Backend Engineering Notes: Dependency Injection, SQLAlchemy, and Connection Pooling

---

1️⃣ Dependency Injection in Python (Compared to Spring Boot)

Spring Boot Concept

- Spring manages objects called *beans*.
- Beans are injected via constructor injection.
- Example: Car → Audi / BMW (polymorphism).
- Spring IoC container manages lifecycle automatically.

How Python Does Dependency Injection

Python does NOT have a built-in IoC container like Spring. Dependency Injection is usually:

1. Manual (passing objects in constructor)
2. Framework-based (FastAPI Depends)
3. Using external DI libraries

In **Python**, things are different:

- Python does **not have a built-in DI container like Spring**.
- Dependency Injection is usually:
- Manual (passing objects via constructor)
- Framework-driven (like FastAPI's `Depends`)
- Or using external DI libraries (e.g. `dependency-injector`)

---

1. Dependency Injection in Pure Python (No Framework)

Python is already dynamic, so DI is simple.

Example: Like Your Car Example

```
from abc import ABC, abstractmethod

class Car(ABC):
    @abstractmethod
```

```python
    def drive(self):
        pass

class Audi(Car):
    def drive(self):
        return "Driving Audi"

class BMW(Car):
    def drive(self):
        return "Driving BMW"

class Driver:
    def __init__(self, car: Car):
        self.car = car

    def start(self):
        return self.car.drive()

# Inject dependency
driver = Driver(Audi())
print(driver.start())
```

Loose coupling is achieved because Driver depends on abstraction (Car), not concrete class.

☑️ What is happening?

- `Driver` depends on `Car`
- `Audi` and `BMW` implement `Car`
- You inject dependency manually
- This is loose coupling
- No container needed

In Python → **DI is just passing objects around**

2. How FastAPI Does Dependency Injection

Now let's understand your example.

FastAPI has a **built-in DI system** using:

```
Depends()
```

Your Database Setup Explained Properly

```python
from sqlalchemy import create_engine
from sqlalchemy.orm import sessionmaker, declarative_base
```

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

engine = create_engine(
    DATABASE_URL,
    connect_args={"check_same_thread": False}
)

SessionLocal = sessionmaker(
    autocommit=False,
    autoflush=False,
    bind=engine
)

Base = declarative_base()
```

What is happening here?

- `engine` → manages connection to database
- `SessionLocal` → factory to create DB sessions
- `Base` → base class for all models

Blog Model

```
from sqlalchemy import Column, Integer, String, Text

class Blog(Base):
    __tablename__ = "blogs"

    id = Column(Integer, primary_key=True, index=True)
    title = Column(String, index=True)
    content = Column(Text)
```

Then:

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

This creates the table.

3. The Important Part: FastAPI Dependency Injection

Now look at this:

```
from fastapi import Depends
from sqlalchemy.orm import Session
```

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

This is where DI happens.

☑️ What is `get_db()`?

This is a **dependency provider function**.

It:

1. Creates a DB session
2. Yields it
3. Automatically closes it after request

Now see how it gets injected:

```
from fastapi import FastAPI

app = FastAPI()

@app.get("/blogs")
def read_blogs(db: Session = Depends(get_db)):
    return db.query(Blog).all()
```

What Happens Internally?

When request comes:

1. FastAPI sees `Depends(get_db)`
2. It calls `get_db()`
3. Gets the yielded `db`
4. Injects it into `read_blogs`
5. After request finishes → `finally` block runs → `db.close()`

☑️ How is this Similar to Spring?

Spring Boot:

```
@Autowired
private BlogRepository repo;
```

FastAPI:

```
def read_blogs(db: Session = Depends(get_db)):
```

Spring:

- Container manages bean lifecycle

FastAPI:

- Dependency system manages function lifecycle

2️⃣ SQLAlchemy Core Concepts

What is Session?

Session represents a conversation with the database.

It:

- Manages transactions
- Tracks object state
- Sends SQL to DB
- Handles commit / rollback

Created using:

```
SessionLocal = sessionmaker(bind=engine)
db = SessionLocal()
```

Important:

- Engine is thread-safe
- Session is NOT thread-safe
- Create one session per request

What is declarative_base()?

```
Base = declarative_base()
```

It creates a base class for ORM models.

Any class inheriting from Base becomes a table.

Example:

```
class Blog(Base):
    __tablename__ = "blogs"

    id = Column(Integer, primary_key=True)
    title = Column(String)
    content = Column(Text)
```

Creating Tables

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

This generates SQL CREATE TABLE statements automatically.

3️⃣ Using Local Docker PostgreSQL

Run PostgreSQL in Docker

```
docker run --name my-postgres
  -e POSTGRES_USER=shiv
  -e POSTGRES_PASSWORD=password
  -e POSTGRES_DB=mydb
  -p 5432:5432
  -d postgres:15
```

SQLAlchemy Connection String

```
DATABASE_URL = "postgresql://shiv:password@localhost:5432/mydb"
```

Install driver:

```
pip install psycopg2-binary
```

4️⃣ Connection Pool & Hikari Concept

What is a Connection Pool?

Opening DB connections is expensive.

Instead of opening a new connection for every request:

- App opens fixed number of connections
- Reuses them
- Returns them to pool

This improves performance and prevents database overload.

Spring Boot (HikariCP)

- Default pool size: 10
- Very fast JDBC connection pool
- Managed by Spring

SQLAlchemy Pool

When you create engine:

```
engine = create_engine(DATABASE_URL)
```

It automatically creates a QueuePool.

Default:

- pool_size = 5
- max_overflow = 10

Configure like Hikari:

```
engine = create_engine(
    DATABASE_URL,
    pool_size=10,
```

```
    max_overflow=0
)
```

5️⃣ Where is psycopg2 Used?

You don't import psycopg2 manually.

When you write:

```
engine = create_engine("postgresql://...")
```

SQLAlchemy automatically loads the PostgreSQL driver (psycopg2).

It acts as the actual communicator between Python and PostgreSQL.

Architecture:

Session → Engine → Connection Pool → psycopg2 → PostgreSQL

6️⃣ What Does Thread-Safe Mean?

Thread-safe means: Multiple threads can use the same object safely without corrupting data.

Example of NOT Thread-Safe

```
counter = 0

def increment():
    global counter
    counter += 1
```

Two threads may overwrite each other's updates (race condition).

Thread-Safe Version

```
import threading

counter = 0
lock = threading.Lock()

def increment():
```

```
    global counter
    with lock:
        counter += 1
```

Lock ensures only one thread modifies counter at a time.

What Does Thread-Safe Mean?

Thread-safe means:

> Multiple threads can use the same resource at the same time without causing data corruption, crashes, or unpredictable behavior.

🛐 First: What Is a Thread?

A thread is like a worker inside your program.

In a web server:

- 100 users hit your API
- Server may handle them using multiple threads
- These threads run at the same time

So you might have:

```
Thread A → handling user 1
Thread B → handling user 2
Thread C → handling user 3
```

Now imagine they all use the same object.

❌ Example of NOT Thread-Safe Code

Let's say we have a shared counter:

```
counter = 0

def increment():
    global counter
    counter += 1
```

Now imagine 2 threads run this at the same time.

Internally `counter += 1` is actually:

```
1. Read counter
2. Add 1
3. Write back
```

If both threads read at the same time:

```
Thread A reads 0
Thread B reads 0
Thread A writes 1
Thread B writes 1
```

Final value = 1
Expected value = 2

💥 This is called a **race condition**.

This code is **not thread-safe**.

✅ Making It Thread-Safe

We use a lock:

```
import threading

counter = 0
lock = threading.Lock()

def increment():
    global counter
    with lock:
        counter += 1
```

Now only one thread can enter the critical section at a time.

No race condition.

Now it is **thread-safe**.

🔸 Real World Analogy

Imagine:

- 2 people withdrawing money from same bank account
- Account balance = ₹1000

If no locking:

```
Person A reads 1000
Person B reads 1000
A withdraws 500 → writes 500
B withdraws 500 → writes 500
```

Bank just lost ₹500.

That system is NOT thread-safe.

If bank locks account during withdrawal:

One finishes first, then next.

That is thread-safe.

Now Connect This to SQLAlchemy

When we say:

> Engine is thread-safe

It means:

- Multiple threads can request DB connections
- Pool internally locks access
- No two threads corrupt pool state

But:

> Session is NOT thread-safe

If two threads use same session:

- They may overwrite transaction state
- Rollback each other's work
- Cause unpredictable bugs

So:

- Engine → safe to share
- Session → one per thread/request

Simple Definition

Thread-safe = safe for concurrent use.

Not thread-safe = unsafe when multiple threads access simultaneously.

🔕 Easy Visual

Not Thread-Safe

```
Thread A ─┐
          ├─ Access same variable → conflict
Thread B ─┘
```

Thread-Safe

```
Thread A ─ Lock ─ Update ─ Unlock
Thread B waits
```

🅱️ Final One-Line Understanding

Thread-safe means:

> "Even if 100 threads use it at the same time, it will behave correctly."

If you want, next I can explain:

- Difference between thread-safe vs async-safe
- Race condition vs deadlock
- How FastAPI handles concurrency internally

That's where backend mastery begins 😄

In SQLAlchemy

- Engine is thread-safe
- Connection Pool is thread-safe
- Session is NOT thread-safe

Correct usage in FastAPI:

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

Each request gets its own session.

What Is a Connection Pool?

First understand the real problem.

When your app talks to a database:

1. It opens a TCP connection
2. Authenticates (username/password)
3. Allocates memory
4. Prepares session
5. Executes query
6. Closes connection

Opening a DB connection is **expensive** (slow + CPU heavy).

If 1000 users hit your API and you open 1000 fresh connections:

💥 Your database will crash.

☑️ Solution: Connection Pool

A connection pool is:

> A fixed set of already-open database connections that are reused.

Instead of:

```
Request → Open connection → Query → Close
```

We do:

```
App starts
↓
Open 10 connections
↓
Store them in a pool
↓
Requests borrow connection
↓
Return connection back to pool
```

🔸 Real-Life Analogy

Think of it like a cab stand.

- 10 taxis waiting
- Customer comes → takes taxi
- Ride finishes → taxi returns
- Next customer uses same taxi

Instead of building a new taxi every time.

What Is Hikari?

☸️ HikariCP

HikariCP is:

> A very fast, lightweight JDBC connection pool used in Java applications.

It is the **default connection pool in Spring Boot**.

Why Is It Famous?

Before Hikari:

- C3P0
- Apache DBCP

They were slower.

HikariCP was built to be:

- Extremely fast
- Low memory
- Low latency
- Production stable

It became default in Spring Boot because:

😰 It's the fastest JDBC pool.

How It Works in Spring Boot

When your Spring Boot app starts:

1. Hikari opens 10 DB connections (default)
2. Keeps them alive
3. When request comes → Spring borrows connection
4. After transaction → connection returned

You configure it like:

```
spring:
  datasource:
    hikari:
      maximum-pool-size: 10
```

What Happens Without Connection Pool?

Imagine:

- 500 users hit API
- Each opens new DB connection
- PostgreSQL default max connections = 100

After 100 connections:

👈 New requests fail

Error:

```
FATAL: too many connections
```

Connection pooling prevents this.

Important Pool Terms

TermMeaning	
pool size	Number of permanent connections
max overflow	Extra temporary connections
idle timeout	Close unused connections
pool timeout	How long to wait if pool full

Connection Pool in Python

SQLAlchemy also has pooling.

When you do:

```
create_engine()
```

It automatically creates:

```
QueuePool
```

Works same way as Hikari.

Visual Comparison

Without Pool

```
Request 1 → Open → Close
Request 2 → Open → Close
Request 3 → Open → Close
```

Very expensive.

With Pool

```
Startup:
  10 connections opened

Request 1 → borrow → return
Request 2 → borrow → return
Request 3 → borrow → return
```

Fast + safe.

Why DBs Limit Connections

Each DB connection:

- Consumes RAM
- Uses CPU
- Allocates backend process (Postgres uses 1 process per connection)

If you allow 10,000 connections:

💥 Server dies.

So pooling controls it.

Backend Architecture Level Thinking

In real production:

```
Users → Load Balancer → App Servers → Connection Pool → PostgreSQL
```

If you have:

- 3 app servers
- Each pool size = 10

Total DB connections = 30

You must configure this carefully.

Simple Definition

Connection Pool

A managed collection of reusable database connections.

HikariCP

A high-performance JDBC connection pool used in Spring Boot.

🅱️ Final Mental Model

Database connections are expensive.

Connection pools:

- Open a few
- Reuse them
- Protect database
- Improve performance

HikariCP is just one implementation (Java world).

SQLAlchemy QueuePool is Python world equivalent.

Final Mental Model

Spring: Container manages beans and connection pool (Hikari).

Python:

- Dependency Injection is explicit.
- SQLAlchemy Engine manages connection pool.
- psycopg2 talks to PostgreSQL.
- One session per request.

These concepts form the foundation of backend architecture.