Using Pytest Fixtures for Layered Test Abstractions

In large and complex Python test suites, **Pytest fixtures** help abstract test setup and teardown logic in a reusable and composable way. This improves maintainability and readability while enforcing good separation of concerns. In a layered testing architecture (unit → integration → end-to-end), fixtures allow you to build from minimal mocks to fully wired systems progressively.

🔁 Layered Fixture Architecture

Let's illustrate the idea by using a layered test setup for a microservice-based API with database dependencies.

1. Unit Test Layer

Minimal setup with mocks or stubs.

```
@pytest.fixture
def mock_db():
    return MagicMock()  # or use pytest-mock's mocker fixture

@pytest.fixture
def service(mock_db):
    return MyService(db=mock_db)
```

Use in test:

```
def test_add_user(service):
    service.add_user("alice")
    service.db.insert.assert_called_once()
```

3. End-to-End (E2E) Layer

Simulates actual API calls, using real infrastructure (Docker Compose, test server).

```
@pytest.fixture(scope="session")
def start_test_server():
    proc = subprocess.Popen(["uvicorn", "main:app"])
    time.sleep(2)  # wait for startup
    yield
    proc.terminate()

@pytest.fixture
def api_client():
    return requests.Session()
```

Use in test:

```
def test_api_register_user(start_test_server, api_client):
    resp = api_client.post("http://localhost:8000/register", json={"na
    assert resp.status_code == 200
```

📌 Summary

Layered test abstractions using Pytest fixtures allow you to:

- Start with unit tests using mocks.
- Gradually increase fidelity with integration and E2E setups.
- Compose tests clearly and maintainably.

This aligns well with modern development where test speed, reliability, and clarity are all critical.