Azure API Training <> Revanture

16th October 2025, Thursday

Unit Testing with Pytest

What is Unit Testing?

Unit testing is a software testing technique that focuses on testing individual units of code, usually functions or methods, in isolation.

By testing these individual units of code across your applications, you can ensure that each part of your code works as expected, making it easier to identify and fix bugs, refactor code, and add new features without introducing regressions.

Why use pytest?

Pytest offers several advantages over other testing frameworks in Python

Simplified syntax: The syntax for tests is concise and easy-to-understand, making it easier and faster to write them.

Automatic test discovery: Test functions are automatically discovered without the need for explicit registration.

Rich plugin ecosystem: A large number of plugins can extend pytest functionalities and connect it with other tools and services, such as Coverage.py, Django or Elasticsearch.

Getting started with Pytest

Install the following dependency

```
pip install pytest httpx
```

Create a simple **FastAPI** app (main.py file name)

```python
from fastapi import FastAPI
app = FastAPI()
@app.get("/")
def read_root():
    return {"message": "Hello, FastAPI!"}

@app.get("/greet/{name}")
def greet(name: str):
    return {"message": f"Hello, {name}!"}
```

Getting started with Pytest

Create another file test_main.py file

```
from fastapi.testclient import TestClient
from main import app
client = TestClient(app)

def test_read_root():
    response = client.get("/")
    assert response.status_code == 200
    assert response.json() == {"message": "Hello, FastAPI!"}

def test_greet_user():
    response = client.get("/greet/avinash")
    assert response.status_code == 200
    data = response.json()
    assert data["message"] == "Hello, avinash!"
```

Getting started with Pytest

Run the following command

```
pytest -v
```

Expected output

```
test_main.py::test_read_root PASSED                    [ 50%]
test_main.py::test_greet_user PASSED                   [100%]
```

Example with Query Parameters and POST

main.py

```
from fastapi import FastAPI

app = FastAPI()

@app.get("/add")
def add_numbers(a: int, b: int):
    return {"result": a + b}

@app.post("/users")
def create_user(user: dict):
    return {"status": "created", "user": user}
```

Example with Query Parameters and POST

test_main.py

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

client = TestClient(app)

def test_add_numbers():
    response = client.get("/add?a=5&b=10")
    assert response.status_code == 200
    assert response.json() == {"result": 15}

def test_create_user():
    response = client.post("/users", json={"name": "Avinash", "email": "avinash@example.com"})
    assert response.status_code == 200
    data = response.json()
    assert data["status"] == "created"
    assert data["user"]["name"] == "Avinash"
```

Example with Query Parameters and POST

Run the following command

```
pytest -v
```

Expected output

```
test_main.py::test_add_numbers PASSED                    [ 50%]
test_main.py::test_create_user PASSED                    [100%]
```

Using TestClient for FastAPI

What is TestClient?

TestClient (from fastapi.testclient) is a built-in testing helper that lets you:

- Simulate HTTP requests (GET, POST, PUT, etc.)
- Test routes without actually starting a server (uvicorn not needed)
- Get back real Response objects with .status_code, .json(), etc.
- Integrate perfectly with pytest
- It internally uses requests, so it behaves just like a real HTTP call.

How It Works

- You import your FastAPI app into the test file.
- You create a TestClient(app) instance.
- You call client.get() / client.post() like normal API requests.
- FastAPI internally executes the route logic and returns a Response object.
- You assert against the HTTP status code, headers, or response JSON.

Types of Test We can do with TestClient

Test Type	Description	Example
Unit Tests (API-level)	Test individual endpoints and responses	GET /users returns status 200
Integration Tests	Test endpoints + database or dependencies	POST /users actually inserts into test DB
Functional Tests	Test full user workflows (multi-step flows)	Register → Login → Access protected route
Error Handling Tests	Verify correct error codes and messages	Missing fields → 422 Unprocessable Entity
Auth / Permission Tests	Test protected routes via dependency overrides	Unauthorized → 401, Authorized → 200
Validation Tests	Test Pydantic model & request body validation	Invalid JSON → 422

Types of Test We can do with TestClient

Test Type	Description	Example
Header & Cookie Tests	Test request/response headers, cookies, tokens	Check JWT in headers, cookie sessions
File Upload Tests	Test multipart/form-data uploads	`/upload` endpoint for file validation
Query & Path Param Tests	Test endpoints with dynamic URLs or query params	`/users/{id}` or `/items?limit=5`
Dependency Override Tests	Replace DB/auth/etc. with fakes	Replace `get_db` or `get_current_user` in tests
Performance / Response Time Checks	Basic latency assertions (non-load)	Check response < 200ms
Mock External API Calls	Patch `requests` or `httpx` while using TestClient	Mock GitHub API or payment gateway
Regression / Contract Tests	Ensure response schema didn't change unexpectedly	Compare response keys to expected schema
Pagination / Filtering Tests	Test query params logic with multiple results	`/products?page=2&limit=5`

Postman for Manual Testing

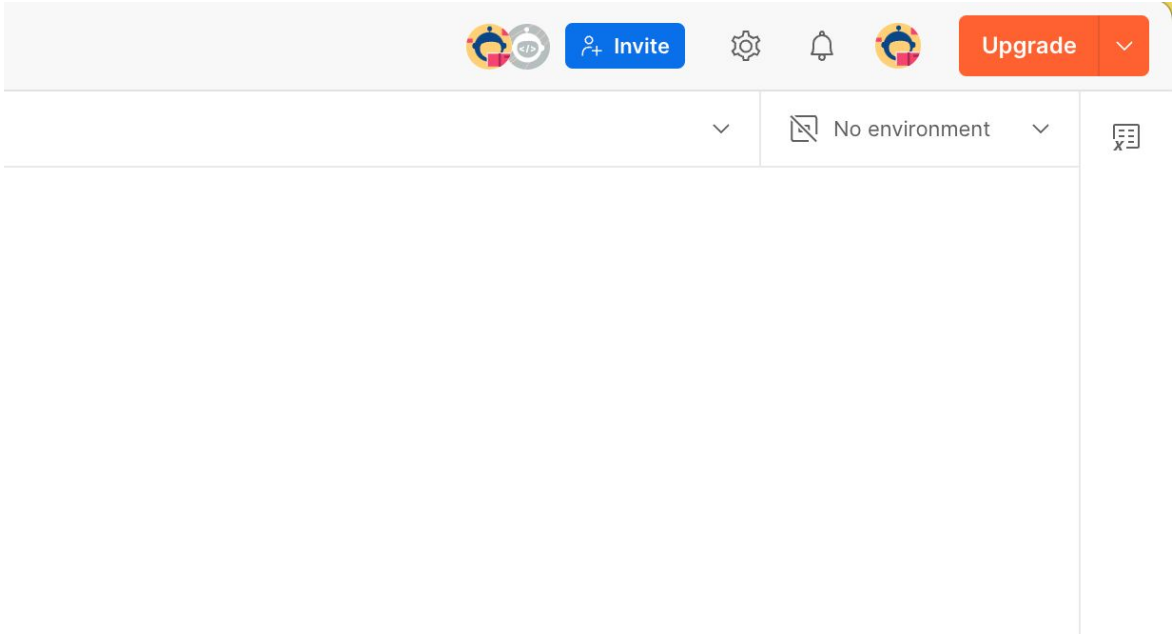
What is Postman?

Postman is an API development and testing tool that allows developers to send HTTP requests, inspect responses, automate tests, and organize APIs into collections.

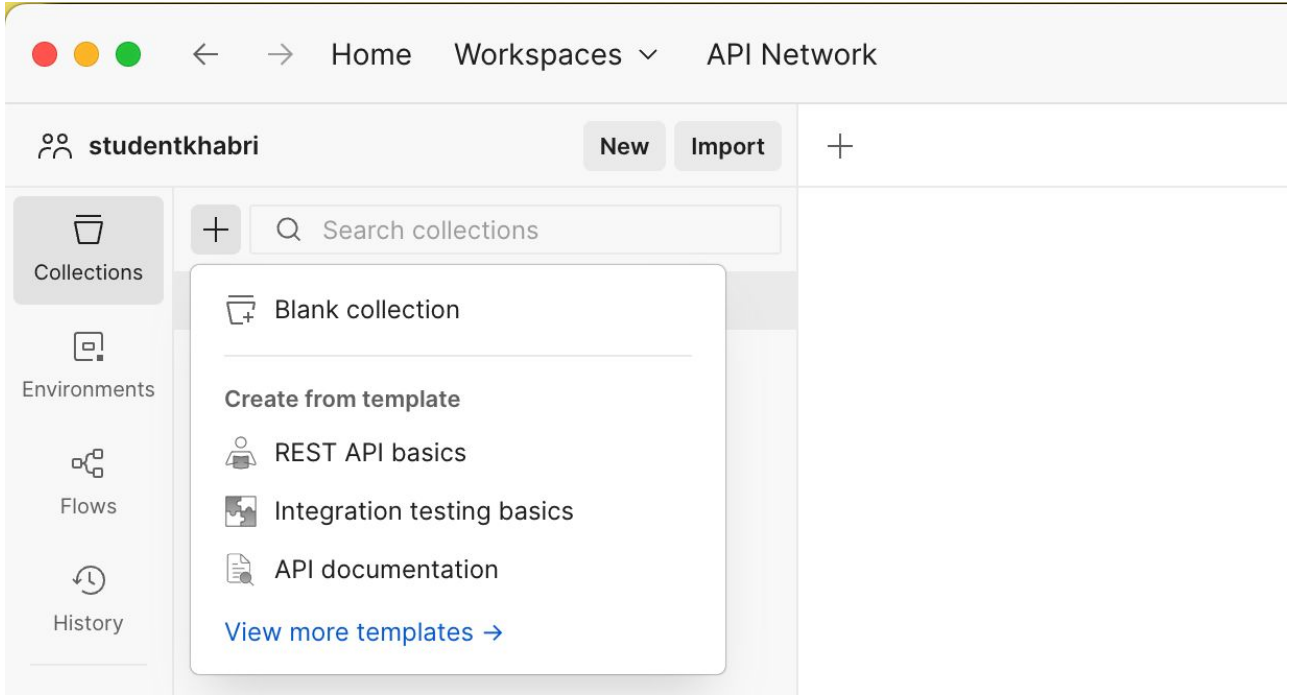
Postman helps

- Test APIs without writing code.
- Save and reuse requests (Collections).
- Validate responses (status code, headers, body).
- Automate test suites with JavaScript-based scripts.
- Share APIs easily within teams.
- Manage environment variables securely (e.g., tokens, base URLs).

Environments & Variables



Collections



Testing in Postman

Postman can help you test your FastAPI applications.

Write a basic First API app

```
from fastapi import FastAPI


app = FastAPI()


@app.get("/hi/{name}")
def hello_name(name: str = "Avinash"):
    return {"message": f"Hi, {name}!"}
```

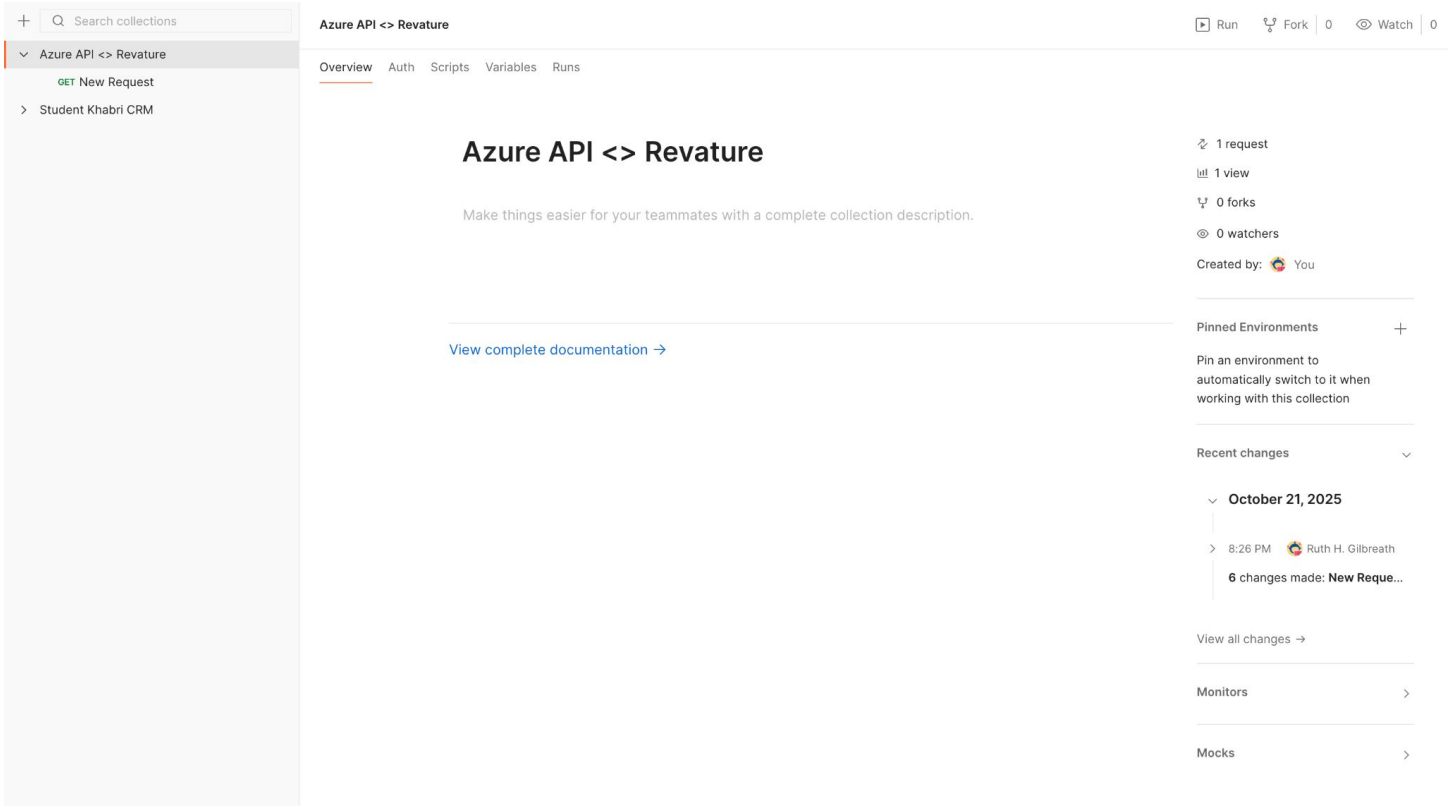
Testing in Postman

Write the following code in the scripts section of postman and then hit run!

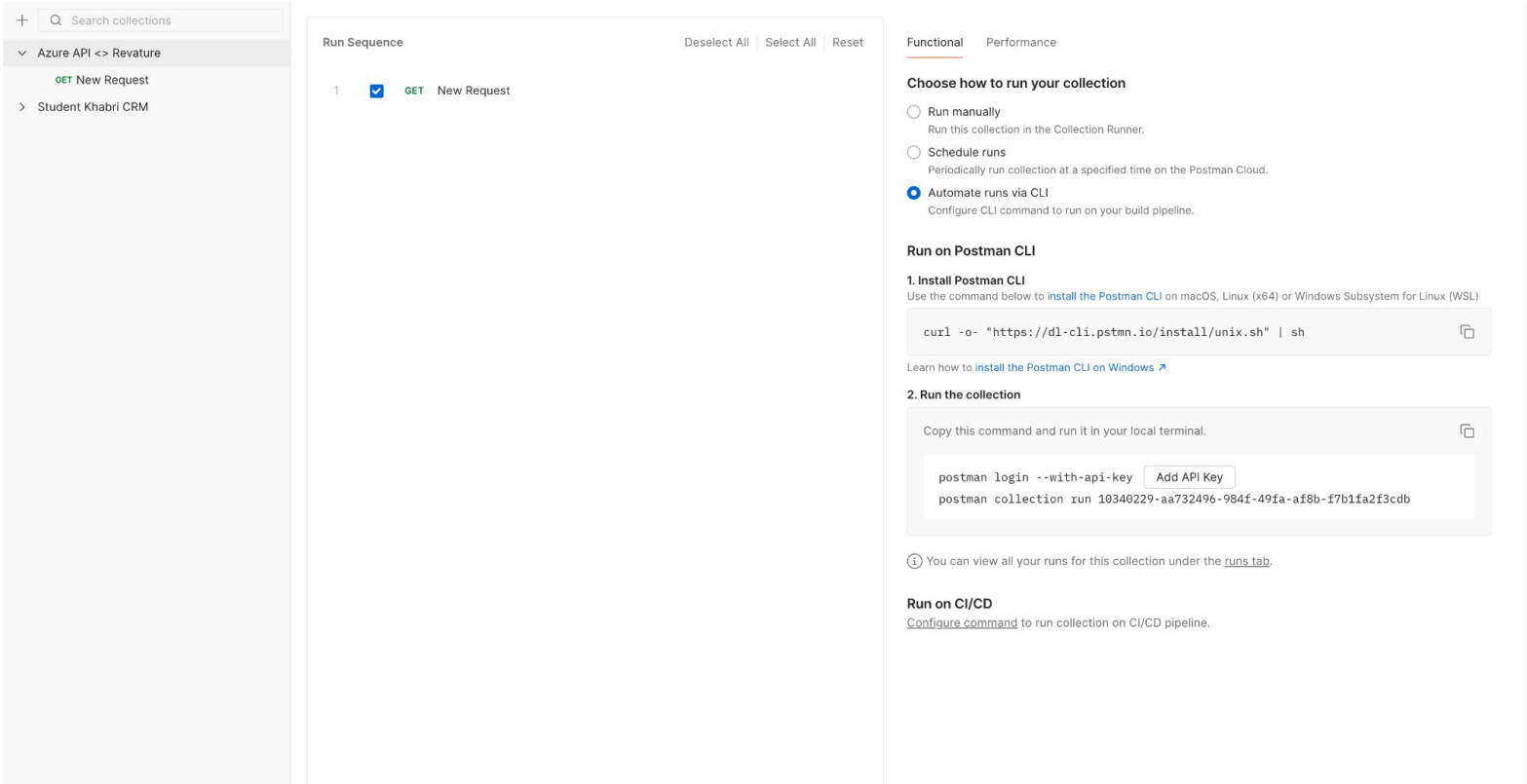
```
pm.test("Status code is 200", function () {
    pm.response.to.have.status(200);
});

pm.test("Response has username", function () {
    var jsonData = pm.response.json();
    pm.expect(jsonData.message).to.eql("Hi, avinash!");
});
```

Automating with Postman



Automating with Postman



Integration Testing for API

What is Integration Testing?

Integration testing is the process of testing how different parts of your system work together, not in isolation.

In the context of APIs, it means:

- You’re not just testing a single function (that’s unit testing)
- You’re testing the interaction between multiple layers:
  - API endpoints
  - Business logic
  - Database
  - External services (mocked or real)

The goal: ensure the entire request flow works correctly, from sending an HTTP request -> through your code -> to the database -> and back as a valid HTTP response.

How Integration Testing Works in APIs

- You send an **HTTP request** to your API (e.g., POST /users).
- The request hits your endpoint.
- It triggers business logic, interacts with the **database**.
- The API returns a **real response**.
- You verify:
  - Status code (200, 201, 400, etc.)
  - JSON structure
  - Data actually stored in DB

Integration Testing

- Create a file main.py and add the following source code
- Create a file test_main.py and add the following source code
- Run the following command
  - pytest -v
- Expected output
  - test_main.py::test_create_user PASSED [100%]

Test Driven Development

What is Test Driven Development?

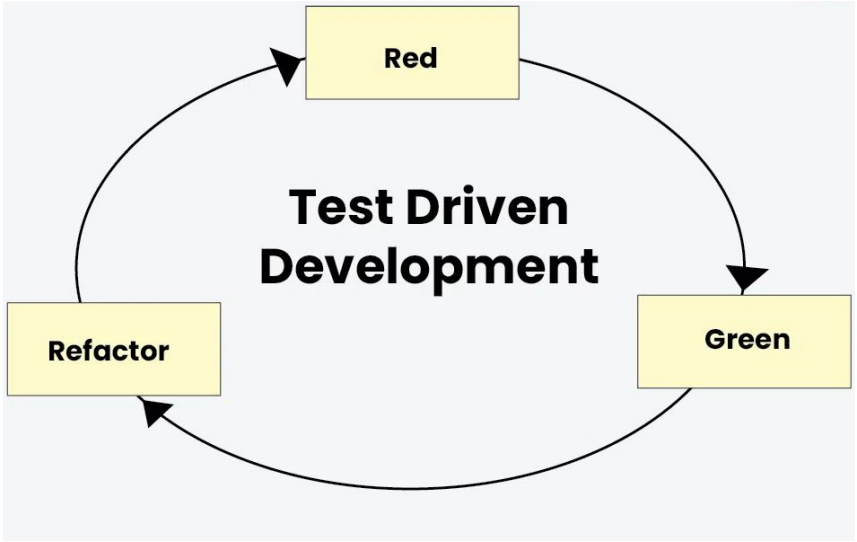
Test-Driven Development (TDD) is a Software development method in which you write Automation Tests before the actual development process starts, which is coding.

TDD = Think first, test second, code last.

It’s about writing tests that define what your code should do, and using them as a guide to write clean, reliable code.

The TDD Cycle: “Red → Green → Refactor”

The process of Test-Driven Development (TDD) follows a repetitive cycle called Red-Green-Refactor.



The TDD Cycle: “Red → Green → Refactor”

Run all the test cases and make sure that the new test case fails.

- Red - Create a test case and make it fail, Run the test cases
- Green - Make the test case pass by any means.
- Refactor - Change the code to remove duplicate/redundancy and Refactor code - This is done to remove duplication of code.

Approaches of Test Driven Development

Inside Out: In Test-Driven Development (TDD), you begin by testing the smallest units of code, such as individual functions or methods.

- Focuses on testing the smallest units first and building up from there.
- The architecture of the software emerges naturally as tests are written.
- Design and architecture are refined during the refactor stage, which can sometimes lead to significant changes.
- Easier to learn for beginners.
- Minimizes the use of mocks.
- Helps prevent over-engineering.

Approaches of Test Driven Development

Outside In: It will focuses on testing user behavior and interactions.

- Testing starts at the outermost level, such as the user interface, and works inward to the details.
- Relies heavily on mocks and stubs to simulate external dependencies.
- Harder to learn but ensures the code meets overall business needs.
- Design is considered during the red stage, aligning tests with business requirements from the start.

Advantages of Test Driven Development (TDD)

- Unit test provides constant feedback about the functions.
- Quality of design increases which further helps in proper maintenance.
- Test driven development act as a safety net against the bugs.
- TDD ensures that your application actually meets requirements defined for it.
- TDD have very short development lifecycle.

Disadvantages of Test Driven Development (TDD)

- Using TDD means writing extra code for tests cases , which can make the overall codebase larger and more Unstructured.
- Passing tests will make the developers think the code is safer only for assuming purpose.
- Keeping a lot of tests up-to-date can be difficult to maintain the information and its also time-consuming process.
- Writing and maintaining the tests can take a long time.
- TDD needs to be a proper testing environment in which it will make effort to set up and maintain the codes and data.

Mocking External Dependencies

What is mocking and why do it?

Mocking is basically replacing real external parts (APIs, DBs, caches, payment gateways, S3, SMTP) with lightweight fake implementations during tests.

Why:

- Make tests fast and deterministic (no real network or slow I/O).
- Isolate the unit under test (find problems faster).
- Simulate error conditions (timeouts, 5xx, etc.) easily.
- Avoid side effects (no real payments, emails, uploaded files).

Common approaches in Python / FastAPI

Dependency overrides (FastAPI): replace a dependency function with a test stub. Great for DB/session/auth services.

unittest.mock.patch / MagicMock: replace specific functions/classes (requests.get, a client method). Works for both sync & async (use AsyncMock for async).

HTTP stubs: responses (sync) or aioresponses (async aiohttp/httpx) to stub HTTP endpoints.

Provider-specific libs: moto to mock AWS (S3, SQS), pytest-localstack for more.

Fake servers (WireMock, httpbin, or local test server) — heavier but realistic.

Example: FastAPI dependency override

app.py

```
from fastapi import FastAPI, Depends, HTTPException

app = FastAPI()

def get_current_user(token: str = ""):
    if token != "secret":
        raise HTTPException(401)
    return {"username": "real_user"}

@app.get("/me")
def me(user = Depends(get_current_user)):
    return {"hello": user["username"]}
```

Example: FastAPI dependency override

test_app.py

```
from fastapi.testclient import TestClient
from app import app, get_current_user  # import the actual
dependency function

client = TestClient(app)

def fake_user():
    return {"username": "test_user"}
```

Example: FastAPI dependency override

test_app.py

```
def test_me_override():
    app.dependency_overrides[get_current_user] = fake_user
    try:
        resp = client.get("/me")
        assert resp.status_code == 200
        assert resp.json() == {"hello": "test_user"}
    finally:
        app.dependency_overrides.pop(get_current_user, None)
```

How to choose what to mock

- Mock network/third-party systems (payments, 3rd-party APIs) — tests should not depend on external services.
- For fast unit tests, mock everything external.
- For integration tests, prefer a lightweight real instance (test DB) to validate behavior.
- For end-to-end, use a real-ish environment (BUT keep E2E slower and fewer).

Best practices

- Prefer dependency injection in app code — it's much easier to override in tests.
- Use app.dependency_overrides or fixtures to inject fakes.
- Keep mocks explicit in tests (avoid over-mocking).
- Simulate failures too (timeouts, 500s) to test error handling.
- Clean up overrides after tests (teardown/fixture) to avoid cross-test leakage.
- Use AsyncMock for async patches.
- Use provider-specific mock libs for richer behavior.

Database Integration Patterns

What is Database Integration?

Database integration patterns are architectural approaches and best practices for connecting and synchronizing data between different databases, applications, and systems. These patterns help solve common challenges in enterprise software development where multiple systems need to share and exchange data.

Common Database Integration Patterns

Shared Database Pattern

Multiple applications directly access the same database. It's simple but creates tight coupling and can lead to schema conflicts and scalability issues.

Database Gateway/Data Access Layer

An abstraction layer sits between applications and the database, providing a clean API and isolating applications from database-specific details.

ETL (Extract, Transform, Load)

Data is periodically extracted from source systems, transformed into the target format, and loaded into the destination database. Common for data warehousing.

Common Database Integration Patterns

Change Data Capture (CDC)

Monitors and captures changes in source databases in real-time or near-real-time, then propagates those changes to other systems.

Event-Driven Integration

Applications publish events when data changes, and other systems subscribe to these events to update their own databases asynchronously.

API-Based Integration

Systems expose APIs (REST, GraphQL) that other applications call to read or write data, maintaining encapsulation and loose coupling.

Common Database Integration Patterns

Message Queue Pattern

Uses message brokers (like Kafka, RabbitMQ) to queue database operations and ensure reliable, asynchronous data transfer between systems.

Database Replication

Creates and maintains copies of databases across different locations for redundancy, load balancing, or geographic distribution.

Migration Strategies and Database Versioning

What is Database Migration?

A database migration means applying incremental changes to your database schema or structure — in a controlled, versioned, and repeatable way.

Why Database Migration?

Without Migration	With Migration
You manually change tables in production.	You apply small, versioned scripts step by step.
Risk of breaking code if schema mismatched.	Code and DB evolve together safely.
No history of changes.	Every schema change is recorded and reversible.

Example of Database Migration

```
/* create table users */


CREATE TABLE users (
 id INT PRIMARY KEY,
 username VARCHAR(100)
);
```

Migration #1: Add email column

```sql
-- add email column to the table
ALTER TABLE users ADD COLUMN email VARCHAR(255);
```

Migration #2: Add created_at timestamp

```
-- add created_at column to the table
ALTER TABLE users ADD COLUMN created_at TIMESTAMP DEFAULT
CURRENT_TIMESTAMP;
```

Migration #3: Add new table for posts

```
-- add new table posts
CREATE TABLE posts (
 id INT PRIMARY KEY,
 user_id INT,
 title VARCHAR(255),
 FOREIGN KEY (user_id) REFERENCES users(id)
);
```

Database Versioning with FastAPI

Database Versioning with FastAPI

Install following libraries in your project

pip install fastapi sqlalchemy alembic pymysql

Create a following folder structure

📁 project

- 📁 alembic
- 📄 alembic.ini
- </> main.py
- </> models.py

Database Versioning with FastAPI

Add following code in alembic.ini file (source code)

Run the following command

```
alembic init alembic
```

Create a new file

Migration 1 — 0001_create_users.py (source code)

Run the following command

alembic revision -m "create users table"

alembic upgrade 0001_create_users

Database Versioning with FastAPI

Create a new file

Migration 2 — 0002_add_email_to_users.py (source code)

Run the following command

alembic revision -m "add email to users"

alembic upgrade 0002_add_email_to_users

Database Versioning with FastAPI

Create a new file

Migration 3 — 0003_add_created_at_to_users.py (source code)

Run the following command

alembic revision -m "add created_at to users"

alembic upgrade 0003_add_created_at_to_users

Database Versioning with FastAPI

Create a new file

Migration 4 — 0004_create_posts.py (source code)

Run the following command

alembic revision -m "create posts table"

alembic upgrade 0004_create_posts