MADS - Deployment 3

Testing

Raoul Grouls, 16 December 2024

Motivation for tests

- Catch Bugs Early
- Refactoring Confidence
- Executable Documentation
- Design Quality
- Deployment Safety
- Collaboration

Types of tests

- Unit tests: test individual components in isolation
- API tests: testing API endpoints
- Integration tests: test entire system running in docker

Best practices

1. **Test Isolation**: Each test should run independently
2. **Clear Naming**: test_what_condition_expectedresult
3. **Don't Test Implementation**: Test behavior, not how it's done
4. **Use Fixtures**: Share setup code between tests
5. **Test Edge Cases**: Not just the happy path
6. **Keep Tests Fast**: Slow tests don't get run
7. **One Assert Per Test**: Clear what failed
8. **Test Data Management**: Use small, focused datasets
9. **Version Control**: Tests should be in version control with code
10. **Automate:** automate testing with Makefile, docker-compose, lefthook, ci/cd

Some syntax

- Create a `tests/` folder
- Add a `__init__.py` file
- Add `pytest` as a depencency
- Test files must start with "test_" or end with “_test.py"
- Test classes must start with “Test"
- Test methods must start with “test_”
- Don't use `__init__` in test classes - use fixtures instead

Fixture

Fixture: Use when you need:

- Per-test isolation
- Dependency injection
- Reusability across multiple test files
- Cleanup after each test

```python
@pytest.mark.unit
class TestCalculator:
    @pytest.fixture
    def calculator(self):
        """Fixture to create a Calculator instance for each test."""
        return Calculator()

    def test_add_positive_numbers(self, calculator):
        """Test adding two positive numbers."""
        result = calculator.add(2, 3)
        assert result == 5
```

Setup_class

Setup_Class: Use when you need:

- One-time setup for entire class
- Shared state across test methods
- Performance optimization for expensive resources

```python
@pytest.mark.hypothesis
class TestCalculator:
    def setup_class(self):
        self.calculator = Calculator()
```

```python
@pytest.mark.unit
class TestCalculator:
    @pytest.fixture
    def calculator(self):
        """Fixture to create a Calculator instance for each test."""
        return Calculator()

    def test_add_positive_numbers(self, calculator):
        """Test adding two positive numbers."""
        result = calculator.add(2, 3)
        assert result == 5

    def test_add_negative_numbers(self, calculator):
        """Test adding negative numbers."""
        result = calculator.add(-1, -1)
        assert result == -2

    def test_add_zero(self, calculator):
        """Test adding zero."""
        result = calculator.add(5, 0)
        assert result == 5

    def test_divide_positive_numbers(self, calculator):
        """Test division with positive numbers."""
        result = calculator.divide(6, 2)
        assert result == 3

    def test_divide_by_zero(self, calculator):
        """Test division by zero raises ValueError."""
        with pytest.raises(ValueError) as exc_info:
            calculator.divide(5, 0)
        assert str(exc_info.value) == "Cannot divide by zero"
```

Hypothesis

Property-Based Testing with Hypothesis:

- Instead of testing specific examples (like 2 + 3), we test mathematical properties that should always hold true
- Hypothesis automatically generates hundreds of test cases with random inputs
- Tests run with many different values each time

Hypothesis

- The advantage of property-based testing is that it can find edge cases that you might not think to test manually.
- For example, it might find that your calculator breaks with very large numbers, negative zeros, or numbers very close to zero.

```python
@pytest.mark.hypothesis
class TestCalculator:
    def setup_class(self):
        self.calculator = Calculator()

    # Property-based tests using Hypothesis
    @given(
        x=st.floats(min_value=-1e6, max_value=1e6),
        y=st.floats(min_value=-1e6, max_value=1e6),
    )
    def test_add_properties(self, x, y):
        """
        Property-based test for addition with random floats.
        Tests properties like commutativity and identity.
        """
        # Filter out NaN and infinity
        assume(
            not any(
                map(lambda n: isinstance(n, float) and (isnan(n) or isinf(n)), [x, y])
            )
        )

        # Test commutativity: a + b = b + a
        assert self.calculator.add(x, y) == self.calculator.add(y, x)

        # Test identity: a + 0 = a
        assert self.calculator.add(x, 0) == x

        # Test associativity: (a + b) + c = a + (b + c)
        c = 42  # Fixed value for testing associativity
        epsilon = 1e-12
        left = self.calculator.add(self.calculator.add(x, y), c)
        right = self.calculator.add(x, self.calculator.add(y, c))
        assert abs(left - right) < epsilon
```

Hypothesis

- **Random Data Generation**: Automatically finds edge cases
- **Scientific Computing**: Tests numerical properties
- **Data Validation**: Tests data preprocessing robustness
- **Model Properties**: Tests model behavior across input space
- **Statistical Properties**: Tests distributional assumptions

Health api

- Monitor service availability
- Include things like memory usage, system resources, connectivity

```python
@app.get("/health")
async def health():
    try:
        test_add = calc.add(2, 3) == 5
        test_divide = calc.divide(6, 2) == 3

        status_code = 200 if (test_add and test_divide) else 500

        return JSONResponse(
            content={
                "status": "healthy" if (test_add and test_divide) else "degraded",
                "timestamp": datetime.datetime.now(timezone.utc).isoformat(),
                "uptime": psutil.Process().create_time(),
                "memory": {
                    "used": psutil.Process().memory_info().rss / 1024 / 1024,
                    "percent": psutil.Process().memory_percent(),
                },
            },
            status_code=status_code,
        )
    except Exception as e:
        return JSONResponse(
            content={"status": "unhealthy", "error": str(e)}, status_code=500
        )
```

Use docker compose for healthchecks

```yaml
services:
  calculator:
    build: .
    ports:
      - "8000:8000"
    healthcheck:
      test: ["CMD-SHELL", "curl -f http://localhost:8000/health && exit 0 || exit 1"]
      interval: 30s
      timeout: 3s
      retries: 2
      start_period: 10s
    restart: unless-stopped
```