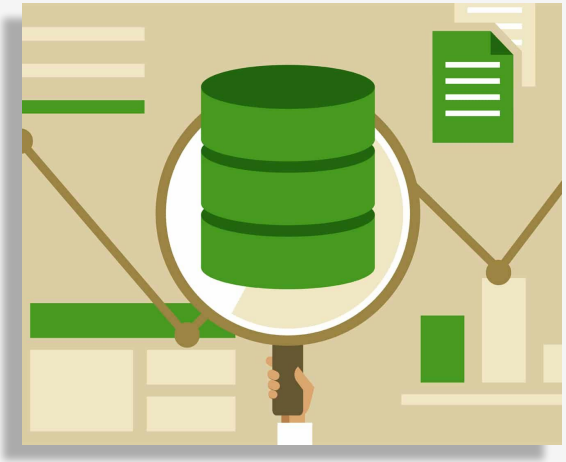


Base de dados ~~Food Delivery~~

Marta Vieira a046756
Juliana Moreira a047188
Felipe Castilho a047152

Introdução

Este trabalho descreve uma implementação técnica de uma camada de base de dados, de um projeto que visa criar um sistema de gestão de entregas de comida, "*Food Delivery*".

Focando em três aspetos fundamentais:

- **Propriedades ACID:** Garantia de transações confiáveis
- **Sistema de Audit:** Rastreamento de alterações na base de dados
- **Replicação e Alta Disponibilidade:** Arquitetura Master-Replica com MaxScale

Tabelas Principais

Tabela dos Códigos Postais

```sql
-- TABELA: codpostal
DROP TABLE IF EXISTS codpostal;
CREATE TABLE codpostal (
    codpostal CHAR(8) PRIMARY KEY,
    localidade VARCHAR(50) NOT NULL,
    cidade VARCHAR(30) NOT NULL
);
```

Tabelas Principais

Tabela dos Entregadores

```sql
-- TABELA: entregadores
DROP TABLE IF EXISTS entregadores;
CREATE TABLE entregadores (
    id INT AUTO_INCREMENT PRIMARY KEY,
    nome VARCHAR(50) NOT NULL,
    email VARCHAR(50),
    telefone CHAR(9),
    codpostal CHAR(8),
    estado ENUM('disponivel', 'ocupado', 'indisponivel') DEFAULT 'disponivel',
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP,
    FOREIGN KEY (codpostal) REFERENCES codpostal(codpostal)
);
```

Tabelas Principais

Tabela dos Clientes

```sql
-- TABELA: clientes
DROP TABLE IF EXISTS clientes;
CREATE TABLE clientes (
    id INT AUTO_INCREMENT PRIMARY KEY,
    nome VARCHAR(512) NOT NULL,
    email VARCHAR(512) NOT NULL,
    telefone CHAR(9) NOT NULL,
    morada VARCHAR(512) NOT NULL,
    codpostal CHAR(8),
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP,
    FOREIGN KEY (codpostal) REFERENCES codpostal(codpostal)
);
```

Tabelas Principais

Tabela dos Restaurantes

```sql
-- TABELA: restaurantes
DROP TABLE IF EXISTS restaurantes;
CREATE TABLE restaurantes (
    id INT AUTO_INCREMENT PRIMARY KEY,
    nome VARCHAR(512) NOT NULL,
    morada VARCHAR(512),
    codpostal CHAR(8),
    email VARCHAR(512),
    telefone CHAR(9),
    especialidade_id INT,
    hora_abertura VARCHAR(512),
    hora_fecho VARCHAR(512),
    estado ENUM('aberto', 'fechado') DEFAULT 'fechado',
    descricao VARCHAR(512),
    taxa_entrega FLOAT,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP,
    FOREIGN KEY (especialidade_id) REFERENCES categorias_pratos(id),
    FOREIGN KEY (codpostal) REFERENCES codpostal(codpostal)
);
```

Tabelas Principais

Tabela dos Ingredientes

```sql
-- TABELA: ingredientes
DROP TABLE IF EXISTS ingredientes;
CREATE TABLE ingredientes (
    id INT AUTO_INCREMENT PRIMARY KEY,
    nome VARCHAR(512) NOT NULL,
    tipo VARCHAR(512) NOT NULL,
    alergeno TINYINT(1) DEFAULT FALSE,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

Tabelas Principais

Tabela dos Pratos

```sql
-- TABELA: pratos
DROP TABLE IF EXISTS pratos;
CREATE TABLE pratos (
    id INT AUTO_INCREMENT PRIMARY KEY,
    restaurante_id INT,
    categoria_id INT,
    nome VARCHAR(512) NOT NULL,
    preco INT NOT NULL,
    descricao VARCHAR(512),
    disponivel TINYINT(1) DEFAULT FALSE,
    vegetariano TINYINT(1) DEFAULT FALSE,
    vegan TINYINT(1) DEFAULT FALSE,
    sem_gluten TINYINT(1) DEFAULT FALSE,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP,
    FOREIGN KEY (restaurante_id) REFERENCES restaurantes(id) ON DELETE CASCADE,
    FOREIGN KEY (categoria_id) REFERENCES categorias_pratos(id)
);
```

Tabelas Principais

Tabela dos Pedidos

```sql
-- TABELA: pedidos
DROP TABLE IF EXISTS pedidos;
CREATE TABLE pedidos (
    id INT AUTO_INCREMENT PRIMARY KEY,
    cliente_id INT NOT NULL,
    restaurante_id INT NOT NULL,
    metodo_pagamento ENUM('cartao', 'dinheiro', 'mbway', 'multibanco', 'paypal') DEFAULT 'dinheiro',
    hora_pedido DATETIME,
    total FLOAT DEFAULT 0,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP,
    FOREIGN KEY (cliente_id) REFERENCES clientes(id),
    FOREIGN KEY (restaurante_id) REFERENCES restaurantes(id)
);
```

Tabelas Principais

Tabela das Entregas

```sql
-- TABELA: entregas
DROP TABLE IF EXISTS entregas;
CREATE TABLE entregas (
    id INT AUTO_INCREMENT PRIMARY KEY,
    pedido_id INT UNIQUE,
    entregador_id INT NOT NULL,
    restaurante_id INT NOT NULL,
    tempo_estimado_min INT,
    tempo_real_min INT,
    estado ENUM('pendente', 'a_caminho', 'entregue', 'cancelado') DEFAULT 'pendente',
    hora_entrega VARCHAR(512),
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP,
    FOREIGN KEY (pedido_id) REFERENCES pedidos(id),
    FOREIGN KEY (entregador_id) REFERENCES entregadores(id),
    FOREIGN KEY (restaurante_id) REFERENCES restaurantes(id)
);
```

Tabelas Principais

Tabela de Categorías de Pratos

```sql
-- TABELA: categorias_pratos
DROP TABLE IF EXISTS categorias_pratos;
CREATE TABLE categorias_pratos (
    id INT AUTO_INCREMENT PRIMARY KEY,
    nome VARCHAR(50) NOT NULL UNIQUE,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

Tabelas de Associação

Tabela Pratos-Ingredientes

```sql
-- TABELA: pratos_ingredientes
DROP TABLE IF EXISTS pratos_ingredientes;
CREATE TABLE pratos_ingredientes (
    id INT AUTO_INCREMENT PRIMARY KEY,
    prato_id INT,
    ingrediente_id INT,
    quantidade VARCHAR(512),
    obrigatorio TINYINT(1) DEFAULT TRUE,
    unidade VARCHAR(20) DEFAULT 'g',
    FOREIGN KEY (prato_id) REFERENCES pratos(id) ON DELETE CASCADE,
    FOREIGN KEY (ingrediente_id) REFERENCES ingredientes(id)
);
```

Um prato tem vários ingredientes e um ingrediente pertence a vários pratos.

Nesta tabela definimos a quantidade de ingredientes num prato, a sua respetiva unidade e se o mesmo é obrigatório.

Tabelas de Associação

Tabela Pedidos-Pratos

```sql
-- TABELA: pedidos_pratos
DROP TABLE IF EXISTS pedidos_pratos;
CREATE TABLE pedidos_pratos (
    id INT AUTO_INCREMENT PRIMARY KEY,
    pedido_id INT,
    prato_id INT,
    quantidade INT DEFAULT 1,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    FOREIGN KEY (pedido_id) REFERENCES pedidos(id) ON DELETE CASCADE,
    FOREIGN KEY (prato_id) REFERENCES pratos(id)
);
```

Um pedido tem vários pratos e um prato pertence a vários pedidos.

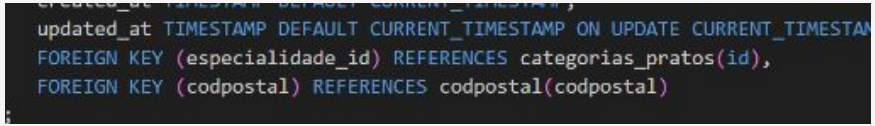
Nesta tabela definimos a quantidade de pratos num pedido e a sua sua respetiva quantidade.

Outras relações identificadas

Relações 1 para muitos
1 entregador tem várias entregas.
1 restaurante tem vários pedidos
1 restaurante tem vários pratos

Relações 1 para 1
1 entrega tem 1 entregador .
1 pedido tem 1 cliente
1 pedido pertence a 1 restaurante
1 prato tem 1 categoria
1 restaurante tem 1 especialidade (categoria)
1 cliente tem 1 codpostal
...

Estas relações são definidas através do uso das chaves estrangeiras com por exemplo na tabela restaurantes:



Tabelas Auditadas

Conseguimos criar um sistema de audit criando uma tabela para este efeito e registando as alterações com triggers e variáveis created_at e updated_at presentes noutras tabelas.
Os triggers de audit destinam-se a registar o que foi alterado, quando foi alterado, quem alterou e que valores fizeram parte deste processo.

Exemplo de trigger Insert - Código Postal

```
DELIMITER //

-- TABELA: codpostal
DROP TRIGGER IF EXISTS audit_codpostal_insert//
CREATE TRIGGER audit_codpostal_insert
AFTER INSERT ON codpostal
FOR EACH ROW
BEGIN
    INSERT INTO audit_log(tabela_nome, operacao, registro_id, dados_novos)
    VALUES ('codpostal','INSERT', NULL,
        JSON_OBJECT('codpostal', NEW.codpostal, 'localidade', NEW.localidade, 'cidade', NEW.cidade)
    );
END//
```

```
-- TABELA: audit_log
CREATE TABLE IF NOT EXISTS audit_log (
    id INT AUTO_INCREMENT PRIMARY KEY,
    tabela_nome VARCHAR(50) NOT NULL,
    operacao ENUM('INSERT','UPDATE','DELETE') NOT NULL,
    registro_id INT,
    dados_anteriores JSON,
    dados_novos JSON,
    utilizador VARCHAR(50),
    data_hora TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    INDEX idx_tabela (tabela_nome),
    INDEX idx_data (data_hora)
);
```

Tabelas Auditadas

Exemplo de um trigger de update e de delete

```
DROP TRIGGER IF EXISTS audit_codpostal_update//
CREATE TRIGGER audit_codpostal_update
AFTER UPDATE ON codpostal
FOR EACH ROW
BEGIN
    INSERT INTO audit_log(tabela_nome, operacao, registro_id, dados_anteriores, dados_novos)
    VALUES ('codpostal','UPDATE', NULL,
        JSON_OBJECT('codpostal', OLD.codpostal, 'localidade', OLD.localidade, 'cidade', OLD.cidade),
        JSON_OBJECT('codpostal', NEW.codpostal, 'localidade', NEW.localidade, 'cidade', NEW.cidade)
    );
END//

DROP TRIGGER IF EXISTS audit_codpostal_delete//
CREATE TRIGGER audit_codpostal_delete
AFTER DELETE ON codpostal
FOR EACH ROW
BEGIN
    INSERT INTO audit_log(tabela_nome, operacao, registro_id, dados_anteriores)
    VALUES ('codpostal','DELETE', NULL,
        JSON_OBJECT('codpostal', OLD.codpostal, 'localidade', OLD.localidade, 'cidade', OLD.cidade)
    );
END//
```

Sempre que um registo é atualizado ou apagado, o trigger guarda no **audit_log** a operação realizada e os dados antes e depois da alteração através das variáveis **OLD** e **NEW** que contêm os registos antes e depois da operação.

Tabelas Auditadas

Todas as tabelas são auditadas e foi criada uma view para facilitar a consulta:

```sql
CREATE OR REPLACE VIEW view_audit_log AS
SELECT
    id,
    tabela_nome,
    operacao,
    registro_id,
    JSON_PRETTY(dados_anteriores) AS dados_anteriores_formatados,
    JSON_PRETTY(dados_novos) AS dados_novos_formatados,
    utilizador,
    data_hora
FROM audit_log
ORDER BY data_hora DESC;
```

Triggers adicionais

Também foram criados triggers para a complementar a base de dados como:

- Calcular total do pedido após inserir, remover ou atualizar um prato.
- Atualizar estado do entregador quando atribuída uma entrega ou a mesma é finalizada.

```
-- Trigger: Atualizar estado do entregador quando atribuída uma entrega
DROP TRIGGER IF EXISTS atualizar_entregador_entrega$$
CREATE TRIGGER atualizar_entregador_entrega
AFTER INSERT ON entregas
FOR EACH ROW
BEGIN
    UPDATE entregadores SET estado = 'ocupado' WHERE id = NEW.entregador_id;
END$$
```

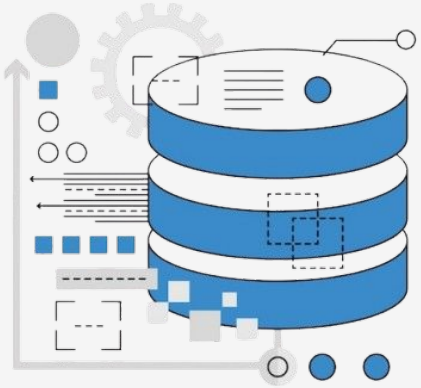
Configuração do Master

```
#Server ID
server-id=1

# Enable GTID mode
gtid_strict_mode=1
gtid_domain_id=1

# Binary logging (gera os ficheiros bin)
log-bin=mysql-bin
binlog-format=ROW

#Grava no disco e só replica a base de dados food_delivery
sync-binlog=1
binlog-do-db=food_delivery
```



Configuração do Master

Além disto, os ficheiros que preenchem a base de dados são executados na criação do container do Master

Estes ficheiros encontram se numa pasta chamada scripts e são copiadas para uma pasta init dentro do container sendo executados por ordem alfabética:

- 01.schema.sql - cria as tabelas sql
- 02.seed.sql - preenche as tabelas com dados
- 03.audit.sql - cria a tabela e os triggers de audit

Configuração da replicação

Aquando da inicialização do container das réplicas, os ficheiros sql na pasta scripts são copiados para uma pasta init e são executados.

Nas réplicas, a configuração da replicação é executada pelo script 01.setup_replication

O comando principal que começa a replicação é START SLAVE

```
# Configura replicação usando GTID (estava a dar erro na iniciação porque a réplica tinha GTIDs antigos)
mariadb -uroot -p${MYSQL_ROOT_PASSWORD} <<EOF
-- Para replicação se estiver a rodar
STOP SLAVE;

-- Reseta replicação
RESET SLAVE ALL;

-- Reseta binlogs da réplica para começar do zero
RESET MASTER;

-- Usa slave_pos que sincroniza automaticamente com o master
CHANGE MASTER TO
    MASTER_HOST='mariadb-master',
    MASTER_USER='repl',
    MASTER_PASSWORD='repl_password',
    MASTER_USE_GTID=slave_pos;

-- Inicia replicação
START SLAVE;

SELECT SLEEP(2);

-- Verifica status
SHOW SLAVE STATUS\G
EOF
```

Configuração do maxscale

Neste ficheiro que também é executado na inicialização do container do maxscale, são definidos os servidores e as suas respetivas portas assim como a do próprio maxscale.

É criado um monitor que verifica os servidores a cada 2s e são habilitadas as funções de *auto_failover* e *auto_rejoin*.

Também criado um router de *readwritesplit* que separa as queries de leituras para as réplicas e as de escrita para o master.

Por fim cria um listener na porta 4006 no maxscale e encaminha tudo para o serviço *readwritesplit* (router) onde são distribuidas as queries.

É tambem disponibilizado um dashboard na porta 8989.

```ini
[maxscale]
threads=auto
admin_host=0.0.0.0
admin_port=8989

# Definir servidores
[master-server]
type=server
address=mariadb-master
port=3306

[replica1-server]
type=server
address=mariadb-replica1
port=3306

[replica2-server]
type=server
address=mariadb-replica2
port=3306

# Monitor (vigia os servidores)
[MariaDB-Monitor]
type=monitor
module=mariadbmon
servers=master-server,replica1-server,replica2-server
user=root
password=root_password
monitor_interval=2000ms
auto_failover=true      #  Promove réplica a Master se Master cair
auto_rejoin=true        #  Reconecta servidores que voltam online
enforce_read_only_slaves=true

# Service (distribui queries)
[Read-Write-Service]
type=service
router=readwritesplit   #  Router que separa leituras e escritas
servers=master-server,replica1-server,replica2-server
user=root
password=root_password
slave_selection_criteria=LEAST_CURRENT_OPERATIONS

# Listener (porta de entrada)
[Read-Write-Listener]
type=listener
service=Read-Write-Service
protocol=MariaDBClient
port=4006
address=0.0.0.0
```

Funcionamento da replicação

1. **Escrita no Master**
   Quando uma operação de escrita é executada no Master, o MariaDB regista essa operação no binary log (binlog), num formato binário compacto.

O binary log contém todos os eventos que modificam dados, incluindo:
– **Statements** (instruções SQL), no formato STATEMENT
– **Alterações linha a linha**, no formato ROW (utilizado neste projeto)
– **Metadata**, ou seja, informações adicionais sobre a operação

2. **GTID (Global Transaction ID)**
   Cada transação que é commitada recebe um GTID único, por exemplo: 1-1-197.
   O GTID permite rastrear transações de forma única em toda a topologia de replicação.
   Facilita processos como failover e auto-rejoin, porque as réplicas sabem exatamente qual foi a última transação aplicada.

Funcionamento da replicação

3. **Ligação das Réplicas**
   Cada réplica mantém uma thread I/O que se liga ao Master.
   Esta thread I/O solicita eventos do binary log ao Master, indicando o GTID a partir do qual quer receber os eventos.
   O Master envia os eventos binários através da ligação de rede.

4. **Relay Log**
   A réplica recebe os eventos enviados pelo Master e escreve-os no relay log.
   O relay log é local à réplica e funciona como um buffer temporário dos eventos recebidos.

Funcionamento da replicação

5. **Aplicação nas Réplicas**
   Uma thread SQL na réplica lê os eventos do relay log.
   Essa thread aplica os eventos sequencialmente na base de dados local.
   Os eventos são aplicados pela mesma ordem em que foram commitados no Master.

6. **Sincronização**
   A réplica mantém o registo do último GTID que foi aplicado com sucesso.
   O atraso (lag) é medido pela diferença entre o GTID atual do Master e o GTID da réplica.

Testes ACID

Implementamos testes acid em Python que validam cada uma das propriedades ACID com 4 cenários de teste por propriedade, totalizando 16 testes.

Exemplo de um teste de **Atomicidade**:
Se uma parte falhar, todas as operações anteriores devem ser anuladas (**rollback**) - teste 2
Se tudo correr bem, todas as alterações são confirmadas (**commit**) - neste caso.

- Cria um código postal (se não existir)
- Obtém ou cria uma categoria
- Cria um restaurante
- Cria um prato associado ao restaurante

```python
def test_atomicity_1():
    try:
        cursor = conn.cursor()

        # Iniciar transação
        conn.start_transaction()

        # Criar codpostal se não existir
        cursor.execute("""
            INSERT IGNORE INTO codpostal (codpostal, localidade, cidade)
            VALUES ('1000-001', 'Lisboa', 'Lisboa')
        """)

        # Obter ou criar categoria
        cursor.execute("SELECT id FROM categorias_pratos WHERE nome = 'Teste' LIMIT 1")
        cat_result = cursor.fetchone()
        if cat_result:
            categoria_id = cat_result[0]
        else:
            cursor.execute("INSERT INTO categorias_pratos (nome) VALUES ('Teste')")
            categoria_id = cursor.lastrowid

        # Criar restaurante
        cursor.execute("""
            INSERT INTO restaurantes (nome, morada, codpostal, email, telefone, especialidade_id)
            VALUES ('Teste Atomic 1', 'Rua Teste', '1000-001', 'teste.atomic1@test.pt', '999999999', %s)
        """, (categoria_id,))
        restaurante_id = cursor.lastrowid

        # Criar prato associado
        cursor.execute("""
            INSERT INTO pratos (restaurante_id, categoria_id, nome, preco)
            VALUES (%s, %s, 'Prato Teste', 10.50)
        """, (restaurante_id, categoria_id))

        # Commit
        conn.commit()

        # Verificar se ambos foram criados
        cursor.execute("SELECT COUNT(*) FROM restaurantes WHERE id = %s", (restaurante_id,))
        restaurante_exists = cursor.fetchone()[0] > 0

        cursor.execute("SELECT COUNT(*) FROM pratos WHERE restaurante_id = %s", (restaurante_id,))
        prato_exists = cursor.fetchone()[0] > 0

        result = restaurante_exists and prato_exists
        print_test("Atomicidade 1", result,
                   f"Restaurante e prato criados: {restaurante_exists} e {prato_exists}")

        # Limpeza
        cursor.execute("DELETE FROM pratos WHERE restaurante_id = %s", (restaurante_id,))
        cursor.execute("DELETE FROM restaurantes WHERE id = %s", (restaurante_id,))
        conn.commit()

        return result
    except Error as e:
        conn.rollback()
        print_test("Atomicidade 1", False, f"Erro: {e}")
        return False
```

Testes ACID

Exemplo de um teste de **Consistência**:
Garante que a base de dados passa sempre de um estado válido para outro estado válido, respeitando todas as regras, restrições e integridade dos dados.

- Cria uma categoria de pato
- Tenta criar outra categoria com o mesmo nome

É esperado que a segunda categoria não seja aceite porque a variável nome é *unique*.

```python
def test_consistency_2():
    try:
        cursor = conn.cursor()

        # Criar primeira categoria com nome único
        categoria_nome = "Categoria Teste UNIQUE"
        cursor.execute("""
            INSERT INTO categorias_patos (nome) VALUES (%s)
        """, (categoria_nome,))
        categoria_id = cursor.lastrowid
        conn.commit()

        # Tentar criar segunda categoria com o mesmo nome (deve falhar - UNIQUE constraint)
        try:
            cursor.execute("""
                INSERT INTO categorias_patos (nome) VALUES (%s)
            """, (categoria_nome,))
            conn.commit()
            result = False
            print_test("Consistência 2", False, "Constraint UNIQUE não foi respeitada!")

            # Limpeza
            cursor.execute("DELETE FROM categorias_patos WHERE nome = %s", (categoria_nome,))
            conn.commit()
        except Error as e:
            # Erro esperado - constraint UNIQUE funcionou
            error_code = e.errno if hasattr(e, 'errno') else None
            if error_code == 1062:  # Duplicate entry
                result = True
                print_test("Consistência 2", True, f"Constraint UNIQUE respeitada: {str(e)[:50]}")
            else:
                result = False
                print_test("Consistência 2", False, f"Erro inesperado: {str(e)[:50]}")

        # Limpeza
        cursor.execute("DELETE FROM categorias_patos WHERE id = %s", (categoria_id,))
        conn.commit()

        return result
    except Error as e:
        conn.rollback()
        print_test("Consistência 2", False, f"Erro: {e}")
        return False
    finally:
        cursor.close()
        conn.close()

def test_consistency_3():
    """Teste 3: Valores devem manter consistência após update"""
    print(f"\n{Colors.BLUE}=== Teste Consistência 3: Consistência após Update ==={Colors.RESET}")
    conn = get_connection()
    if not conn:
        return False

    try:
        cursor = conn.cursor()
```

Testes ACID

Exemplo de um teste de **Durabilidade**:
Os dados commitados permanecem persistentes na base de dados mesmo após uma desconexão e nova reconexão ao sistema.

- Cria um código postal (se não existir)
- Obtém ou cria uma categoria
- Reconecta à base de dados
- Verifica a persistência dos dados

É esperado que os dados continuem depois de fechar a ligação á base de dados.

```python
def test_durability_1():
    """Teste 1: Dados commitados persistem após reconexão"""
    print(f"\n{Colors.BLUE}=== Teste Durabilidade 1: Persistência após Commit ==={Colors.RESET}")
    conn = get_connection()
    if not conn:
        return False

    try:
        cursor = conn.cursor()

        # Criar codpostal se não existir
        cursor.execute("""
            INSERT IGNORE INTO codpostal (codpostal, localidade, cidade)
            VALUES ('1000-009', 'Lisboa', 'Lisboa')
        """)

        # Obter ou criar categoria
        cursor.execute("SELECT id FROM categorias_pratos WHERE nome = 'Teste' LIMIT 1")
        cat_result = cursor.fetchone()
        if cat_result:
            categoria_id = cat_result[0]
        else:
            cursor.execute("INSERT INTO categorias_pratos (nome) VALUES ('Teste')")
            categoria_id = cursor.lastrowid

        # Criar e commitar
        cursor.execute("""
            INSERT INTO restaurantes (nome, morada, codpostal, email, telefone, especialidade_id)
            VALUES ('Durability Test 1', 'Rua Teste', '1000-009', 'durability.test1@test.pt', '999999999', %s)
        """, (categoria_id,))
        restaurante_id = cursor.lastrowid
        conn.commit()

        # Fechar conexão
        cursor.close()
        conn.close()

        # Reconectar e verificar
        conn2 = get_connection()
        cursor2 = conn2.cursor()
        cursor2.execute("SELECT COUNT(*) FROM restaurantes WHERE id = %s", (restaurante_id,))
        exists = cursor2.fetchone()[0] > 0

        result = exists
        print_test("Durabilidade 1", result,
                   f"Dados persistem após reconexão: {exists}")

        # Limpeza
        cursor2.execute("DELETE FROM restaurantes WHERE id = %s", (restaurante_id,))
        conn2.commit()

        cursor2.close()
        conn2.close()
        return result
    except Error as e:
        conn.rollback()
        print_test("Durabilidade 1", False, f"Erro: {e}")
        return False
```

Conclusão

Esta arquitetura garante que o sistema continue a funcionar mesmo em caso de falhas e permite escalar horizontalmente conforme a carga necessária ou desejada.

Os desafios encontrados durante o desenvolvimento foram superados através de pesquisa, diferentes testes e debugging, e da sua respetiva documentação.