Relatório de Validação e Testes Unitários da API

Este documento detalha a metodologia de testes e o processo de depuração aplicado à API de back-end do sistema de agendamento.

1. Metodologia de Teste

A validação da API foi conduzida através de testes unitários automatizados para garantir a correção funcional dos *endpoints* de CRUD. A pilha de testes foi configurada da seguinte forma:

- **Framework:** Pytest.
- **Cliente HTTP:** TestClient do FastAPI, para executar requisições à API em memória.
- **Banco de Dados:** SQLite operando em modo *in-memory* (:memory:).
- **Isolamento:** A dependência get_db do FastAPI foi sobrescrita (dependency_overrides) para injetar a sessão do banco de dados de teste, garantindo que os testes não tivessem estado persistente nem interferissem no banco de dados de desenvolvimento.

2. Processo de Depuração e Resolução de Erros

A execução inicial dos testes revelou um comportamento anômalo: operações de leitura (GET) e de falha lógica (ex: POST /agendar em slot ocupado, retornando 400) passavam. No entanto, todas as operações de escrita bem-sucedidas (POST /agendar em slot livre e POST /cancelar) falhavam com HTTP 500 Internal Server Error.

O processo para diagnosticar e corrigir esta falha seguiu as etapas abaixo.

2.1. Diagnóstico: Exposição da Exceção Raiz

O manipulador de exceção genérico (except Exception as e:) nos *endpoints* do main.py estava mascarando a exceção original, retornando apenas uma mensagem genérica.

- **Ação:** O manipulador de exceção foi modificado temporariamente para incluir o tipo e a mensagem da exceção original na resposta JSON: detail=f"ERRO REAL: {type(e).__name__} - {str(e)}"
- **Ação Complementar:** Os testes (test_main.py) foram atualizados para imprimir o corpo da resposta JSON em caso de falha de *assert* de status.

2.2. Causa-Raiz Identificada

A execução dos testes com a modificação acima expôs a exceção subjacente: TypeError: can't compare offset-naive and offset-aware datetimes

- **Análise:** O erro ocorria na lógica de negócio (crud.py) ao comparar um datetime com fuso horário (offset-aware), proveniente do código (datetime.now(UTC)), com um

datetime sem fuso horário (offset-naive), retornado pelo banco de dados SQLite. O Python não permite esta comparação direta.

2.3. Decisão de Engenharia e Resolução

Foi necessário padronizar o tratamento de datas e horas em toda a aplicação.

- **Tentativa 1 (Falha):** A aplicação de DateTime(timezone=True) nos modelos (models.py) não foi eficaz, pois o driver do SQLite em modo *in-memory* não implementa o suporte a fuso horário de forma consistente.
- **Decisão e Solução 2 (Sucesso):** Foi adotado o padrão **"Naive-UTC"**. Esta abordagem consiste em tratar todos os *timestamps* como objetos datetime *naive* (sem fuso), onde todos os valores representam implicitamente o horário UTC.
    1. Toda a aplicação (modelos, lógica CRUD e testes) foi refatorada para usar datetime.utcnow() em vez de datetime.now(UTC).
    2. Isso garantiu que as comparações fossem sempre entre dois objetos *naive* (naive <= naive), resolvendo o TypeError.

3. Resultado e Validação Final

Após a refatoração para o padrão "Naive-UTC", todos os 6 testes unitários foram executados com sucesso, validando a funcionalidade completa da API.

Os DeprecationWarning referentes ao uso de datetime.utcnow() (observados na saída do pytest) são esperados e foram intencionalmente suprimidos no pytest.ini. Esta é uma ação consciente, documentando que o uso da função é uma decisão de engenharia para garantir a compatibilidade com o back-end do SQLite.