional checks on the data before FastAPI processes it.

Summary:

- **`pip3 install "uvicorn[standard]"`**: Installs Uvicorn, a high-performance ASGI server for running FastAPI apps.
- **`uvicorn main:app --reload`**: Runs a FastAPI app with auto-reload for development.
- **Pydantic**: A library used for data validation and parsing, crucial in FastAPI for ensuring that the data structures used in requests and responses are valid and follow the expected schema.
- **Model Classes and Nested Models**: Pydantic models allow you to structure complex data with nested relationships and validate them.
- **URL Query Parameters**: FastAPI allows filtering or customizing API responses using query parameters.
- **Request Body and POST Requests**: FastAPI uses Pydantic models to define and validate request bodies for POST requests.
- **Pre-validators**: Functions in Pydantic that validate or transform data before it's processed further.

This overview should give you a detailed understanding of how FastAPI, Uvicorn, and Pydantic work together for building modern, high-performance web APIs.

You're encountering a deprecation warning because the Pydantic library has updated its API in **Pydantic v2**. In Pydantic v2, `@validator` has been replaced by `@field_validator`, and the syntax for field validation has changed slightly.

Here's how to migrate from Pydantic v1 style `@validator` to the new Pydantic v2 style `@field_validator`.

Old Pydantic v1 Style (`@validator`):

In Pydantic v1, you would use `@validator` to create custom validation for fields.

Example (Pydantic v1):

```
python
Copy code
from pydantic import BaseModel, validator

class Item(BaseModel):
    name: str
    price: float
```

```
    description: str | None = None

    @validator('name')
    def check_name_not_empty(cls, value):
        if not value.strip():
            raise ValueError("Name cannot be empty")
        return value
```

New Pydantic v2 Style (`@field_validator`):

In Pydantic v2, you should use `@field_validator`. The syntax has been simplified, and you now define which fields you are validating directly in the decorator's argument.

Migrated Example (Pydantic v2):

```
python
Copy code
from pydantic import BaseModel, Field, field_validator

class Item(BaseModel):
    name: str
    price: float
    description: str | None = None

    @field_validator('name')
    def check_name_not_empty(cls, value):
        if not value.strip():
            raise ValueError("Name cannot be empty")
        return value
```

Key Changes in Pydantic v2:

1. **`@validator` → `@field_validator`**:
   - You now use `@field_validator` to validate individual fields.
   - The field name(s) to be validated are passed directly to `@field_validator`.
2. **Return Values**:
   - Just like in Pydantic v1, you still need to return the validated value. If you don't return the value, it will default to `None`.
3. **Field Validators for Multiple Fields**:
   - In Pydantic v2, you can use the same `@field_validator` to validate multiple fields by passing a list of field names.

   Example (Validating multiple fields):

   ```
   python
   Copy code
   @field_validator('name', 'description')
   def check_fields_not_empty(cls, value):
       if not value.strip():
           raise ValueError("Field cannot be empty")
       return value
   ```

Full Example of Pydantic v2 with `@field_validator`:

```
python
Copy code
from pydantic import BaseModel, Field, field_validator

class Item(BaseModel):
    name: str
    price: float = Field(ge=0)
    description: str | None = None

    @field_validator('name')
    def check_name_not_empty(cls, value):
        if not value.strip():
            raise ValueError("Name cannot be empty")
        return value

# Example usage:
item = Item(name="Guitar", price=1500)
```

In this new version:

- `@field_validator` replaces `@validator`.
- The validation logic stays largely the same.
- It's also easy to validate multiple fields using one validator by passing multiple field names to `@field_validator`.

By making this migration, you'll avoid deprecation warnings and ensure your code is compatible with Pydantic v2.

Let me explain what **`value.strip()`**, **`cls`**, and **`value`** are in the context of a **Pydantic field validator**.

1. What does `value.strip()` do?

`strip()` is a **string method** in Python that removes any leading and trailing whitespace (spaces, tabs, newlines) from a string.

Example:

```
python
Copy code
text = "  Hello World  "
clean_text = text.strip()
print(clean_text)  # Output: "Hello World"
```

In the context of the validator, `value.strip()` is used to:

- **Remove any leading and trailing spaces** from the input string.
- Ensure that the string has actual content and is not just spaces.

For instance:

```
python
Copy code
value = "   "
if not value.strip():
    raise ValueError("Field cannot be empty")
```

- Here, `value.strip()` will turn `" "` (which is just spaces) into an empty string `""`, allowing the validator to raise a `ValueError` because the string is essentially "empty."

2. What is `cls`?

`cls` stands for **class**. In a Pydantic validator, `cls` refers to the class where the validator is being defined. It's a reference to the model class that Pydantic is validating.

Example:

```
python
Copy code
class Item(BaseModel):
    name: str
    price: float

    @field_validator('name')
    def check_name_not_empty(cls, value):  # 'cls' refers to the Item class
        if not value.strip():
            raise ValueError("Name cannot be empty")
        return value
```

- `cls` here is a reference to the `Item` class itself.
- While you don't always need to use `cls`, it allows access to the class-level properties and methods inside the validator.

3. What is `value`?

`value` is the **value of the field** that you are validating. It is the input provided by the user for that particular field.

In the context of a Pydantic validator:

- `value` is the data that has been passed to the model for a specific field (like `name` in this case).
- The validator checks whether the `value` provided by the user meets certain criteria, and either raises an error or returns the cleaned value.

Example:

```
python
Copy code
item = Item(name="   Guitar  ")
```

- In this example, `value` inside the validator would initially be `" Guitar "`.
- The validator can then perform actions on `value`, like using `value.strip()` to remove unnecessary spaces.

Summary:

- **`value.strip()`** removes leading and trailing whitespace from the string `value`.
- **`cls`** refers to the class (`Item`) in which the validator is defined.
- **`value`** is the actual input being validated (the value of the field `name` in the example).

In the example validator:

```
python
Copy code
@field_validator('name')
def check_name_not_empty(cls, value):
    if not value.strip():  # Clean the value and check if it’s not empty
        raise ValueError("Name cannot be empty")
    return value
```

- `value.strip()` removes any extra spaces from `value`.
- If the resulting string is empty, it raises an error.
- If it's valid, it returns the cleaned value (with leading/trailing spaces removed).

1. Project Overview:

- **FastAPI**: A high-performance web framework for building APIs. It allows defining routes (URL endpoints) that users can access.
- **SQLModel**: A library that combines SQLAlchemy and Pydantic. It helps define data models (schemas) that map to tables in a relational database, making it easy to work with databases.
- **Pydantic**: A data validation and parsing library. In FastAPI, it helps define the structure of the data (input/output) and ensures that the data sent to/from the API is valid.
- **SQLite**: A lightweight relational database used to store data.
- **Uvicorn**: An ASGI server that serves your FastAPI application.

---

2. `schemas.py` File: Defining Data Models

In this file, you're defining **data models** using **Pydantic** and **SQLModel**. These models define the structure of data that your API will send and receive.

Enumerations (Enums):

- **`GenreURLChoices`** and **`GenreChoices`** are enums. They define valid choices for a band's genre.

```
python
Copy code
class GenreURLChoices(Enum):
    ROCK = "rock"
    METAL = "metal"
    GRUNGE = "grunge"
    ELECTRONIC = "electronic"

class GenreChoices(str, Enum):
    rock = "Rock"
    metal = "Metal"
    grunge = "Grunge"
    electronic = "Electronic"
```

  - `GenreURLChoices` are lowercased and are used for URL query parameters.
  - `GenreChoices` are capitalized and used in the data model to validate genre.

Pydantic Model Example:

- **`Album`** **and** **`BandBase`** are **Pydantic models** that define the schema (structure) of data for your API.

```
python
Copy code
class Album(BaseModel):
    title: str
    release_date: date
```

  The `Album` model defines the structure of an album with a `title` (string) and `release_date` (date). FastAPI uses this structure to validate data when receiving or sending albums.

SQLModel Models:

- **`AlbumBase`** **and** **`BandBase`** are **SQLModel models** that map to database tables. These models inherit from SQLModel and help create and query tables in the SQLite database.

```
python
Copy code
class AlbumBase(SQLModel):
    title: str
    release_date: date
    band_id: int = Field(default=None, foreign_key="band.id")

class Album(AlbumBase, table=True):
    id: int = Field(default=None, primary_key=True)
```

```
band: "Band" = Relationship(back_populates="albums")
```

- `AlbumBase` defines the structure of an album, including a `band_id` that links to a specific band (foreign key relationship).
- `Album` is the model for the **albums table** in the database, where each album has an `id` (primary key).
- **Relationship**: `Album` has a relationship with the `Band` model. Each album belongs to a band, and this relationship is bidirectional (`band` references the parent band).

Validators:

- **Validators** in Pydantic allow you to customize how data is validated or transformed before it's processed.

  ```
  python
  Copy code
  @validator('genre', pre=True)
  def title_case_genre(cls, value):
      return value.title() # RoCK -> Rock
  ```

  This validator ensures that the genre string is always in title case, meaning it corrects inputs like "RoCK" to "Rock" automatically.

---

3. `models.py`: Database Models with SQLModel

In this file, you're defining models that map to database tables using **SQLModel**. SQLModel is built on top of SQLAlchemy and Pydantic, making it easy to work with relational databases and validate data.

Band and Album Models (SQLModel):

These models define your database schema. `Band` and `Album` correspond to tables in the database, and SQLModel helps create and interact with these tables.

```
python
Copy code
class Band(SQLModel, table=True):
    id: int = Field(default=None, primary_key=True)
    name: str
    genre: GenreChoices
    albums: list[Album] = Relationship(back_populates="band")
```

- **`Band` model**:
  - The `table=True` parameter means this model corresponds to a database table (`band` table).

- The `id` field is the primary key.
- It has a list of albums, creating a relationship with the `Album` model. This means a band can have multiple albums.

---

4. `db.py`: Database Setup and Session Management

This file is responsible for setting up the connection to the **SQLite** database and managing database sessions.

Database Initialization:

```
python
Copy code
DATABASE_URL = "sqlite:///db.sqlite"
engine = create_engine(DATABASE_URL, echo=True)
```

- `DATABASE_URL` specifies the location of the SQLite database file (`db.sqlite`).
- `create_engine()` creates a connection to the SQLite database.

Initializing the Database:

```
python
Copy code
def init_db():
    SQLModel.metadata.create_all(engine)
```

- This function creates all the tables in the database based on the models defined in the `models.py` file. The tables are created when the server starts.

Session Management:

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

- This function is a **context manager** that provides a session (connection) to the database.
- The `yield` keyword allows you to use the session in other parts of the application, making sure it's properly closed when done.

---

5. `main.py`: FastAPI Application

This file defines the FastAPI routes and handles the HTTP requests. The routes in this file interact with the database using the models and sessions defined earlier.

FastAPI Setup:

```
python
Copy code
app = FastAPI(lifespan=lifespan)
```

- The FastAPI instance (`app`) is created with a **lifespan** context manager. This ensures the database is initialized when the server starts.

Lifespan Context Manager:

```
python
Copy code
@asynccontextmanager
async def lifespan(app: FastAPI):
    init_db()
    yield
```

- This context manager initializes the database when the application starts.

GET Request for Bands:

```
python
Copy code
@app.get("/bands")
async def bands(genre: GenreURLChoices | None = None, q: Annotated[str |
None, Query(max_length=10)] = None, session: Session = Depends(get_session))
-> list[Band]:
    band_list = session.exec(select(Band)).all()
    if genre:
        band_list = [band for band in band_list if band.genre.value.lower()
== genre.value]
    if q:
        band_list = [band for band in band_list if q.lower() in
band.name.lower()]
    return band_list
```

- This route handles GET requests to `/bands`.
- You can filter the bands by **genre** or search by **name** (`q` query parameter).
- `session: Session = Depends(get_session)` uses the session created in `db.py` to query the database for bands.

GET Request for a Specific Band:

```
python
Copy code
@app.get("/bands/{band_id}")
async def band(band_id: Annotated[int, Path(title="The band ID")], session:
Session = Depends(get_session)) -> Band:
    band = session.get(Band, band_id)
    if band:
        return band
    raise HTTPException(status_code=404, detail="Band not found")
```

- This route handles GET requests for a specific band by ID.

- If the band exists, it returns the band's data. Otherwise, it raises a `404` error (Band not found).

POST Request to Create a Band:

```
python
Copy code
@app.post("/bands")
async def create_band(band_data: BandCreate, session: Session =
Depends(get_session)) -> Band:
    band = Band(name=band_data.name, genre=band_data.genre)
    session.add(band)

    if band_data.albums:
        for album in band_data.albums:
            album_obj = Album(title=album.title,
release_date=album.release_date, band=band)
            session.add(album_obj)

    session.commit()  # Save to the database
    session.refresh(band)  # Refresh to get the new band's ID
    return band
```

- This route handles POST requests to create a new band.
- It validates the incoming request data using the `BandCreate` model.
- The `session.add()` method adds the band and any albums to the database.
- `session.commit()` commits the changes to the database,
  and `session.refresh(band)` updates the band object with its new ID.

---

6. Final Thoughts

In summary:

- **FastAPI** is used to handle HTTP requests and define API routes.
- **SQLModel** is used to define database models and manage database interactions.
- **SQLite** is your database engine, and you're using SQLModel to map Python objects to SQL tables.
- **Pydantic** is used for data validation and ensuring the data conforms to the models.

This project involves setting up an API that allows users to create, retrieve, and filter bands and their albums, with data stored in a SQLite database.

Let's break down the different parts of the FastAPI code examples you provided. We'll go through the syntax step by step, explain what's happening in each part, and clarify concepts like sessions, query parameters, and path parameters.

First Example:

```
python
Copy code
@app.get("/bands")
async def bands(
    genre: GenreURLChoices | None = None,
    q: Annotated[str | None, Query(max_length=10)] = None,
    session: Session = Depends(get_session)
) -> list[Band]:
    band_list = session.exec(select(Band)).all()
    if genre:
        band_list = [band for band in band_list if band.genre.value.lower()
== genre.value]
    if q:
        band_list = [band for band in band_list if q.lower() in
band.name.lower()]
    return band_list
```

Explanation of Parts:

`@app.get("/bands")`

- **`@app.get()`**: This is a **route decorator** in FastAPI that registers an HTTP GET request handler for the `/bands` URL path.
    - When someone accesses the `/bands` endpoint using a **GET** request, this function (`bands()`) is executed.

Function Signature (`async def bands()`):

- **`async`**: FastAPI supports asynchronous functions, allowing it to handle many requests efficiently. This means that while waiting for one operation (like a database call), the server can handle other requests in the meantime.

Parameters:

The function takes several parameters. Let's go over each one:

1. **`genre: GenreURLChoices | None = None`**:
    - **`genre`** is an optional query parameter (denoted by `| None`), which allows the user to filter bands by genre.

- `GenreURLChoices` is likely an **enum** that defines valid genre choices (e.g., rock, metal, grunge, etc.).
- **`= None`**: If the user doesn't provide this query parameter, its default value is `None`, meaning no genre filtering will happen by default.

2. **`q: Annotated[str | None, Query(max_length=10)] = None`**:
   - **`q`** is another optional query parameter. In this case, it's used to search for bands by a keyword.
   - `Annotated[str | None, Query(max_length=10)]` does the following:
     - `str | None`: The `q` parameter can be a string or `None`.
     - `Query(max_length=10)`: It's a query parameter limited to a maximum length of 10 characters.
     - **`= None`**: If not provided, `q` defaults to `None`.
3. **`session: Session = Depends(get_session)`**:
   - **`session`** is the database session, which is used to interact with the database.
   - **`Depends(get_session)`**: This tells FastAPI to **inject** the database session into the function by calling `get_session()`. `Depends` is a FastAPI utility used for dependency injection.
   - `get_session` is likely a function that returns a database session connected to the SQLite or another database.

Return Type (`-> list[Band]`):

- **`-> list[Band]`**: This specifies the function's return type, indicating that it returns a list of `Band` objects.

Function Logic:

1. **`band_list = session.exec(select(Band)).all()`**:
   - **`session.exec()`**: This method executes a SQL query using SQLModel (or SQLAlchemy). In this case, it's selecting all bands from the `Band` table.
   - **`select(Band)`**: This is equivalent to a `SELECT * FROM Band` SQL query.
   - **`.all()`**: This retrieves all the results from the query and stores them in the `band_list`.
2. **Filtering by Genre**:

```
python
Copy code
if genre:
    band_list = [band for band in band_list if band.genre.value.lower()
== genre.value]
```

   - If a genre was provided as a query parameter, this filters the bands to include only those whose genre matches the provided value.
3. **Filtering by Keyword (`q`)**:

```
python
```

```
Copy code
if q:
    band_list = [band for band in band_list if q.lower() in
band.name.lower()]
```

- If a search query (`q`) is provided, this filters the `band_list` to include only those bands whose name contains the search term.

4. **Return the Filtered List**:

```
python
Copy code
return band_list
```

- The function returns the final filtered list of bands.

Second Example:

```
python
Copy code
@app.get("/bands/{band_id}")
async def band(band_id: Annotated[int, Path(title="The band ID")], session:
Session = Depends(get_session)) -> Band:
    band = session.get(Band, band_id)
    if band:
        return band
    raise HTTPException(status_code=404, detail="Band not found")
```

Explanation of Parts:

`@app.get("/bands/{band_id}")`

- This is another route decorator, but this time it defines a route with a **path parameter** (`band_id`).
- The `/{band_id}` part means this endpoint expects a specific band ID in the URL, like `/bands/5`.

Parameters:

1. **`band_id: Annotated[int, Path(title="The band ID")]`**:
   - **`band_id`**: This is a **path parameter** that represents the ID of a band.
   - **`Annotated[int, Path(title="The band ID")]`**:
     - `int`: The `band_id` must be an integer.
     - `Path()`: This declares `band_id` as a required path parameter, and you can attach metadata to it like `title="The band ID"`, which is used for documentation (in the OpenAPI schema).
2. **`session: Session = Depends(get_session)`**:

- Just like the first example, this injects a database session using the `Depends(get_session)` dependency.

Return Type (`-> Band`):

- This indicates that the function returns a single `Band` object.

Function Logic:

1. **`band = session.get(Band, band_id)`**:
   - **`session.get()`**: This method retrieves a record from the database by its primary key (in this case, `band_id`).
   - It's essentially equivalent to `SELECT * FROM Band WHERE id = band_id`.
   - If the band with the provided `band_id` exists, it is returned; otherwise, `None` is returned.
2. **Return the Band or Raise 404 Error**:
   - If the band exists (`if band:`), it's returned.
   - If the band does not exist, a **404 Not Found** error is raised using:

     ```
     python
     Copy code
     raise HTTPException(status_code=404, detail="Band not found")
     ```

Session Methods (session.add, session.get, session.commit, session.refresh)

FastAPI uses SQLModel (or SQLAlchemy) for database operations, and these session methods allow interaction with the database.

`session.add()`:

- **What it does**: Adds a new object (record) to the database session.
- **Example**:

  ```
  python
  Copy code
  new_band = Band(name="Nirvana", genre="Grunge")
  session.add(new_band)
  ```

  This tells the session to keep track of `new_band` as a new record to be inserted into the database.

`session.get()`:

- **What it does**: Fetches a record from the database by its primary key.
- **Example**:

```
python
Copy code
band = session.get(Band, 5)
```

This retrieves the `Band` with the ID `5`. If no band with that ID exists, it returns `None`.

`session.commit()`:

- **What it does**: Commits (saves) the current transaction to the database.
- **Example**:

```
python
Copy code
session.commit()
```

After you add or modify records, calling `session.commit()` saves those changes to the database. Without calling `commit()`, the changes won't be persisted.

`session.refresh()`:

- **What it does**: Updates the attributes of an object from the database after a transaction.
- **Example**:

```
python
Copy code
session.refresh(band)
```

After inserting a new record, `session.refresh()` can be used to update the object with the latest state from the database, like assigning the generated primary key (ID).

Summary:

- **Route Decorators** (`@app.get()`): These define HTTP endpoints in FastAPI. For example, `/bands` is a GET request to retrieve a list of bands.
- **Query Parameters**: Parameters like `genre` and `q` in the first example allow users to filter results by passing data in the URL (e.g., `?genre=rock`).
- **Path Parameters**: Parameters like `band_id` are extracted directly from the URL (e.g., `/bands/5`).
- **Session Methods**:
  - **`session.add()`**: Adds a new record to the database.
  - **`session.get()`**: Fetches a record by primary key.
  - **`session.commit()`**: Saves changes to the database.
  - **`session.refresh()`**: Updates the object with the latest data from the database.

This should help clarify how FastAPI, SQLModel, and database sessions work together to handle HTTP requests and interact with the database!

1. How Everything Connects (Using an Analogy)

Let’s use a **restaurant** analogy to explain how all the components in your FastAPI project connect.

- **FastAPI**: Think of FastAPI as the **waitstaff** at a restaurant. It's responsible for interacting with the customers (API users). It listens to requests like "I want to see the menu" (GET request) or "I want to place an order" (POST request). FastAPI handles these requests, processes the data, and sends a response back to the customer.
- **SQLModel**: This is the **kitchen** where all the orders are handled. SQLModel takes care of preparing the data (like creating new bands, storing albums) and ensures that everything follows the restaurant's rules (data structure). It also stores the data in the database (kitchen inventory).
- **SQLite**: This is the **pantry** or **inventory** of the restaurant. It holds all the ingredients (data) that the kitchen (SQLModel) uses to fulfill orders (requests). SQLite is lightweight and easy to manage, just like a small pantry.
- **Pydantic**: This is the **recipe book** the kitchen (SQLModel) follows. It defines exactly what ingredients (data) are allowed in each dish (model). For example, it ensures that every album has a title and a release date, just like how every recipe requires specific ingredients. If someone tries to send a wrong ingredient (invalid data), Pydantic rejects the order.
- **Uvicorn**: This is the **front door** of the restaurant. It's the entrance through which all the customers (requests) come in. Uvicorn handles the incoming traffic and passes it to the waitstaff (FastAPI) so they can process it.
- **Session/Database connection**: This is like the **kitchen staff's connection to the pantry** (SQLite). Every time an order is made, the kitchen staff (SQLModel) needs to "check the inventory" (session) and make sure they have the ingredients to prepare the dish. The session is how the kitchen communicates with the pantry (database).

---

2. Difference Between Old and New Code in `main.py`

Your **old code** and **new code** both achieve similar tasks, but they differ in how they handle data.

Old Code (Without SQLModel):

In the old code, you're manually managing the band data using a hardcoded `BANDS` list. This is like managing everything in a small notebook without a real database.

```
python
```

```
Copy code
BANDS = [
    {'id': 1, 'name': 'Metallica', 'genre': 'Metal'},
    # more bands...
]
```

- When a band is created, you manually append it to this list, and when querying, you filter through this list.
- **This works**, but it's inefficient as the list is stored in memory and doesn't persist when the application restarts.

Limitations of the old approach:

- **No persistence**: Data disappears when the app shuts down.
- **Manual filtering**: You filter the list in-memory for querying data, which can be slow and inefficient for larger datasets.
- **No relationships**: There's no way to easily connect related models like `Band` and `Album` in a clean and scalable way.

New Code (With SQLModel and SQLite):

In the new code, you're using **SQLModel** and a **SQLite** database, which means you're moving from a handwritten notebook to a **full-fledged inventory system**. The data is now stored in a relational database, and SQLModel handles creating, reading, updating, and deleting entries in that database.

```
python
Copy code
class Band(SQLModel, table=True):
    id: int = Field(default=None, primary_key=True)
    name: str
    genre: GenreChoices
    albums: list[Album] = Relationship(back_populates="band")
```

- You're using SQLModel to define **database tables** for `Band` and `Album`.
- Each `Band` can have multiple `Album`s, and these relationships are stored in the database.
- FastAPI interacts with the database via SQLModel and **sessions**. When you create a new band or album, it's persisted (saved) in the SQLite database.
- **Queries are faster and scalable** because you're using SQL queries via SQLModel, rather than manually filtering Python lists.

Advantages of the new approach:

- **Persistence**: Data is stored in a database and survives even after you shut down the app.
- **Scalability**: You can handle larger datasets efficiently.
- **Relationships**: You can easily model relationships between entities like `Band` and `Album`.

- **Better querying**: SQLModel allows you to run optimized SQL queries instead of manually filtering lists.

---

3. Database Migrations with Alembic

Now that you're using a **relational database**, you'll eventually need to **change the structure** of your database (e.g., adding new columns, changing relationships). This is where **Alembic** comes in.

What is Alembic?

Alembic is a **database migration tool** that helps manage changes to your database schema over time. It keeps track of the **version history** of your database schema, so whenever you need to add a new field, create a new table, or modify relationships, Alembic will handle applying those changes to your actual database.

What Does Alembic Do?

- **Creates Migration Scripts**: Alembic generates migration scripts that describe changes in your database. These scripts are like "instructions" for how to upgrade or downgrade your database.
- **Version Control for Schema**: Just like how Git tracks changes in your code, Alembic tracks changes in your database schema. Every time you make a change (like adding a column to the `Band` table), Alembic keeps a record of that change.
- **Applies Migrations**: When you run Alembic commands, it applies those migration scripts to your database, making the necessary changes to match your updated model definitions.

Why Do We Need Alembic?

Imagine you've already created the `Band` and `Album` tables in your database, and then later on, you decide to add a new field like `description` to the `Band` model. If you're manually managing the database, you'd have to write raw SQL queries to modify the table and update the records.

Alembic automates this process:

- It tracks your changes (e.g., adding a new field).
- It generates the SQL commands needed to apply those changes to the database.
- It ensures that your database schema stays in sync with your Python models.

Without Alembic, managing schema changes would become messy and error-prone, especially in a collaborative project with multiple developers or a large, complex database.

How to Use Alembic in Your Project:

1. **Install Alembic**:

   ```
   bash
   Copy code
   pip install alembic
   ```

2. **Initialize Alembic**: This command sets up Alembic in your project, creating a `migrations` folder and configuration files.

   ```
   bash
   Copy code
   alembic init migrations
   ```

3. **Create a Migration Script**: When you make changes to your models (e.g., adding a new column or table), create a migration script.

   ```
   bash
   Copy code
   alembic revision --autogenerate -m "Added description to Band"
   ```

   The `--autogenerate` flag inspects your models and generates the necessary migration scripts automatically.

4. **Apply the Migrations**: After generating the migration script, apply it to the database.

   ```
   bash
   Copy code
   alembic upgrade head
   ```

This command updates your database to the latest version, applying all pending migration scripts.

1. What is Database Migration?

Think of your **database** as a library and each **table** in the database as a section in that library (e.g., a table for bands, a table for albums). Over time, you might decide to change things in the library—add a new section (table), remove an old one, or change the way the books (data) are organized (modify a table's columns).

Database migration is the process of making changes to the database's structure (called the **schema**) without losing the data. In simpler terms, it's like **remodeling your library** while ensuring that the books are kept safe and organized properly.

2. The Problem Without Migration:

Without a migration tool like **Alembic**, making changes to a database can be complex and error-prone. Let's say you have a database for a music app, and you initially create a table called `bands` with the following columns:

- `id` (integer)
- `name` (string)
- `genre` (string)

Later, you realize you need to add more information to the `bands` table, such as the band's **description**.

Without Migration Tools:

You'd have to manually:

- Write an SQL query to **alter** the table (e.g., `ALTER TABLE bands ADD COLUMN description TEXT;`).
- Ensure that the database is updated in all environments (development, testing, production).
- Manually track the schema changes, which becomes messy if there are lots of changes.

With Alembic:

Alembic automatically generates the necessary SQL commands (like the `ALTER TABLE` query) and ensures that all environments are updated consistently with the changes, keeping track of the changes in versioned migration files. It makes schema changes easy and less risky.

3. Analogy: Remodeling a Library

Imagine your database is a **library**, and each **table** is a section of that library. Initially, your library only has a few sections, but over time, you may want to:

- Add new sections (e.g., a table for albums).
- Modify existing sections (e.g., add a new column to the bands table).
- Remove sections (e.g., drop an unused table).

With **Alembic**, it's like having a construction team that:

- **Plans and records** every remodel (migration script).
- **Carries out the remodel** in a safe way (applying the migration).
- **Keeps a log** of every change made, so you can undo changes if needed (rollback).

4. How Alembic Works:

a. Version Control for the Database:

Alembic gives your database a **version history** just like **Git** tracks changes in code. Every time you make a change (e.g., adding a new column or table), Alembic records that change as a **migration script**.

These migration scripts act like checkpoints in your database's version history:

- **Migration 1**: Create the `bands` table.
- **Migration 2**: Add the `albums` table.
- **Migration 3**: Add the `description` column to the `bands` table.

If something goes wrong, you can **roll back** to a previous version of the database, just like you would with Git commits.

b. Generating Migrations:

Alembic can **autogenerate** migrations by comparing your SQLModel models (or SQLAlchemy models) with the current database schema. This means that when you change your models, Alembic can automatically figure out how to update the database to match the new models.

Here's how it works:

1. **Initial Model**: Suppose you have the following model for `Band`:

   ```
   python
   Copy code
   class Band(SQLModel, table=True):
       id: int = Field(default=None, primary_key=True)
       name: str
       genre: GenreChoices
   ```

   When you run the initial migration, Alembic will create the `bands` table with these fields (`id`, `name`, `genre`).

2. **Modifying the Model**: Now you decide to add a new column, `description`, to the `Band` model:

   ```
   python
   Copy code
   class Band(SQLModel, table=True):
       id: int = Field(default=None, primary_key=True)
       name: str
       genre: GenreChoices
       description: str | None = None  # New column
   ```

   You'd run the Alembic command to autogenerate a migration script:

```
bash
Copy code
alembic revision --autogenerate -m "Added description to Band"
```

Alembic looks at your models and generates a script that contains SQL commands to update the `bands` table by adding a new column `description`.

3. **Applying the Migration**: Once the migration script is generated, you apply the changes to your database with:

```
bash
Copy code
alembic upgrade head
```

This runs the migration and updates the database schema to include the new `description` column in the `bands`table.

c. Migration Scripts:

The migration script generated by Alembic is a Python file that contains instructions on how to upgrade or downgrade the database.

Example of a migration script:

```
python
Copy code
"""Added description to Band table

Revision ID: abcd1234
Revises: prev1234
Create Date: 2023-10-21 12:34:56

"""
from alembic import op
import sqlalchemy as sa

# The unique identifier for this migration
revision = 'abcd1234'
# The migration it builds on top of
down_revision = 'prev1234'

def upgrade():
    # Add the description column to the bands table
    op.add_column('bands', sa.Column('description', sa.String(),
nullable=True))

def downgrade():
    # If we need to roll back, remove the description column
    op.drop_column('bands', 'description')
```

- **`upgrade()`**: This function contains the SQL command to **upgrade** the database schema by adding the `description`column.

- **`downgrade()`**: This function defines how to **roll back** the change, i.e., if you want to undo the migration, it will remove the `description` column.

5. Why Migrations Are Important:

Migrations are crucial for a few reasons:

- **Consistency**: When you work with a team or deploy an app to multiple environments (e.g., development, testing, production), migrations ensure that every environment's database is updated in the same way.
- **Version Control**: Just like with code, tracking changes in your database is important. Migrations provide a version history of your database, which helps when debugging or rolling back to a previous state.
- **Automation**: Manually updating a database can lead to mistakes. Alembic automates this process, reducing the risk of human error.
- **Scalability**: As your project grows, you'll frequently need to update the database schema (e.g., adding tables, modifying columns). Migrations make this process smooth and manageable.

---

6. Step-by-Step Example of Alembic in Action:

Let's go through a practical example of using Alembic for database migration:

a. Install Alembic:

```
bash
Copy code
pip install alembic
```

b. Initialize Alembic:

In the root of your project, run:

```
bash
Copy code
alembic init migrations
```

This creates a `migrations/` folder and a file called `alembic.ini`, which contains Alembic's configuration.

c. Set up the Alembic Config File:

In the `alembic.ini` file, configure the `sqlalchemy.url` to point to your database:

```
ini
Copy code
sqlalchemy.url = sqlite:///db.sqlite
```

d. Autogenerate a Migration:

After you've modified your SQLModel models, run the following command to autogenerate a migration:

```
bash
Copy code
alembic revision --autogenerate -m "Added description to Band"
```

Alembic will compare your models with the database schema and generate a migration script.

e. Apply the Migration:

Run the following command to apply the migration to the database:

```
bash
Copy code
alembic upgrade head
```

f. Rollback (Downgrade) the Migration:

If something went wrong, you can roll back the last migration with:

```
bash
Copy code
alembic downgrade -1
```

---

7. Final Thoughts:

- **Alembic** is your construction team that handles remodeling your database, ensuring every change is recorded and can be undone if needed.
- **Migrations** are essential to keep track of changes to the database schema as your application evolves.
- **Without migrations**, managing changes in large projects or across multiple environments would be chaotic and error-prone.
- **Alembic automates** the process of updating the database in a safe and controlled way, so you don't have to write complex SQL queries manually.

Database Interaction in FastAPI Apps with SQLModel is a key component when building applications that interact with relational databases. This interaction involves creating models, executing queries, handling database connections, and performing operations like creating, reading, updating, and deleting records (CRUD). Let's break down how SQLModel works in FastAPI, including the details of how it integrates with FastAPI to provide an efficient way to interact with databases.

1. What is SQLModel?

SQLModel is a library that combines the best features of **Pydantic** and **SQLAlchemy** to make database interaction in Python easier. It simplifies defining database models (like SQLAlchemy) while providing data validation and parsing (like Pydantic). SQLModel is fully compatible with FastAPI, making it a popular choice for building database-backed web APIs.

Key features of SQLModel:

- **SQLAlchemy ORM (Object-Relational Mapping)**: SQLModel uses SQLAlchemy under the hood, which allows us to interact with databases as if we were using Python objects.
- **Pydantic Validation**: SQLModel inherits Pydantic's data validation and parsing features, ensuring that data follows strict rules.
- **Type Hints**: SQLModel takes advantage of Python's type hints for both data validation and database schema generation.

2. SQLModel's Role in FastAPI

In a typical FastAPI app, SQLModel serves several key purposes:

- **Defining database tables** (models that map directly to the database structure).
- **Handling data validation** when working with API requests and responses.
- **Managing database connections** via sessions.
- **Running queries** to retrieve, insert, update, or delete data.

3. How SQLModel Integrates with FastAPI

To understand how SQLModel interacts with databases in FastAPI, let's go step by step:

a. Defining Database Models

The first step in interacting with a database is defining models that map to your database tables. SQLModel models are Python classes that inherit from `SQLModel`, and they define the columns in the table using fields.

Example of a `Band` and `Album` model:

```
python
Copy code
from sqlmodel import SQLModel, Field, Relationship
from typing import List
from datetime import date

class Album(SQLModel, table=True):
    id: int = Field(default=None, primary_key=True)
    title: str
    release_date: date
```

```
    band_id: int = Field(foreign_key="band.id")

    band: "Band" = Relationship(back_populates="albums")

class Band(SQLModel, table=True):
    id: int = Field(default=None, primary_key=True)
    name: str
    genre: str
    albums: List[Album] = Relationship(back_populates="band")
```

- **`SQLModel`**: This is the base class that allows the class to function as both a Pydantic model (for data validation) and a SQLAlchemy model (for database interaction).
- **`table=True`**: This parameter tells SQLModel that this model represents a table in the database.
- **Fields**: `Field()` is used to define the columns in the table. For example, `id` is the primary key, and `band_id` in `Album` is a foreign key that links the album to a specific band.
- **Relationships**: The `Relationship()` function defines how the models are related. Here, a `Band` has many `Album`s, and an `Album` belongs to a `Band`.

In this example, the `Band` table will have an `id`, `name`, and `genre`, while the `Album` table will have an `id`, `title`, `release_date`, and `band_id` (foreign key referencing the `Band` table).

b. Setting up the Database (Creating the Engine and Session)

Next, we need to set up the database engine and session to interact with the SQLite database (or any other database). SQLModel uses SQLAlchemy's engine to connect to the database.

```
python
Copy code
from sqlmodel import SQLModel, create_engine, Session

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

def init_db():
    SQLModel.metadata.create_all(engine)

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

- **Database URL**: This specifies where the database is located. In this case, it's an SQLite database (`db.sqlite`).
- **`create_engine()`**: This function creates a connection to the database using the provided URL. The `echo=True`option prints the SQL queries being run, which can be helpful for debugging.

- **`init_db()`**: This function creates all the tables in the database based on the models we've defined (`Band`, `Album`). It uses `SQLModel.metadata.create_all()` to look at all the models that have `table=True` and creates the corresponding tables in the database.
- **Session**: `get_session()` provides a way to access the database using a context manager. Sessions are used to perform queries and transactions in the database.

c. Creating a FastAPI Application

Once the models and database setup are ready, we can create a FastAPI application to interact with the database.

Here's an example of a FastAPI route that interacts with the `bands` and `albums` tables:

```
python
Copy code
from fastapi import FastAPI, Depends, HTTPException
from sqlmodel import Session, select
from models import Band, Album, get_session

app = FastAPI()

@app.post("/bands/", response_model=Band)
async def create_band(band: Band, session: Session = Depends(get_session)):
    session.add(band)
    session.commit()
    session.refresh(band)  # Refresh to get the new band's ID
    return band

@app.get("/bands/", response_model=list[Band])
async def get_bands(session: Session = Depends(get_session)):
    bands = session.exec(select(Band)).all()
    return bands

@app.get("/bands/{band_id}", response_model=Band)
async def get_band(band_id: int, session: Session = Depends(get_session)):
    band = session.get(Band, band_id)
    if not band:
        raise HTTPException(status_code=404, detail="Band not found")
    return band
```

- **`@post("/bands/")`**: This route handles the creation of a new band. It accepts a `Band` model in the request body, adds it to the database, and returns the newly created band (including the assigned ID).
- **`@get("/bands/")`**: This route retrieves all the bands from the database by executing a `SELECT` query on the `Band`table.
- **`@get("/bands/{band_id}")`**: This route retrieves a specific band by its ID. If the band doesn't exist, it raises a `404`error.

d. Executing Queries

SQLModel allows you to execute SQL queries in a Pythonic way. For example:

- **Inserting records**: `session.add(band)` adds a new record to the database.
- **Committing the transaction**: `session.commit()` saves the changes to the database.
- **Querying records**: `session.exec(select(Band)).all()` retrieves all records from the `Band` table.

SQLModel abstracts away most of the complexity of writing raw SQL queries, while still allowing you to write SQLAlchemy-like queries using Python.

e. Database Relationships in FastAPI

Since SQLModel is built on SQLAlchemy, it allows defining relationships between tables, like **one-to-many** (a band can have multiple albums) or **many-to-many** (if needed). Relationships are managed using `Relationship()` and foreign keys.

For example:

- In the `Band` model, we define the relationship with `albums`:

```
python
Copy code
albums: List[Album] = Relationship(back_populates="band")
```

- In the `Album` model, we define the foreign key and the reverse relationship:

```
python
Copy code
band_id: int = Field(foreign_key="band.id")
band: "Band" = Relationship(back_populates="albums")
```

This allows FastAPI to easily query related data. For example, you can fetch a `Band` along with its `albums`, thanks to this relationship.

---

4. CRUD Operations Using SQLModel in FastAPI

CRUD stands for **Create, Read, Update, Delete**, which are the four basic operations you'll commonly perform on database records. Let's look at how you can implement each of these using SQLModel and FastAPI.

Create (POST)

This operation inserts new data into the database.

```
python
Copy code
@app.post("/albums/")
async def create_album(album: Album, session: Session =
Depends(get_session)):
    session.add(album)
    session.commit()
    session.refresh(album)  # To get the auto-generated ID
    return album
```

- The `session.add(album)` inserts the new album into the `albums` table, and `session.commit()` saves the changes.

Read (GET)

This operation retrieves data from the database.

```
python
Copy code
@app.get("/albums/{album_id}", response_model=Album)
async def get_album(album_id: int, session: Session = Depends(get_session)):
    album = session.get(Album, album_id)
    if not album:
        raise HTTPException(status_code=404, detail="Album not found")
    return album
```

- The `session.get(Album, album_id)` fetches the album with the specified ID.

Update (PUT/PATCH)

This operation updates an existing record in the database.

```
python
Copy code
@app.put("/albums/{album_id}")
async def update_album(album_id: int, updated_album: Album, session: Session
= Depends(get_session)):
    album = session.get(Album, album_id)
    if not album:
        raise HTTPException(status_code=404, detail="Album not found")

    album.title = updated_album.title
    album.release_date = updated_album.release_date
    session.commit()
    return album
```

- This fetches the album by its ID, updates its fields, and commits the changes to the database.

Delete (DELETE)

This operation deletes data from the database.

```
python
Copy code
@app.delete("/albums/{album_id}")
async def delete_album(album_id: int, session: Session =
Depends(get_session)):
    album = session.get(Album, album_id)
    if not album:
        raise HTTPException(status_code=404, detail="Album not found")

    session.delete(album)
    session.commit()
    return {"message": "Album deleted successfully"}
```

- The `session.delete(album)` deletes the record, and `session.commit()` saves the change.

---

5. Database Transactions

SQLModel (and SQLAlchemy) supports **transactions**, which means multiple queries can be grouped together in a single atomic operation. Either all of them succeed, or none of them are applied.

For example, in this `create_band` route, both the `Band` and its associated `Albums` are added in a single transaction. If any error occurs during the process, the transaction can be rolled back.

```
python
Copy code
@app.post("/bands/")
async def create_band(band_data: BandCreate, session: Session =
Depends(get_session)):
    band = Band(name=band_data.name, genre=band_data.genre)
    session.add(band)

    if band_data.albums:
        for album in band_data.albums:
            album_obj = Album(title=album.title,
release_date=album.release_date, band=band)
            session.add(album_obj)

    session.commit()  # Commit the entire transaction
    session.refresh(band)  # Refresh to get the band's ID
    return band
```

Conclusion

Using **SQLModel** in a FastAPI app allows you to seamlessly interact with databases in a way that is both Pythonic and performant. It abstracts away much of the complexity of working with SQL queries, providing easy-to-use models, relationships, and sessions, all while retaining SQLAlchemy's power.

Here's a summary of the key concepts:

- **SQLModel models** represent database tables and allow you to validate and serialize data.
- **Sessions** are used to query and commit changes to the database.
- **Relationships** help you model how different tables are connected, making it easy to fetch related data.
- **CRUD operations** are the basic building blocks of database interaction.
- **Transactions** allow you to ensure multiple operations are applied consistently.

SQLModel combines the best of **SQLAlchemy** (for database interaction) and **Pydantic** (for data validation) to make building FastAPI apps with databases more efficient and easier to manage.

1. Annotated Type for Data Validation + Metadata

In FastAPI, **data validation** is a core part of handling incoming requests. FastAPI integrates **Pydantic** to automatically validate incoming data and ensure it meets the required schema. With **Annotated** types, you can add both **validation**rules and **metadata** for parameters, query strings, path parameters, and request bodies in a more flexible way.

What is Annotated?

The `Annotated` type in Python allows us to **attach metadata** to the type hints of function parameters. In FastAPI, `Annotated` can be used to:

- **Specify additional validation constraints** (like minimum or maximum values, lengths, etc.).
- Add metadata to parameters (such as descriptions or examples).
- Combine validations from different FastAPI components (e.g., `Query`, `Path`, `Body`).

Example of Annotated for Query Parameters:

```
python
Copy code
from fastapi import FastAPI, Query
from typing import Annotated

app = FastAPI()

@app.get("/items/")
async def read_items(
```

```
    q: Annotated[str | None, Query(max_length=50, min_length=3,
description="Search query string")]
):
    return {"query": q}
```

- `Annotated[str | None, Query(max_length=50, min_length=3)]`:
  - This declares that the `q` parameter is of type string (or None if it's optional).
  - `Query()` is used to define additional validation: here the query string must be between 3 and 50 characters.
  - **Metadata**: The description argument provides extra information about the query parameter, which can appear in the OpenAPI docs.

Why Use Annotated?

Using `Annotated`, you can consolidate multiple features (validation + metadata) into a single type declaration. It makes the code cleaner and more readable.

Example of `Annotated` for Path Parameters:

```
python
Copy code
from fastapi import Path

@app.get("/items/{item_id}")
async def read_item(item_id: Annotated[int, Path(title="The ID of the item",
ge=1)]):
    return {"item_id": item_id}
```

- **Path parameter `item_id`**: The value must be an integer (`int`) and it has to be greater than or equal to 1 (`ge=1`).
- The `title` metadata adds a description that will appear in the API documentation.

---

2. Request Body and POST Requests | Pydantic Pre-Validators

When handling **POST requests**, FastAPI uses **Pydantic models** to validate and serialize the data in the request body. Pydantic models are Python classes that define the structure of the data you expect and include automatic validation.

Request Body and POST Requests

When an API client sends data to your server (such as in a **POST** request), the data is typically sent in the **body** of the HTTP request. FastAPI uses Pydantic to define and validate this data.

Example of a POST Request with Pydantic Validation:

```
python
Copy code
from fastapi import FastAPI
```

```
from pydantic import BaseModel

app = FastAPI()

class Item(BaseModel):
    name: str
    price: float
    description: str | None = None

@app.post("/items/")
async def create_item(item: Item):
    return {"item": item}
```

- **`Item` model**: This Pydantic model defines the structure of an item (with `name`, `price`, and optional `description`).
- **Request body**: When a user sends a POST request to `/items/`, they must provide data that matches the `Item`model.
- FastAPI will automatically validate that the incoming data conforms to the expected structure (e.g., `name` must be a string, `price` must be a float).
- If the validation fails, FastAPI automatically returns an error response explaining the issue.

Pydantic Pre-Validators

Pre-validators in Pydantic allow you to customize how certain fields in a Pydantic model are validated **before** the actual validation process takes place. These are useful when you want to manipulate or clean data before it is stored or processed.

Example of a Pre-Validator:

```
python
Copy code
from pydantic import BaseModel, validator

class Item(BaseModel):
    name: str
    price: float
    description: str | None = None

    @validator('name')
    def check_name_not_empty(cls, value):
        if not value.strip():
            raise ValueError("Name cannot be empty")
        return value

@app.post("/items/")
async def create_item(item: Item):
    return {"item": item}
```

- **`@validator('name')`**: This pre-validator ensures that the `name` field is not an empty string (whitespace is stripped before validation). If the name is empty, it raises a validation error.

- Pre-validators allow you to enforce custom validation logic that goes beyond the default validation (like minimum length or type checking).

Advantages of Pre-Validators:

- **Data normalization**: You can modify incoming data (e.g., convert a string to lowercase) before it's validated.
- **Complex validation**: When simple field validation isn't enough, pre-validators allow for more complex logic.

---

3. FastAPI and Pydantic - URL Query Parameters for Filtering

Query parameters in URLs are commonly used for **filtering** data in APIs. FastAPI allows you to define query parameters using Pydantic models and provides the tools to easily filter and validate these parameters.

What are URL Query Parameters?

Query parameters are key-value pairs sent in the URL after the `?` character. For example:

```
bash
Copy code
http://127.0.0.1:8000/items?name=rock&has_albums=true
```

In this example:

- `name=rock`: This is a query parameter where the key is `name` and the value is `"rock"`.
- `has_albums=true`: This is another query parameter.

How to Define Query Parameters in FastAPI

You can use FastAPI's `Query()` to define query parameters and add validation constraints such as required parameters, default values, or length constraints.

Example of Filtering with Query Parameters:

```
python
Copy code
from fastapi import FastAPI, Query
from typing import List

app = FastAPI()

@app.get("/items/")
async def get_items(
    name: str | None = None,
    price_min: float | None = Query(default=None, ge=0),
```

```
    price_max: float | None = Query(default=None, le=1000)
):
    items = [{"name": "Rock", "price": 500}, {"name": "Jazz", "price": 200}]

    if name:
        items = [item for item in items if name.lower() in
item["name"].lower()]

    if price_min is not None:
        items = [item for item in items if item["price"] >= price_min]

    if price_max is not None:
        items = [item for item in items if item["price"] <= price_max]

    return items
```

- **Query parameters**:
  - `name`: An optional query parameter to filter items by name.
  - `price_min`: A query parameter to filter items with a price greater than or equal to `price_min`.
  - `price_max`: A query parameter to filter items with a price less than or equal to `price_max`.
- **Filtering Logic**:
  - The list of items is filtered based on the query parameters. For example, if `price_min` is provided, the items with a price greater than or equal to `price_min` will be included in the response.

Combining Query Parameters with Filtering:

You can use multiple query parameters to allow for complex filtering in your API.

Example of URL Query Filtering:

Let's say you have a route like this:

```
arduino
Copy code
http://127.0.0.1:8000/items?name=rock&price_min=100&price_max=500
```

- The `get_items()` function will:
  - **Filter by name**: It will only return items that have `"rock"` in their name (case-insensitive).
  - **Filter by price range**: It will only return items with prices between 100 and 500.

Advantages of Query Parameters:

- **Flexible filtering**: Users can specify multiple filters (e.g., price range, name, etc.) in a single request.

- **Default values**: You can set default values for optional query parameters.
- **Validation**: You can use Pydantic and FastAPI's built-in validation to ensure the query parameters are valid (e.g., price must be a positive number).

---

4. Summary

- **Annotated Type for Data Validation + Metadata**: FastAPI uses the `Annotated` type to attach validation rules and metadata to parameters, query strings, path parameters, and request bodies. It allows for a clean and flexible way to validate data while also adding useful documentation.
- **Request Body and POST Requests | Pydantic Pre-Validators**: FastAPI uses Pydantic models to define the structure of request bodies for POST requests. Pre-validators in Pydantic allow for custom validation logic before the data is processed, making it easier to clean or normalize the data.
- **URL Query Parameters for Filtering**: Query parameters are key-value pairs sent in the URL that allow for dynamic filtering. FastAPI makes it easy to define and validate query parameters using `Query()`, allowing you to build powerful APIs with flexible filtering options.

Each of these features provides powerful tools for creating fast, efficient, and well-validated APIs in FastAPI while keeping your code clean and maintainable.

```python
from typing import Annotated, get_type_hints, get_origin, get_args
from functools import wraps

#decorator
def check_value_range(func):
    @wraps(func)
    def wrapped(number):
        type_hints = get_type_hints(double, include_extras=True)
        hint = type_hints['number']
        if get_origin(hint) is Annotated:
            hint_type, *hint_args = get_args(hint)
            low, high = hint_args[0]
            print (low, high) # 0 100
            if not low <= number <= high:
                raise ValueError(f"Number must be between {low} and {high}")
        return func(number)
    return wrapped

@check_value_range
def double(number: Annotated[int, (0,100)]) -> int:
    return number * 2


result = double(4)
print(result) # 8
```

1. Imports and Why We Need Them

`from typing import Annotated, get_type_hints, get_origin, get_args`

- **`Annotated`**: This is a type from Python's `typing` module, which allows you to add metadata to type hints. In this case, it's used to define both the type (`int`) and a constraint on the value (in this case, a tuple `(0, 100)` representing a valid range).

  Example: `Annotated[int, (0, 100)]` means the input should be an integer, and its value must be between `0` and `100`.

- **`get_type_hints()`**: This function retrieves type hints from the function. In this example, it extracts the type hints for the `double` function, including the `Annotated` metadata (like the valid range).
- **`get_origin()`**: This function is used to get the original type for a given type hint. For instance, if the type hint is `Annotated`, `get_origin(hint)` will return `Annotated`. It helps in determining whether the type hint includes annotations or not.

- **`get_args()`**: This function retrieves the arguments inside a type hint. In this example, it gets the type (`int`) and the additional metadata (in this case, the tuple `(0, 100)` that represents the valid range).

```
from functools import wraps
```

- **`wraps`**: This is a utility decorator provided by Python to ensure that the decorated function retains its original name and metadata (such as its docstring). When you create a decorator, the function being decorated can lose some of its attributes (like its name or docstring). Using `@wraps`, the original function's attributes are preserved in the wrapper function.

---

2. The Decorator: `check_value_range`

A **decorator** is a function that modifies the behavior of another function. In this case, the `check_value_range` decorator is used to **validate** the input value of the function `double()` to ensure it falls within a specific range.

How does it work?

- The decorator **wraps** the original function `double()`. This means that when you call `double()`, you're actually calling the decorator first, which checks the value and then passes the value to `double()` if it's valid.
- Inside the decorator, the following happens:
  - **Retrieve type hints**: The `get_type_hints()` function is used to get the type hints for the `double` function. It looks at the function's parameter (here, `number`) and returns a dictionary containing the type hint (in this case, `Annotated[int, (0, 100)]`).
  - **Check if `Annotated` is used**: The `get_origin()` function checks if the hint is of type `Annotated`. If it is, that means there's additional metadata (like a value range) associated with the type.
  - **Extract the valid range**: `get_args()` is used to extract the actual type (`int`) and the range `(0, 100)` from the `Annotated` hint. This range is then used to check if the input value falls within the valid range.
  - **Raise an error if out of range**: If the input number is outside the valid range, a `ValueError` is raised.

Step-by-step Breakdown of the Decorator:

1. **`check_value_range` function**: This is the decorator function. It takes in the original function `double` and creates a wrapper function (`wrapped`).
2. **`@wraps(func)`**: This ensures that the wrapped function retains the original `double` function's name and metadata.

3. **`type_hints = get_type_hints(double, include_extras=True)`**:
    - This line extracts the type hints from the `double` function, including any extra metadata (like the `(0, 100)` range).
    - The result of this would be: `{'number': Annotated[int, (0, 100)]}`.
4. **`if get_origin(hint) is Annotated:`**:
    - This checks if the type hint for the `number` parameter is `Annotated`. In this case, it is, because we defined `number` as `Annotated[int, (0, 100)]`.
5. **`hint_type, *hint_args = get_args(hint)`**:
    - `get_args(hint)` extracts the type (`int`) and the metadata `(0, 100)`. The result is a tuple like `(int, (0, 100))`.
    - `hint_type` would be `int`, and `hint_args[0]` would be `(0, 100)`.
6. **`low, high = hint_args[0]`**:
    - This unpacks the tuple `(0, 100)` into two variables: `low = 0` and `high = 100`. These are the bounds for the valid range of the `number` parameter.
7. **Range validation**:

```
python
Copy code
if not low <= number <= high:
    raise ValueError(f"Number must be between {low} and {high}")
```

    - This checks whether the input number is within the range `(0, 100)`. If not, it raises a `ValueError` with a message like `"Number must be between 0 and 100"`.
8. **If valid, call the original function**:

```
python
Copy code
return func(number)
```

    - If the number is within the valid range, the original `double()` function is called with the number as an argument.

---

3. The Function: **`double()`**

```
python
Copy code
@check_value_range
def double(number: Annotated[int, (0,100)]) -> int:
    return number * 2
```

- **`@check_value_range`**: This applies the `check_value_range` decorator to the `double()` function. This means that every time `double()` is called, the decorator runs first to check if the number is valid.

- **`number: Annotated[int, (0, 100)]`**: The parameter `number` is an `int`, but with an additional annotation that says its value must be between 0 and 100. This is where the range metadata is defined.
- **Return value**: The function returns the input number multiplied by 2. For example, `double(4)` returns `8`.

---

4. Calling the Function: `double(4)`

```
python
Copy code
result = double(4)
print(result)  # 8
```

Here's what happens when you call `double(4)`:

1. **Decorator is invoked**: When you call `double(4)`, the `check_value_range` decorator is triggered before the actual `double()` function is executed.
2. **Value is validated**: The decorator checks if the value `4` is within the range `(0, 100)` by extracting the range from the `Annotated` type hint.
3. **Function executes**: Since `4` is within the valid range, the original `double()` function is called, which returns `4 * 2 = 8`.
4. **Result**: The result `8` is printed.

Summary of What's Happening:

1. **Annotated Types**: The `Annotated` type allows you to attach additional metadata to a type hint. In this case, `Annotated[int, (0, 100)]` indicates that the number must be an `int` between `0` and `100`.
2. **Decorator**: The `check_value_range` decorator is responsible for checking whether the input number falls within the valid range specified in the `Annotated` type hint. If the number is valid, it allows the original function to execute. If not, it raises a `ValueError`.
3. **Type Hint Inspection**: The `get_type_hints()`, `get_origin()`, and `get_args()` functions are used to inspect the type hint on the function parameter, check if it is `Annotated`, and then extract the valid range.
4. **Validation Logic**: The decorator ensures that the input value for the `double()` function is always within the allowed range (0 to 100), making it a reusable validation mechanism.

This code structure allows for **dynamic and flexible input validation** while keeping the actual function code (like `double()`) clean and focused on its main task (doubling the input).