Testing Strategies: Making Sure Our Code Plays Nicely Together

Integration Testing with Testcontainers

A Practical Guide to Building Reliable Tests

What is Integration Testing? 🤔

It's testing how different parts (modules or services) of our application work together.

Unit Test

Does the car's engine start?

(Tests one small piece in isolation)

Integration Test

Do the engine, wheels, and steering work together to move the car?

(Tests how the pieces connect and interact)

The Challenge: Real-World Dependencies 🚧

Our code needs to talk to external services like a database (e.g., PostgreSQL) or a message queue (e.g., Kafka).

> **Problem**
>
> Using a shared development database is messy and unreliable. What if someone else changes the data?

> **Solution**
>
> We need a way to have a real, clean database just for our test.

Our Secret Weapon: Testcontainers! 🦸

Testcontainers lets us launch real services (like a database) in a temporary, disposable environment called a Docker container, just for our tests.

Realistic

We test against a real PostgreSQL database, not a fake one.

Isolated

Your test won't interfere with anyone else's.

Clean

The database is created, used for the test, and then automatically thrown away. No mess!

Our Goal: Write Integration Tests for the Booking Flow ✅

We will write a test that simulates a user creating a booking and verifies that the booking details are saved correctly in the database.

> **Test Flow**
>
> **Test Code ➡️ Booking Service API ➡️ Real Database**
>
> (Database from Testcontainers)

This gives us extremely high confidence that our most critical features, like making a booking, are working exactly as expected.

Key Takeaways

- Integration tests verify that different parts of your application work together correctly
- Shared environments lead to flaky, unreliable tests
- Testcontainers provides isolated, disposable dependencies for each test run
- Testing against real services (not mocks) catches more bugs before production
- Clean, isolated tests are predictable and maintainable

Check Your Understanding

Question 1

What is the main difference between a unit test and an integration test?

Question 2

Why is it a bad idea to use a shared development database for running our tests?

Question 3

What problem does Testcontainers solve for us?

Integration Testing using Testcontainers

Integration Testing Improvement

Python + FastAPI + Testcontainers

Key Achievement

Automated integration testing with real PostgreSQL containers on Windows using Docker Desktop

- **FastAPI** - Modern API framework
- **Testcontainers** - Real services testing
- **SQLAlchemy** - Database ORM
- **Pydantic** - Data validation

Context & Goals

Teaching Goal: Integration testing with Python ecosystem

Platform: Windows laptop with Docker Desktop

What We Built

A Task Manager API with two endpoints:

```
POST /tasks → Create task (title, done=false)
GET /tasks → List all tasks
```

Why Testcontainers?

The Strategy

Development: SQLite (simple, zero-install)

Testing: PostgreSQL via Testcontainers (production-like)

- SQLite is file-based and great for local development
- Testcontainers excels at starting real external dependencies
- Test with PostgreSQL to simulate realistic production database
- Keep classroom setup simple while maintaining test realism

Success Criteria

1. **Run app locally** with SQLite database
2. **Execute pytest** which automatically:
    - Starts throwaway PostgreSQL container
    - Applies schema automatically
    - Runs requests against FastAPI app
    - Tears down cleanly after tests
3. **Repeatable by students** on Windows with Docker Desktop

Project Structure

```
fastapi-testcontainers-demo/
├─ app/
│ ├─ __init__.py
│ ├─ config.py
│ ├─ database.py
│ ├─ models.py
│ ├─ schemas.py
│ ├─ crud.py
│ ├─ main.py
│ └─ logging_conf.py
├─ tests/
│ ├─ __init__.py
│ ├─ conftest.py
│ └─ test_tasks_integration.py
├─ requirements.txt
└─ README.md
```

Focused Scope

Keeping It Beginner-Friendly

✓ In Scope

- SQLAlchemy Core/ORM basics
- create_all for schema setup
- Simple API endpoints
- Container-based testing

✗ Out of Scope

- Complex ORM patterns
- Alembic migrations
- Authentication/Security
- Production deployment

Quick Start

Prerequisites: Docker Desktop must be running

```
# Install dependencies
pip install -r requirements.txt

# Run integration tests
pytest -q
```

Result: Automated tests spin up PostgreSQL, run tests, and clean up automatically

See README.md for quick start and run pytest -q to execute integration tests (Docker Desktop must be running).