1. Introdução

Este documento descreve, de forma detalhada, o processo de implementação de testes unitários no projeto *DataProcessorService*, desde a justificativa da abordagem escolhida até a explicação sobre a estrutura adotada e as práticas utilizadas. A utilização de testes unitários é uma prática fundamental no desenvolvimento de software moderno, sendo essencial para garantir que as funções implementadas correspondam corretamente aos requisitos definidos e se comportem conforme o esperado.

Os testes unitários implementados visam validar isoladamente cada função crítica do sistema, assegurando que, independentemente de suas dependências, o comportamento seja sempre previsível e controlado. A escolha por testes de unidade, e não por testes de integração ou de sistema neste momento, se deve à necessidade de garantir que as funções fundamentais estejam corretas, antes mesmo de se preocupar com fluxos completos ou interações entre componentes distintos.

2. Motivação pela escolha de testes unitários

A decisão por implementar testes unitários por função está diretamente relacionada às vantagens proporcionadas por esse tipo de abordagem. Ao testar cada função isoladamente, conseguimos identificar rapidamente eventuais erros, evitando que falhas pequenas se propaguem e impactem outras partes do sistema. Além disso, testes unitários favorecem a segurança no processo de refatoração de código, uma vez que garantem que modificações não alterem o comportamento das funções de maneira indesejada.

Outro aspecto relevante é a possibilidade de automatização. Testes unitários são rápidos, não dependem de infraestrutura complexa e podem ser facilmente integrados em pipelines de integração contínua (CI). Dessa forma, a equipe de desenvolvimento obtém feedback imediato sempre que uma nova versão do código é enviada para o repositório.

No contexto do *DataProcessorService*, a necessidade de integração com serviços externos, como o banco de dados MongoDB e o broker MQTT, reforça a importância de isolar os testes. Optou-se, portanto, pelo uso intensivo de técnicas de mocking, que permitem simular essas dependências, garantindo que os testes possam ser executados sem a necessidade de configurar ou manter instâncias reais desses serviços.

3. Estrutura da implementação

A estrutura dos testes foi cuidadosamente organizada para refletir a divisão lógica do sistema. Todos os testes foram concentrados em uma pasta chamada *tests*, que se localiza na raiz do projeto. Esta pasta contém três arquivos principais, cada um responsável por testar um conjunto de funções relacionado a um domínio específico do sistema.

O arquivo *test_storage.py* contém os testes das funções relacionadas à lógica de negócio, como a validação e construção de objetos de alerta e medida. Já o arquivo *test_ingestion.py* é dedicado à validação das operações de persistência no banco de dados MongoDB, como a inserção, recuperação e remoção de dados. Por fim, o arquivo *test_config.py* contém os testes dos componentes relacionados à infraestrutura, especialmente aqueles que lidam com conexões e configurações para os serviços de MongoDB e MQTT.

Essa divisão visa não apenas a organização do código, mas também a facilitação do entendimento e da manutenção dos testes. Caso novos componentes sejam adicionados ao sistema no futuro, novos arquivos podem ser criados seguindo a mesma lógica de separação por contexto.

4. Tecnologias utilizadas

O projeto é desenvolvido na linguagem Python, na versão 3.13.2, que oferece ampla compatibilidade com bibliotecas de suporte a testes e integração com serviços diversos. Para a implementação dos testes unitários, utilizou-se o framework *pytest*, que proporciona uma sintaxe simples, poderosa e altamente configurável, adequada tanto para projetos pequenos quanto para sistemas complexos.

Para a criação de objetos simulados, utilizou-se o módulo unittest.mock, que permite criar substitutos para objetos reais, interceptar chamadas de métodos e definir comportamentos específicos durante a execução dos testes. Além disso, as bibliotecas pymongo e paho-mqtt foram mantidas como dependências do projeto, pois fazem parte do funcionamento das funções testadas, mesmo que, nos testes, suas funcionalidades sejam simuladas.

A biblioteca pydantic foi empregada na definição de modelos e validação de configurações. Durante o processo de testes, entretanto, os modelos e configurações foram frequentemente simulados por meio de mocks, evitando dependências diretas do ambiente ou necessidade de configurações específicas para execução dos testes.

5. Implementação dos testes

A principal estratégia adotada na implementação dos testes foi o isolamento completo das funções testadas, evitando qualquer tipo de dependência de infraestrutura externa ou estado compartilhado entre os testes. Isso foi possível através do uso extensivo de mocks, criados com unittest.mock.

O padrão seguido para a escrita de cada teste foi o seguinte: inicialmente, configurava-se o ambiente de teste, o que incluía a criação de instâncias mockadas das classes dependentes ou a substituição de funções por versões simuladas. Em seguida, a função em teste era executada com parâmetros controlados, cuidadosamente escolhidos para representar cenários típicos, extremos ou de erro. Por fim, o resultado da execução era validado utilizando asserções (assert), que verificavam se o valor retornado ou o efeito colateral produzido estava de acordo com o esperado.

Nos casos em que a função em teste realizava chamadas internas a outros métodos ou serviços, também foi verificado, por meio de métodos como assert_called_once_with, se essas chamadas foram feitas corretamente.

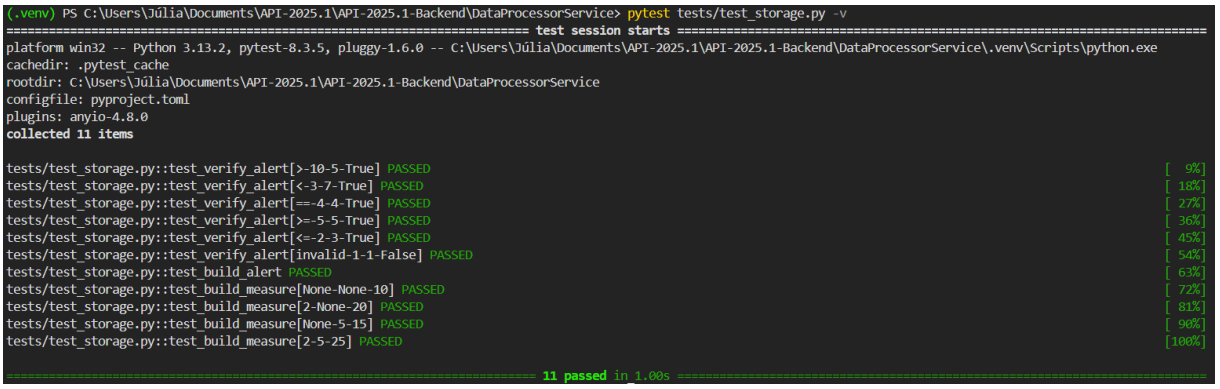
Por exemplo, ao testar a função de inserção de dados no MongoDB, verificou-se não apenas que a função insert_data executava sem erros, mas também que o método insert_one da coleção Mongo havia sido chamado exatamente uma vez, com os dados fornecidos como parâmetro.

6. Execução dos testes

A execução dos testes é realizada de maneira simples, utilizando-se o comando pytest -v a partir da raiz do projeto. O parâmetro -v (verbose) é recomendado, pois fornece uma saída mais detalhada, listando cada teste executado, bem como seu resultado individual.

Caso se deseje executar apenas um subconjunto dos testes, ou mesmo um teste específico, é possível informar o caminho para o arquivo de teste como argumento para o comando. Por exemplo, para executar apenas os testes relacionados à lógica de negócios, presentes no arquivo test_storage.py, pode-se utilizar:

pytest tests/test_storage.py -v



Esta flexibilidade permite que a equipe de desenvolvimento valide rapidamente partes específicas do sistema, conforme necessidade, sem a obrigatoriedade de executar toda a suíte de testes.

7. Cuidados e práticas recomendadas

Durante a implementação dos testes, algumas práticas foram adotadas para garantir a qualidade, a confiabilidade e a manutenção da suíte de testes. Primeiramente, assegurou-se que todas as dependências externas fossem devidamente mockadas, evitando que o sucesso ou falha dos testes dependesse da disponibilidade de serviços como bancos de dados ou brokers de mensagens.

Além disso, buscou-se sempre validar não apenas os valores retornados pelas funções, mas também os efeitos colaterais e as interações com outros objetos e componentes do sistema. Essa abordagem mais abrangente confere maior robustez aos testes, pois garante que as funções não apenas produzem o resultado correto, mas também seguem o fluxo de execução esperado.

Por fim, adotou-se uma estrutura de organização clara, com separação por contexto, o que facilita a compreensão, a manutenção e a expansão futura da suíte de testes.

8. Comandos utilizados durante o processo

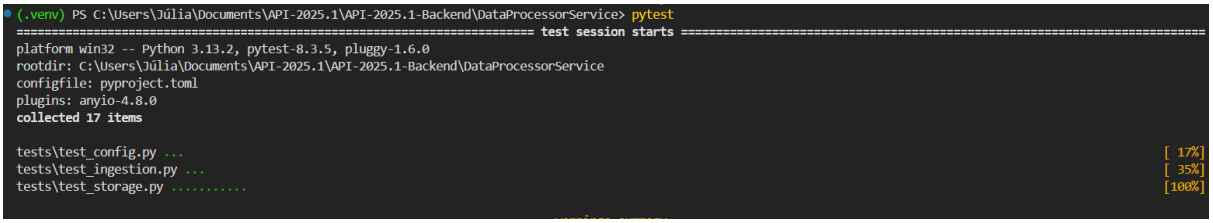
Durante o desenvolvimento e execução dos testes, os seguintes comandos foram utilizados:

Para instalar a biblioteca de testes:

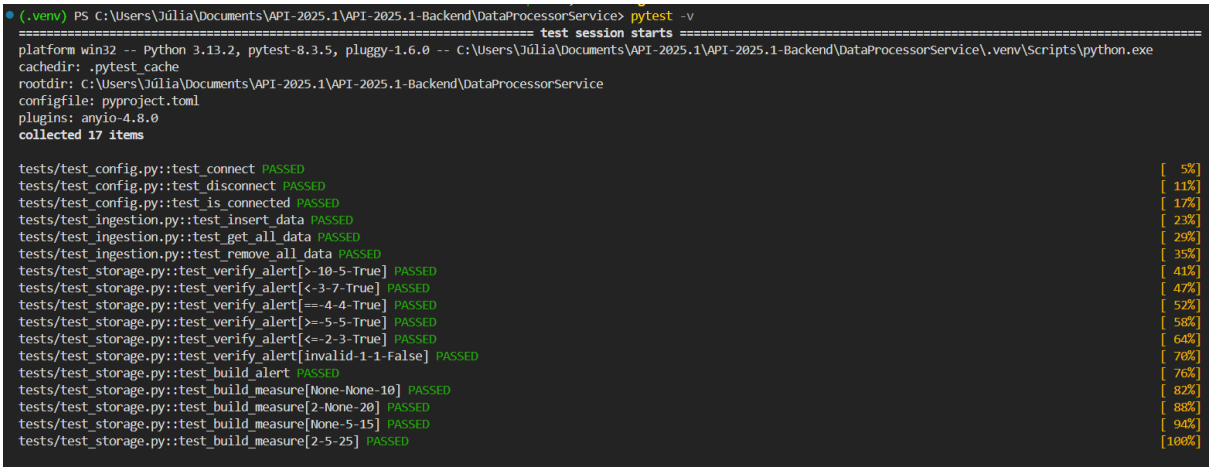
pip install pytest

Para executar toda a suíte de testes:

pytest

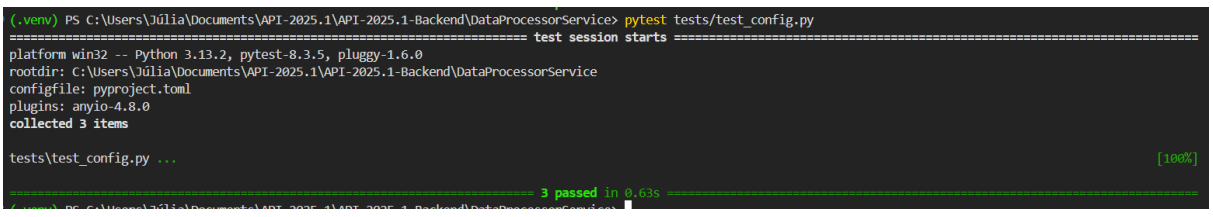


pytest -v

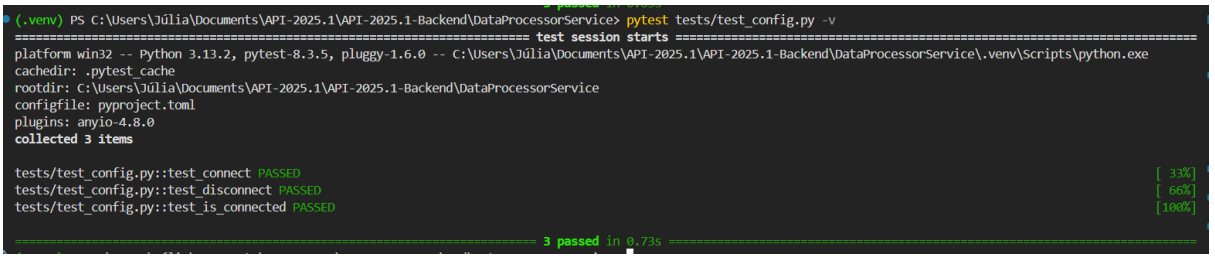


Para executar um arquivo de teste específico:

pytest tests/test_config.py



pytest tests/test_config.py -v



Estes comandos são padrão no ambiente Python e proporcionam uma execução rápida e eficaz dos testes.

9. Considerações finais

A implementação de testes unitários no projeto *DataProcessorService* foi fundamental para garantir a confiabilidade e a qualidade das funções críticas do sistema. A adoção da abordagem de testes por função isolada permitiu identificar rapidamente potenciais falhas, além de proporcionar segurança para futuras manutenções e evoluções do código.

O uso de mocks para simular dependências externas foi decisivo para a obtenção de testes determinísticos e independentes, que podem ser executados de forma repetida e consistente, sem a necessidade de configuração de ambientes externos.

A estrutura adotada favorece a organização e a clareza, sendo facilmente expansível para abarcar novos componentes ou funcionalidades que venham a ser adicionados ao sistema no futuro. Como evolução natural, recomenda-se a implementação de testes de integração e de sistema, que permitirão validar o funcionamento conjunto dos diversos componentes e garantir a aderência do sistema como um todo aos requisitos de negócio.

10. Referências

MESZAROS, Gerard. *xUnit test patterns: refactoring test code*. Boston: Addison-Wesley, 2007.

PYTEST. *pytest documentation*. Disponível em: https://docs.pytest.org. Acesso em: 21 maio 2025.

PYTHON SOFTWARE FOUNDATION. *unittest.mock — mock object library*. Disponível em: https://docs.python.org/3/library/unittest.mock.html. Acesso em: 21 maio 2025.

PYDANTIC. *Pydantic documentation*. Disponível em: https://docs.pydantic.dev. Acesso em: 21 maio 2025.

MONGODB. *MongoDB Python driver*. Disponível em: https://pymongo.readthedocs.io. Acesso em: 21 maio 2025.

ECLIPSE FOUNDATION. *Eclipse Paho MQTT Python client*. Disponível em: https://www.eclipse.org/paho/. Acesso em: 21 maio 2025.