

Module 8: Testing FastAPI Backends



Overview

- Why testing matters
- Testing Basics
- Types of testing
 - Unit tests
 - Integration tests
 - End-to-end (E2E) tests

Why Testing Matters

- Reliability of APIs
 - Testing ensures that your APIs consistently produce the correct output for a given input.
- Catching regressions (bugs) early
 - Run automated tests when code changes
- Confidence in deployments
 - When your tests are passing --> the new version of your API is not breaking existing functionality.

FastAPI Testing Tools

- `pytest`
- `httpx`
- FastAPI's `TestClient`

Project's Structure

```
module08_testing_apis/  
├─ main.py  
├─ tests/  
│   └─ test_main.py
```

main.py

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

Test Example

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

client = TestClient(app)

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

Run pytest

- From the project's root folder run:

```
python -m pytest
```

- Result:

```
===== test session starts =====
platform darwin -- Python 3.11.2, pytest-8.4.1, pluggy-1.6.0
rootdir: /Users/margitantal/PythonProjects/FASTAPI/module8_testing_apis
plugins: anyio-4.10.0
collected 1 item

tests/test_main.py . [100%]

===== 1 passed in 0.16s =====
```

Organizing tests

- Directory structure (`tests/`)
- Grouping by feature/module
- Naming conventions

Directory Structure

```
project/
├── app/
│   ├── main.py
│   ├── models.py
│   ├── routes/
│   │   ├── users.py
│   │   └── items.py
│   └── dependencies.py
└── tests/
    ├── __init__.py
    ├── conftest.py           # fixtures and setup
    ├── test_main.py         # basic smoke tests
    ├── unit/
    │   ├── test_models.py
    │   └── test_dependencies.py
    └── integration/
        ├── test_users.py
        └── test_items.py
```


Naming conventions

- Naming is critical --> `pytest` discovers test files and functions automatically.
- File naming: Prefix with `test_` → e.g., `test_users.py`
- Test function naming: Prefix with `test_` → e.g.,

```
def test_create_user_success():  
    ...  
  
def test_create_user_duplicate_email():  
    ...
```

- Describe what is being tested and under what condition

Unit Testing

- **Scope:** focused on *single* functions
- Dependencies are *mocked*

What to Unit Test

```
fake_db = {  
    1: {"id": 1, "name": "Alice"},  
    2: {"id": 2, "name": "Bob"}  
}  
  
def get_user_from_db(user_id: int):  
    return fake_db.get(user_id)  
  
@app.get("/users/{user_id}")  
def read_user(user_id: int, get_user=Depends(get_user_from_db)):  
    user = get_user  
    if not user:  
        raise HTTPException(status_code=404, detail="User not found")  
    return user
```

Unit Testing `read_user` (1)

```
import pytest
from fastapi.testclient import TestClient
from main import app, get_user_from_db

client = TestClient(app)

# --- Mock dependency ---
def mock_get_user(user_id: int):
    if user_id == 1:
        return {"id": 1, "name": "Mocked Alice"}
    return None

# --- Override dependency ---
app.dependency_overrides[get_user_from_db] = mock_get_user
```

Unit Testing `read_user` (2)

```
# --- Tests ---  
def test_read_user_success():  
    response = client.get("/users/1")  
    assert response.status_code == 200  
    assert response.json() == {"id": 1, "name": "Mocked Alice"}  
  
def test_read_user_not_found():  
    response = client.get("/users/999")  
    assert response.status_code == 404  
    assert response.json() == {"detail": "User not found"}
```

Integration Testing

- **Scope:** tests functions together with their real dependencies (not isolated/mock)
- **Database:** uses a real database instance, but separate from the production database (e.g., test-specific or in-memory)

What to Integration Test (1)

```
from sqlalchemy import create_engine, Column, Integer, String
from sqlalchemy.orm import declarative_base
from sqlalchemy.orm import sessionmaker, Session

DATABASE_URL = "sqlite:///./app.db" # real DB for dev, overridden in tests

engine = create_engine(DATABASE_URL, connect_args={"check_same_thread": False})
SessionLocal = sessionmaker(autocommit=False, autoflush=False, bind=engine)
Base = declarative_base()

## --- Model ---
class User(Base):
    __tablename__ = "users"
    id = Column(Integer, primary_key=True, index=True)
    name = Column(String, index=True)

## Create tables
Base.metadata.create_all(bind=engine)
```

What to Integration Test (2)

```
def get_db():  
    db = SessionLocal()  
    try:  
        yield db  
    finally:  
        db.close()  
  
@app.get("/users/integration/{user_id}")  
def read_user2(user_id: int, db: Session = Depends(get_db)):  
    user = db.query(User).filter(User.id == user_id).first()  
    if not user:  
        raise HTTPException(status_code=404, detail="User not found")  
    return {"id": user.id, "name": user.name}
```


Integration Testing `read_user2` (1)

```
import pytest
from fastapi.testclient import TestClient
from sqlalchemy import create_engine
from sqlalchemy.orm import sessionmaker
from main import app, Base, get_db, User

# --- Setup test database ---
SQLALCHEMY_DATABASE_URL = "sqlite:///./test.db"
engine = create_engine(SQLALCHEMY_DATABASE_URL, connect_args={"check_same_thread": False})
TestingSessionLocal = sessionmaker(autocommit=False, autoflush=False, bind=engine)

# Create fresh schema
Base.metadata.drop_all(bind=engine)
Base.metadata.create_all(bind=engine)
```

Integration Testing `read_user2` (2)

```
def override_get_db():
    db = TestingSessionLocal()
    try:
        yield db
    finally:
        db.close()

app.dependency_overrides[get_db] = override_get_db
client = TestClient(app)

# --- Fixtures ---
@pytest.fixture
def setup_test_data():
    db = TestingSessionLocal()
    user = User(id=1, name="Integration Alice")
    db.add(user)
    db.commit()
    db.refresh(user)
    db.close()
    return user
```

Integration Testing `read_user2` (3)

```
def test_read_user2_success(setup_test_data):  
    response = client.get(f"/users/integration/{setup_test_data.id}")  
    assert response.status_code == 200  
    assert response.json() == {"id": 1, "name": "Integration Alice"}  
  
def test_read_user2_not_found():  
    response = client.get("/users/integration/999")  
    assert response.status_code == 404  
    assert response.json() == {"detail": "User not found"}
```

Integration Testing - Key Points

- **Real DB:** The test uses SQLite (test.db) with SQLAlchemy.
- **Fresh schema:** We drop and recreate tables before tests to avoid stale state.
- **Fixtures:** `setup_test_data` inserts test data into the DB.
- **Overrides:** We override `get_db` so the app uses our test DB instead of production.
- **End result:** This is a true integration test — FastAPI endpoint + SQLAlchemy ORM & database working together.

CI/CD and Test coverage

- Continuous testing:
 - GitHub Actions / GitLab CI
 - Run tests on every push
- Test coverage

```
python3 -m pytest -cov
```

End-to-End Testing

- E2E Testing Setup
 - Run full app + DB + external services
 - Use `docker-compose`
- Tools for E2E
 - pytest with live server
 - Postman/Newman collections
 - Playwright (for frontend + backend)

Homework

[Link to homework](#)

Section: Practical Exercises: Testing APIs

Remember

- Write tests early
- Use dependency overrides
- Automate with CI/CD
- Balance between unit, integration, and E2E