Azure API Training <> Revanture

15th October 2025, Wednesday

Dependency Injection System

What is Dependency Injection?

Writing clean, reusable, and testable code without repetition.

This is very useful when you need to:

- Have shared logic (the same code logic again and again).
- Share database connections.
- Enforce security, authentication, role requirements, etc.

The Problem Without Dependency Injection

Imagine this situation:

- You have several endpoints in your app.
- Each endpoint needs to verify a token, get app settings, or load configuration file.
- You copy-paste that logic in every route

Example: Without DI

```
from fastapi import FastAPI, Header, HTTPException

app = FastAPI()

@app.get("/profile")

def profile(authorization: str = Header(None)):

    if authorization != "Bearer secret-token":

        return {"message": "Invalid token"}

    return {"message": "User profile data"}
```

Problem?

Problem	Description
Repetition	Token verification logic repeated everywhere
Tight coupling	Business logic mixed with auth logic
Hard to test	You can’t easily replace auth logic in tests
Difficult to maintain	Changing auth rule → update every endpoint
Messy code	Shared logic spread across routes

Solution? Enter Dependency Injection

Dependency Injection lets you define reusable components (like authentication, settings, or validation) and inject them automatically into endpoints.

FastAPI’s Dependency System

- FastAPI uses the Depends() function for DI.
- You define a function (dependency).
- FastAPI automatically calls it and injects its return value into your endpoint.

Using DI with FastAPI (1/2)

```
from fastapi import FastAPI, Depends, HTTPException, Header
app = FastAPI()
def get_current_user(token: str = Header( None)):
    if token != "secret123":
        raise HTTPException(status_code= 401, detail="Invalid or missing
token")
    return {
        "id": 1,
        "name": "Avinash Seth",
        "email": "avinash@demo.com"
    }
```

Using DI with FastAPI (2/2)

```
@app.get("/profile")
def profile(user = Depends(get_current_user)):
    return {"message": f"Welcome, {user['name']}!"}


@app.get("/settings")
def settings(user = Depends(get_current_user)):
    return {"message": f"{user['name']}'s settings page"}
```

Example: Inject Configuration Settings

```
from fastapi import FastAPI, Depends
from pydantic import BaseModel

app = FastAPI()

class Settings(BaseModel):
    app_name: str = "My FastAPI App"
    version: str = "1.0.0"

def get_settings():
    return Settings()

@app.get("/info")
def info(settings: Settings = Depends(get_settings)):
    return {"app": settings.app_name, "version": settings.version}
```

Dependencies Can Depend on Other Dependencies

```python
from fastapi import Depends, FastAPI

app = FastAPI()

def get_settings():
    return {"debug": False} # True for Debug Mode, False for Production Mode

def get_logger(settings=Depends(get_settings)):
    if settings["debug"]:
        return "Logger in DEBUG mode"
    return "Logger in PROD mode"

@app.get("/logs")
def logs(logger=Depends(get_logger)):
    return {"message": logger}
```

Conclusion

- Use Depends() to inject reusable logic
- Keeps routes clean, testable, and maintainable
- Define dependencies once, reuse everywhere
- Easily override them for testing
- Works for auth, settings, logging, and more

Background Tasks and async operations

Background Task

Background tasks are pieces of work your application schedules to run after it returns a response to the client (fire-and-forget from the client's perspective). They let the API return quickly while heavier, non-critical work runs separately in the background.

Big corporate example

Company: Global e-commerce platform (millions of users).

User action: Customer places an order.

What must happen:

- Return an immediate confirmation (low latency) to the user.
- Charge the card (call payment gateway).
- Send order confirmation email & SMS.
- Update analytics and inventory.
- Generate a long PDF invoice and store it in object storage.

Background Task Example

```
from fastapi import FastAPI, BackgroundTasks
from datetime import datetime, timedelta, timezone
app = FastAPI()
IST = timezone(timedelta(hours=5, minutes=30))
def get_ist_timestamp() -> str:
    now = datetime.now(IST)
    return now.strftime("%Y-%m-%d %H:%M:%S")
```

Background Task Example

```
def write_audit_log(message: str):
    timestamp = get_ist_timestamp()
    with open("audit.log", "a") as f:
        f.write(f"[{timestamp}] {message}\n")
```

Background Task Example

```
@app.post("/order")
def place_order(customer_email: str, background_tasks:
BackgroundTasks):
    background_tasks.add_task(write_audit_log, f"Order placed for
{customer_email}")
    return {
        "status": "order received",
        "timestamp": get_ist_timestamp()
    }
```

Async Operations

Async operations means using async/await and non-blocking I/O so the server can handle many requests concurrently without being blocked by slow operations (network calls, I/O, etc.).

Async tasks run in the same event loop (unless moved to a thread/process), so they’re very efficient for I/O-bound work.

Async Operations (Example code)

```
from fastapi import FastAPI, BackgroundTasks
from datetime import datetime, timedelta, timezone
import asyncio
app = FastAPI()
IST = timezone(timedelta(hours=5, minutes=30))

def get_ist_timestamp() -> str:
    now = datetime.now(IST)
    return now.strftime("%Y-%m-%d %H:%M:%S")
```

Async Operations (Example code)

```
async def send_email_async(to: str, subject: str):
    await asyncio.sleep(5)  # Simulate delay
    timestamp = get_ist_timestamp()
    with open("email.log", "a") as f:
        f.write(f"[{timestamp}] Sent email to {to}: {subject}\n")
```

Async Operations (Example code)

```
@app.post("/order")
def place_order(customer_email: str, background_tasks:
BackgroundTasks):
    background_tasks.add_task(send_email_async, customer_email,
"Thanks for your order!")
    return {
        "status": "order received",
        "timestamp": get_ist_timestamp()
    }
```

Best practices

- **Idempotency**: design background jobs to be idempotent; use idempotency keys for operations like payments to avoid duplicates.
- **Retries & backoff**: for critical jobs, use a task queue with retry/backoff logic.
- **Monitoring & visibility**: log job start/finish/failure;
- **Error handling**: catch exceptions inside tasks and record them; with Celery, configure retries and alerting.
- **Resource limits**: avoid running unbounded tasks in the web process — prefer external workers for heavy load.
- **Security & credentials**: never store secrets in code — load from environment or secret manager.
- **Graceful shutdown**: ensure background tasks are allowed to finish or are handled properly on shutdown; for durable work prefer queues so workers can finish.
- **Testing**: override dependencies in tests and provide fake queues or synchronous behavior for deterministic testing.

File upload and downloads

How to upload a file

- pip install python-multipart

How to upload a file

```
from fastapi import FastAPI, File, UploadFile
from datetime import datetime, timedelta, timezone
from pathlib import Path
import shutil
app = FastAPI()
UPLOAD_DIR = Path("uploads")
UPLOAD_DIR.mkdir(exist_ok=True)
IST = timezone(timedelta(hours=5, minutes=30))
```

How to upload a file

```
def get_ist_timestamp():
    now = datetime.now(IST)
    return now.strftime("%Y-%m-%d %H:%M:%S")
```

How to upload a file

```
@app.post("/upload")
def upload_file(file: UploadFile = File(...)):
    timestamp = get_ist_timestamp()
    file_name = f"{timestamp.replace(' ', '_').replace(':', '-')}_{file.filename}"
    file_path = UPLOAD_DIR / file_name

    with open(file_path, "wb") as buffer:
        shutil.copyfileobj(file.file, buffer)

    return {
        "message": "File uploaded successfully!",
        "file_name": file_name,
        "timestamp_ist": timestamp
    }
```

How to download a file

```
from fastapi.responses import FileResponse
from fastapi import HTTPException

@app.get("/download/{filename}")
def download_file(filename: str):
    safe_name = Path(filename).name  # Prevent path traversal
    file_path = UPLOAD_DIR / safe_name

    if not file_path.exists():
        raise HTTPException(status_code=404, detail="File not found")

    # Return file as downloadable response
    return FileResponse(
        path=file_path,
        filename=safe_name,
        media_type="application/octet-stream"
    )
```

Custom Middleware

What is a Middleware?

What is Middleware?

A middleware is a function or component that sits between the client request and your application.

It lets you process or modify the request before it reaches your endpoints

and/or the response before it goes back to the client.

How we can use Middleware?

- Check authentication tokens or IPs for every request
- Record request method, path, and response time
- Measure execution time or track slow requests
- Add or modify response headers (like CORS or caching)
- Validate or clean data before it hits your routes

Building our Custom Middleware

```
from fastapi import FastAPI, Request
import time
from datetime import datetime, timedelta, timezone
app = FastAPI()
IST = timezone(timedelta(hours=5, minutes=30))
```

Building our Custom Middleware

```
@app.middleware("http")
async def log_requests(request: Request, call_next):
    start_time = time.time()
    start_ts = datetime.now(IST).strftime("%Y-%m-%d %H:%M:%S")

    print(f"[{start_ts}] Incoming request: {request.method} {request.url.path}")

    response = await call_next(request)

    duration = time.time() - start_time
    print(f"Completed in {duration:.2f}s\n")

    return response
```

Building our Custom Middleware

```
@app.get("/")
def home():
    return {"message": "Welcome to FastAPI Middleware Demo!"}


@app.get("/hello")
def hello():
    return {"greeting": "Hello from /hello route"}
```

Async Database Operations

What Are Database Operations?

Database operations are the actions your app performs on a database:

- **Read (SELECT)** – Fetching data
- **Write (INSERT)** – Adding new records
- **Update (UPDATE)** – Modifying existing records
- **Delete (DELETE)** – Removing data

In most apps, these operations happen many times per second — every time a user logs in, views a profile, or checks out.

Synchronous (Sync) Database Calls

In a synchronous program, every operation runs one after another, the next line waits until the previous one is done.

```
def get_user_from_db(user_id):
    result = db.execute(f"SELECT * FROM users WHERE id={user_id}")
    return result

def main():
    user = get_user_from_db(1)
    print("User loaded!")
```

Asynchronous (Async) Database Operations

Async database operations don’t block your app while waiting for the database.

They use non-blocking I/O — meaning your app can handle other requests in the meantime.

Example: How to perform Async Insert

pip install asyncmy greenlet

Download and make changes the the following code (source code)

Run the code

fastapi dev filename.py

Task

Try Async delete, select or update operation

Connection Pooling and Session Management

What is Connection Pooling?

Connection pooling is a technique used by applications to reuse existing database connections instead of creating a new one every time a request is made.

It’s a “pool” (collection) of open, ready-to-use database connections that your application keeps in memory, so when a new request comes in, it can borrow one instead of reconnecting to the database from scratch.

The Problem Without Connection Pooling

Normally, every time your app needs data, it:

- Connects to the database (handshake + auth)
- Executes the query
- Closes the connection

If 100 users send requests, your app opens and closes 100 connections!

That’s slow, CPU-heavy, and can crash your DB with too many concurrent connections.

How Connection Pooling Solves It

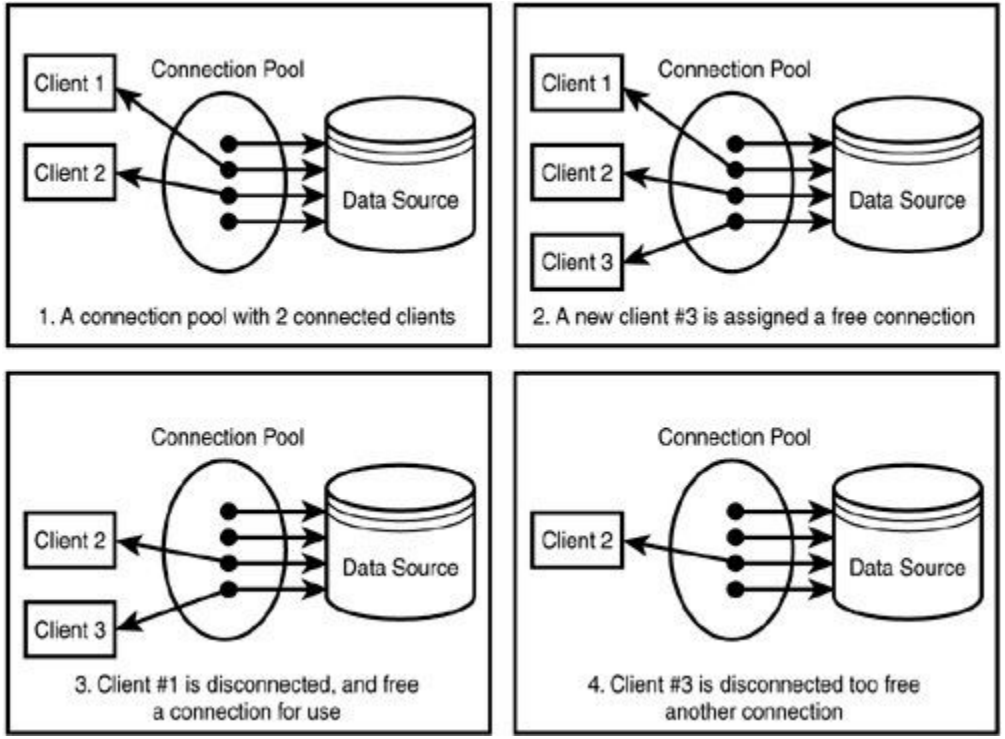
With connection pooling:

- A small number of connections are opened once and kept alive.
- When a request needs to run a query, it borrows a connection from the pool.
- After the query is done, the connection is returned to the pool for reuse.

So the app doesn’t constantly “connect–disconnect–connect”.

It’s more like: “Use a ready connection, then give it back.”

How Connection Pooling Solves It



What is a session in DB Session Management?

A session is a unit-of-work / transactional context that:

- holds a DB connection from the pool while you work,
- tracks ORM objects you create/read/modify,
- lets you commit() or rollback() changes,
- and releases the connection back to the pool when closed.

Correct session lifecycle management prevents connection leaks, ensures transactions are atomic, and avoids concurrency issues.

Common pitfalls & how to avoid them

- Leaked sessions: always close in finally or use context managers (with / async with).
- Reusing session across threads: don’t. Create per-request session.
- Long-running transactions: avoid; keep transactions as short as possible to reduce locks.
- Session inside dependency but used in background task: background job should open its own session.

Example code

How to implement connection pool & session management using fastapi (source code)

Project

Upload File to Azure Blob Storage