1. Introdução

O desenvolvimento de sistemas computacionais de alta qualidade requer a aplicação sistemática de processos que assegurem a correção, a confiabilidade e a manutenção do código. Entre esses processos, destaca-se a prática de testes unitários, uma das técnicas fundamentais da engenharia de software para detecção precoce de defeitos.

No contexto do projeto WeatherDataService, que visa a implementação de serviços relacionados ao processamento e análise de dados meteorológicos, foi adotada uma abordagem metodológica baseada na realização de testes unitários por função isolada.

Esta documentação objetiva apresentar, de forma minuciosa e estruturada, o padrão de testes unitários estabelecido para o projeto, o detalhamento do processo de implementação, as justificativas técnicas para a escolha deste modelo, a metodologia para execução dos testes e os critérios objetivos para avaliação do sucesso.

Adicionalmente, busca-se garantir que todos os profissionais e acadêmicos envolvidos compreendam integralmente os fundamentos, as práticas e as decisões técnicas que sustentam este modelo de testes, consolidando a qualidade e a sustentabilidade do desenvolvimento.

2. Padrão de Testes Unitários Adotado

A padronização adotada para os testes unitários no projeto WeatherDataService fundamenta-se no conceito de teste de unidade, definido como a verificação isolada de um elemento mínimo de código, tal como uma função ou método, assegurando que esse elemento opere conforme o especificado, de forma determinística e previsível.

Diferentemente de testes de integração ou de sistema, que validam a interação entre múltiplos componentes ou a aplicação como um todo, os testes unitários concentram-se exclusivamente em validar o comportamento individual e isolado de uma função, sendo, portanto, independentes do contexto global.

2.1 Princípios norteadores

- **Isolamento:**

Cada teste unitário é desenhado para verificar uma única função em condições controladas, sem dependências externas como bancos de dados, sistemas de arquivos ou serviços de terceiros.

- **Determinismo:**

O comportamento testado deve ser previsível e reprodutível, isto é, dada a mesma entrada, a função deve produzir sempre o mesmo resultado.

- **Responsabilidade única:**

Cada teste deve verificar apenas um comportamento ou um aspecto da função, evitando validar múltiplas funcionalidades em um mesmo teste.

- **Clareza e objetividade:**

O propósito de cada teste deve ser evidente pela sua nomenclatura e pela sua estrutura.

2.2 Estrutura padrão dos testes

Todos os testes unitários foram organizados na pasta específica:

/tests/unit/functions

Cada arquivo de teste possui nomenclatura correspondente à função testada, seguindo a convenção:

test_<nome_da_função>.py

Por exemplo:

- Para a função password_hash → test_password_hash.py.
- Para a função convert de ConvertDates → test_convert.py.

2.3 Estrutura lógica dos testes: AAA (Arrange, Act, Assert)

Cada função de teste segue a consagrada estrutura AAA, utilizada mundialmente na comunidade de engenharia de software:

1. **Arrange (Preparação):**

Configuração do ambiente de teste e preparação dos dados de entrada necessários.

2. **Act (Ação):**

Execução da função sob teste com os dados previamente preparados.

3. **Assert (Asserção):**

Verificação do resultado obtido, confrontando-o com o resultado esperado para confirmar a correção do comportamento.

Este modelo propicia **clareza**, **previsibilidade** e **padronização** na construção dos testes.

3. Como foi implementado

A implementação dos testes unitários foi conduzida com rigor metodológico, dividida nas seguintes etapas:

3.1 Levantamento e seleção das funções a serem testadas

Inicialmente, procedeu-se à identificação das funções criticamente importantes no contexto do sistema. Foram priorizadas funções que:

- Executam lógica de negócios relevante.
- São amplamente reutilizadas por outros componentes.
- Demandam elevada confiabilidade.

As principais funções selecionadas foram:

- ConvertDates → responsável pela transformação de timestamps em objetos datetime.
- Singleton → garante a instância única de classes, conforme o padrão de projeto Singleton.
- PasswordManager → encarregado pela criptografia e validação de senhas.
- TokenManager → provê a geração de tokens JWT para autenticação.
- BasicResponse → estrutura genérica para encapsulamento de respostas.

3.2 Definição dos casos de teste

Para cada função, foram definidos:

- **Cenários positivos:** verificando se o comportamento esperado ocorre com entradas válidas.

Ex.: verify_password retorna True quando a senha correta é fornecida.

- **Cenários negativos:** avaliando o comportamento com entradas inválidas ou limites.

Ex.: verify_password retorna False quando uma senha incorreta é fornecida.

Além disso, sempre que pertinente, foram incluídos testes para entradas de borda, como None, strings vazias, números negativos ou limites superiores.

3.3 Implementação dos testes utilizando Pytest

A ferramenta escolhida para execução e gerenciamento dos testes foi o Pytest, justificada por sua:

- Sintaxe clara e objetiva.
- Suporte a fixtures e mocks.
- Compatibilidade ampla com sistemas Python.
- Capacidade de gerar relatórios detalhados.

Cada teste foi implementado como uma função Python, iniciando-se obrigatoriamente com o prefixo test_, conforme a convenção estabelecida pelo Pytest para auto-descoberta de testes.

3.4 Garantia de isolamento absoluto

Todos os testes foram desenvolvidos com o princípio de isolamento, assegurando que:

- Não dependem de um estado global.
- Não realizam operações de entrada/saída.
- Não utilizam serviços externos.

Quando necessário, criaram-se objetos simulados (mocks), por exemplo, no caso da simulação de um usuário para geração de token.

Esse isolamento é fundamental para garantir a confiabilidade, reprodutibilidade e velocidade dos testes.

4. Por que foi escolhido este modelo

A adoção do modelo de testes unitários por função isolada decorreu de múltiplos fatores técnicos e estratégicos:

4.1 Modularidade

Testes isolados promovem maior clareza e entendimento sobre o comportamento de cada unidade do sistema, facilitando a manutenção e evolução do código.

4.2 Eficiência

Por não dependerem de infraestrutura externa ou de estado compartilhado, os testes unitários são extremamente rápidos, podendo ser executados de maneira contínua e automatizada a cada alteração no código.

4.3 Precisão na detecção de defeitos

Quando um teste falha, é possível identificar com precisão qual função específica apresentou comportamento incorreto, reduzindo o tempo de diagnóstico e correção.

4.4 Aderência às melhores práticas

O modelo está alinhado com os preceitos do Desenvolvimento Orientado a Testes (TDD), promovendo a construção de código mais confiável, modular e testável.

4.5 Facilidade de manutenção e expansão

Com a estrutura por função isolada, a adição de novos testes requer apenas a criação de novos arquivos na pasta tests/unit/functions, mantendo a organização e escalabilidade do projeto.

5. Como executar os testes unitários

5.1 Pré-requisitos

Antes de proceder à execução dos testes, devem ser atendidas as seguintes condições:

- Ativação do ambiente virtual do projeto.
- Instalação de todas as dependências, incluindo o pytest.

Posicionamento do terminal na raiz do projeto.
Configuração adequada de variáveis de ambiente, se necessárias.

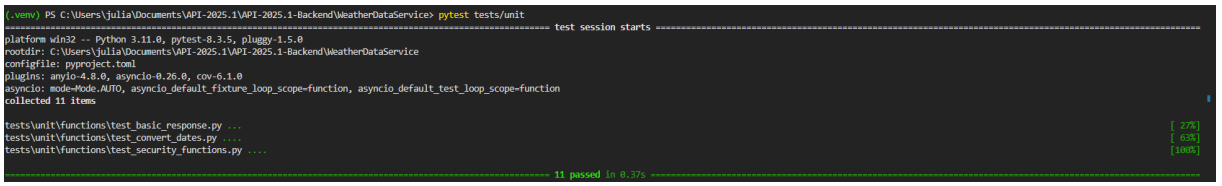
5.2 Executar todos os testes unitários

Com o terminal posicionado na raiz do projeto, executar:

pytest tests/unit

Este comando:

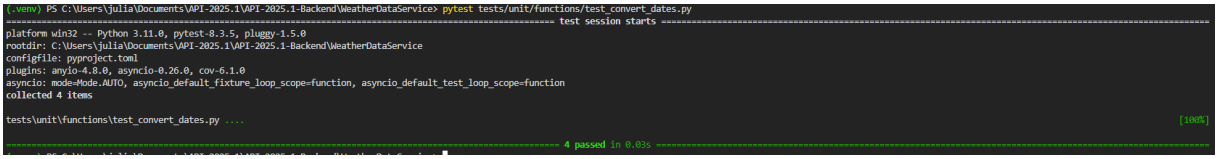
- Instrui o Pytest a buscar, de maneira recursiva, todos os arquivos iniciados por test_ dentro da pasta tests/unit.
- Executa todas as funções de teste contidas nesses arquivos.
- Gera um relatório com o status de cada teste.



5.3 Executar um arquivo de teste específico

Para executar apenas os testes de uma função específica, por exemplo ConvertDates:

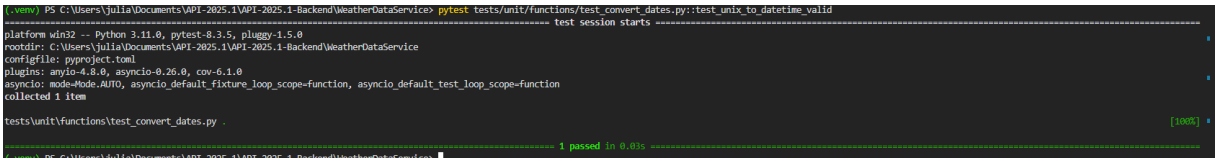
pytest tests/unit/functions/test_convert_dates.py



5.4 Executar uma função de teste específica

Se for necessário executar somente uma função específica de teste dentro de um arquivo, por exemplo test_unix_to_datetime, utilizar:

pytest tests/unit/functions/test_convert_dates.py::test_unix_to_datetime_valid



5.5 Executar com relatório detalhado (modo verbose)

Para obter um relatório detalhado, exibindo o nome de cada teste e seu resultado, utilizar:

pytest -v

O relatório informará:

- O nome de cada teste.

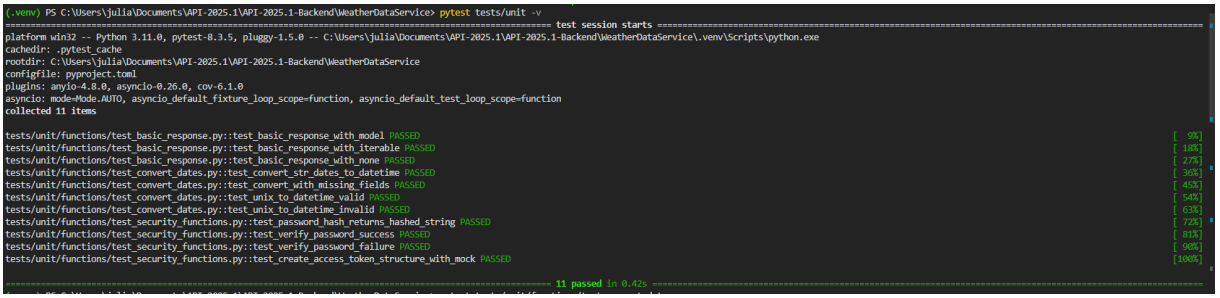
- O resultado:

  ✔ **Passed** — teste bem-sucedido.

  ✘ **Failed** — teste falhou.

  ⚠ **Error** — erro de execução.

- O tempo de execução de cada teste.



6. Como saber se os testes passaram ou não

Após a execução, o Pytest apresentará um resumo no terminal contendo:

- O número total de testes executados.
- Quantos passaram.
- Quantos falharam.
- Quantos apresentaram erros.

Critérios para considerar a execução bem-sucedida:

- Todos os testes devem obter o status Passed.
- Nenhum teste pode apresentar status Failed ou Error.
- O comportamento validado pelos testes deve estar em conformidade com a regra de negócio implementada.

O que fazer em caso de falha:

1. Revisar a função testada, identificando o motivo da falha.
2. Corrigir a função ou, se for o caso, ajustar o teste, caso a regra de negócio tenha sido alterada.
3. Executar novamente os testes para garantir que todos estejam passando.

Nota:

Testes que falham indicam problemas a serem resolvidos; nunca devem ser ignorados ou desativados.

7. Papel dos Testes Unitários no Fluxo de Integração Contínua

No contexto do projeto, os testes unitários desempenham um papel fundamental na garantia de qualidade e estabilidade do código desde os primeiros estágios do desenvolvimento até o deploy final. Conforme o fluxo de Integração Contínua utilizado pela equipe, os testes unitários estão integrados em etapas automatizadas da pipeline e são executados de forma contínua a cada modificação relevante no repositório.

7.1 Etapas do processo com foco nos testes unitários

- **Push para branch Feature**:
  Assim que o desenvolvedor realiza um push para a branch de feature, o pipeline de CI é iniciado. Nesta etapa, os testes unitários são executados juntamente com as ferramentas de verificação de estilo, como Linter e Mypy. Isso assegura que alterações simples já passem por validações automáticas, impedindo que código com comportamentos quebrados avance para o repositório principal.
- **Pull Request para branch Develop**:
  Ao abrir um Pull Request, o código é novamente validado por testes unitários. Além disso, são executados testes de integração para validar a interação entre partes do sistema. Se os testes unitários falharem, o merge para a branch develop é bloqueado automaticamente. Essa etapa permite que o ambiente de homologação contenha apenas código funcional validado de forma isolada e integrada.
- **Pull Request para branch Release**:
  Nesta etapa, ocorre uma validação mais rigorosa: além da execução de testes unitários e de integração, o código passa por ferramentas de análise de qualidade, como o Sonar. A aprovação do Pull Request só é possível se todos os testes passarem e os critérios de qualidade estiverem atendidos. O objetivo aqui é garantir que nenhuma falha de regra de negócio ou lógica seja levada para a produção.
- **Merge final para branch Main**:
  Antes do deploy automático da versão final, o pipeline executa novamente os testes unitários e de integração. Caso qualquer etapa falhe, o processo é interrompido. Isso garante que a versão entregue ao ambiente de produção esteja completamente validada.

Por que isso é importante?

- Servem como barreira inicial contra erros lógicos simples que podem causar efeitos em cascata no sistema.
- Facilitam a manutenção, pois ao identificar falhas, mostram com precisão o módulo/função responsável.
- Ajudam no desenvolvimento seguro de novas features, uma vez que alterações são testadas automaticamente.

- Reduzem o tempo de revisão manual, pois automatizam a validação de regras de negócio.
- Estão integrados ao pipeline de CI, tornando a validação contínua e em tempo real, sem depender de esforço manual.

A integração dos testes unitários no fluxo de CI garante que cada nova entrega seja validada automaticamente em minutos, sem a necessidade de validação manual para comportamentos elementares. Isso resulta em desenvolvimento mais ágil, seguro e com menor custo de manutenção.

8. Considerações finais

A aplicação disciplinada e sistemática de testes unitários por função isolada, conforme delineado nesta documentação, assegura a construção de um sistema robusto, manutenível e de alta qualidade.

Esse modelo promove:

- Detecção precoce de defeitos.
- Agilidade no desenvolvimento.
- Conformidade com as melhores práticas da engenharia de software.