Documentação dos Testes de Integração - DataProcessorService

1. Introdução

Este documento descreve, de forma detalhada, o processo de implementação dos testes de integração no projeto **DataProcessorService**, incluindo a justificativa da abordagem adotada, a estrutura utilizada e as práticas aplicadas. Os testes de integração têm como objetivo validar o comportamento correto do sistema como um todo, garantindo que diferentes componentes interajam entre si de forma consistente e alinhada aos requisitos de negócio.

Estes testes verificam a integração entre módulos internos (como camadas de lógica de negócio e persistência) e serviços externos (como MongoDB e MQTT), assegurando que a aplicação se comporta adequadamente em cenários reais e complexos.

---

2. Motivação pela Escolha de Testes de Integração

Com a crescente complexidade do sistema e a necessidade de garantir que as funcionalidades implementadas funcionem corretamente em conjunto, foi adotada a estratégia de testes de integração para complementar os testes unitários já existentes.

Diferente dos testes unitários, que validam funções isoladamente, os testes de integração buscam:

- Validar o fluxo completo de execução.
- Detectar falhas de comunicação entre componentes internos e externos.
- Garantir a compatibilidade e o comportamento esperado entre camadas e serviços.

Além disso, os testes de integração são essenciais para evitar regressões em funcionalidades críticas antes de realizar merges para as branches `develop` e `main`.

---

3. Processo de Planejamento e Execução

Antes do início de cada sprint, as tarefas que exigem testes de integração são previamente identificadas e selecionadas. Essas tarefas possuem **prioridade de entrega elevada** e

devem estar até o **Dia X da sprint** na coluna **"Pronto para Teste de Integração"** no planner da equipe.

Etapas:

1. **Planejamento**:
    - Reunião prévia para identificar riscos e confirmar se as tasks com testes de integração estarão prontas dentro do prazo.
2. **Criação de Exemplos de Teste**:
    - O responsável por DevOps/Testes cria exemplos de cenários de integração com base nos requisitos da tarefa.
3. **Execução dos Testes**:
    - O testador realiza os testes com base nos exemplos fornecidos.
    - Os testes são executados automaticamente pela pipeline e também podem ser feitos manualmente.
4. **Detecção e Correção de Erros**:
    - Em caso de falha, a pipeline acusa o erro.
    - Uma nova tarefa é criada, e um desenvolvedor cria uma branch de hotfix para corrigir o problema.
    - O ciclo de testes se reinicia para o hotfix, com o objetivo de resolução rápida.

O testador tem **até 3 dias** para concluir os testes, permitindo tempo para ajustes antes do encerramento da sprint.

---

4. Estrutura da Implementação

A estrutura dos testes de integração é organizada dentro do diretório `tests/`, em arquivos específicos para integração, separados dos testes unitários.

- **Fixtures** são utilizadas para criar dados falsos (mock data), simulando interações reais com o banco de dados e serviços externos.
- Após a execução de cada teste, os dados criados nas fixtures são automaticamente **excluídos do banco de dados**, mantendo o ambiente limpo.

5. Tecnologias Utilizadas

- **Python 3.13.2**: Linguagem base do projeto.
- **pytest**: Framework principal para testes.
- **unittest.mock**: Para mocking de objetos e serviços externos.
- **pymongo**: Comunicação com o banco de dados MongoDB.
- **paho-mqtt**: Cliente MQTT para envio/recebimento de mensagens.
- **pydantic**: Validação de modelos e configurações.
- **taskipy**: Ferramenta para automação de tarefas.
- **ruff**: Ferramenta de linting e formatação de código.

---

6. Implementação dos Testes

Os testes de integração são desenvolvidos com foco em simular a execução real do sistema, utilizando dados dinâmicos criados via fixtures.

Padrão de Teste Adotado:

1. **Configuração Inicial**: Criação de dados mockados ou fixtures.
2. **Execução**: Chamada da função ou endpoint real.
3. **Validação**:
    - Assertiva sobre o valor retornado.
    - Verificação de persistência no banco ou envio de mensagem via MQTT.
4. **Limpeza**: Os dados criados são automaticamente excluídos.

Exemplo:

```
def test_measurement_integration(measurement_fixture):
    response = client.post("/measurements/", json=measurement_fixture)
    assert response.status_code == 201

    result = client.get(f"/measurements/{response.json()['id']}")
    assert result.status_code == 200
    assert result.json()['value'] == measurement_fixture['value']
```

7. Execução dos Testes

Os testes podem ser executados automaticamente pela pipeline de CI/CD ou manualmente pelo desenvolvedor/testador.

```
[tool.taskipy.tasks]
run = 'fastapi dev main.py'
pre_test = 'task lint'
test = 'pytest --cov=app -vv'
post_test = 'coverage html'
lint = 'ruff check . && ruff check . --diff'
format = 'ruff check . --fix && ruff format .'
```

Para rodar os testes manualmente:
task test

8. Cuidados e Boas Práticas

- **Isolamento**: Cada teste cria e limpa seus dados, garantindo independência.
- **Automação Total**: A pipeline executa todos os testes automaticamente em PRs.
- **Cobertura Real**: Os testes simulam interações reais entre serviços e banco.
- **Validação de Fluxos**: Verificações abrangem dados retornados e efeitos colaterais.
- **Hotfix Ágil**: Em caso de falha, um desenvolvedor é alocado para correção imediata com branch própria.

9. Considerações Finais

A implementação de testes de integração no projeto **DataProcessorService** é essencial para validar o funcionamento completo das funcionalidades implementadas, principalmente em sistemas que dependem de comunicação entre múltiplos serviços e camadas.

Essa abordagem garante maior confiança no produto entregue e previne falhas graves em ambientes de produção, promovendo um ciclo de desenvolvimento mais seguro, robusto e confiável.

A estrutura adotada, baseada em automação, uso de fixtures, e divisão clara de responsabilidades, proporciona uma base sólida para a escalabilidade da suíte de testes conforme o projeto evolui.

---

10. Referências

- MESZAROS, Gerard. *xUnit test patterns: refactoring test code*. Addison-Wesley, 2007.
- PYTEST. *pytest documentation*. Disponível em: https://docs.pytest.org
- PYTHON SOFTWARE FOUNDATION. *unittest.mock — mock object library*. Disponível em: https://docs.python.org/3/library/unittest.mock.html
- PYDANTIC. *Pydantic documentation*. Disponível em: https://docs.pydantic.dev
- MONGODB. *MongoDB Python driver*. Disponível em: https://pymongo.readthedocs.io
- ECLIPSE FOUNDATION. *Eclipse Paho MQTT Python client*. Disponível em: https://www.eclipse.org/paho