Base de Dados

Carlos Costa
Joaquim Sousa Pinto
Sérgio Matos

Base de Dados de uma Perfumaria:

Perfumaria Orquídea

Pedro Bastos, 93150
Hugo Almeida, 93195

DETI
Universidade de Aveiro
12-06-2020

Conteúdo

1 Introdução

No âmbito da disciplina de Base de Dados, foi pedido o desenvolvimento de um projeto que envolvesse a elaboração de um sistema de Base de Dados.

Como tema do trabalho, foi escolhido um problema real, de uma Perfumaria que tem também um gabinete de estética e necessitava de um sistema deste tipo. A construção deste sistema foi realizada em SQL e com uma aplicação *Windows Forms* em *Visual Studio* com *C#*.

2 Análise de Requisitos

Esta solução de base de dados está idealizada para ser implementada numa plataforma online de vendas de uma perfumaria que também tem um salão de estética. Após o processo de comunicação com o cliente da solução de base de dados, familiar de um membro do grupo, foi feita esta análise de requisitos.

Informação associada ao "problema" do mundo:

Existem **Utilizadores** na plataforma que podem ser do tipo Cliente ou Funcionário. Ambos têm os seguintes dados guardados, sendo que os a negrito são obrigatórios:

- **Sexo**
- **Data de Nascimento**
- Morada
  - Endereço
  - Apt./Suite/Andar
  - Localidade
  - País
  - Código-Postal
- **Nome**
  - **Primeiro Nome**
  - **Último Nome**
- **Email**
- **Password**
- Telemóvel
- Número Contribuinte
- Foto

Existem **Produtos** na plataforma que podem ser do tipo Perfume ou Cosmética. Ambos têm os seguintes dados guardados, sendo que os a negrito são obrigatórios:

- **Imagem**

- **Identificador**
- Descrição
- **Marca**
- **Nome**
- Linha
- Tamanho
- Categoria do utilizador - Homem/Senhora/Criança (Destinatário)
- **Categoria do Produto - After-Shave/Rosto**
- **Preço**
- Família Olfativa

Existem **Compras** na plataforma que podem ser do tipo Presencial ou Online. Ambos têm os seguintes dados guardados, sendo que os a negrito são obrigatórios:

- **Número de Compra**
- **Número de contribuinte do cliente**
- **Data da compra**
- **Tipo de Pagamento**

Existem **Cupões** na plataforma que podem ter os seguintes dados guardados, sendo que os a negrito são obrigatórios:

- **Data de início da validade**
- **Data de final da validade**
- **Identificador**
- **Pontos que atribui**

Existem **Serviços** na plataforma que podem ter os seguintes dados guardados, sendo que os a negrito são obrigatórios:

- **Tipo**
- **Identificador**
- **Preço**
- **Duração**

Existem utilizadores do tipo **Funcionários** na plataforma que podem ter os seguintes dados guardados, sendo que os a negrito são obrigatórios:

- **Indicação se é Administrador**
- **Salário**

Existem utilizadores do tipo **Clientes** na plataforma podem ter os seguintes dados guardados, sendo que os a negrito são obrigatórios:

- **Pontos**
- Forma de Pagamento
- **Indicação de subscrição de Newsletter**

Existem produtos do tipo **Cosmética** na plataforma podem ter os seguintes dados guardados, sendo que os a negrito são obrigatórios:

- Tipo do produto - Desmaquilhantes/Tónicos...

Existem Compras do tipo **Online** na plataforma podem ter os seguintes dados guardados, sendo que os a negrito são obrigatórios:

- Rating da encomenda por estrelas
- **Escolha se quer ser embrulhado como presente**
- **Morada**
  - **Endereço**
  - **Apt./Suite/Andar**
  - **Localidade**
  - **País**
  - **Código-Postal**
- **Telemóvel**
- Observações
- Número de rastreio

Um cliente pode adicionar produtos como favoritos.

Um cliente pode usar um cupão uma única vez para adicionar pontos à sua conta.

Um cliente pode realizar várias compras quer estas sejam presenciais ou online. Por predefinição, os dados como Nº Contribuinte, Tipo de pagamento, Telemóvel (apenas compra online), etc, da compra são os dados que o utilizador tem guardado na sua conta, podendo ser guardados dados diferentes naquela compra em questão.

Nos dados de um cliente, tanto a morada como o tipo de pagamento, telemóvel e número de contribuinte são opcionais sendo que ao realizar uma compra, muitos destes são definidos para a compra em questão e poderão (ou não) ser guardados na tabela do utilizador.

Um produto pode ter uma promoção associada com o respectivo nome e percentagem de desconto.

Ao realizar uma compra, esta compra vai ser composta por produtos com as respetivas unidades adquiridas. Na compra podem ser descontados os pontos do cliente e/ou acumulados.

Um cliente pode marcar um serviço, ficando a data dessa marcação guardada.

A duração do serviço depende do funcionário que o realiza. Uma compra presencial tem obrigatoriamente um funcionário a supervisionar.

3 DER

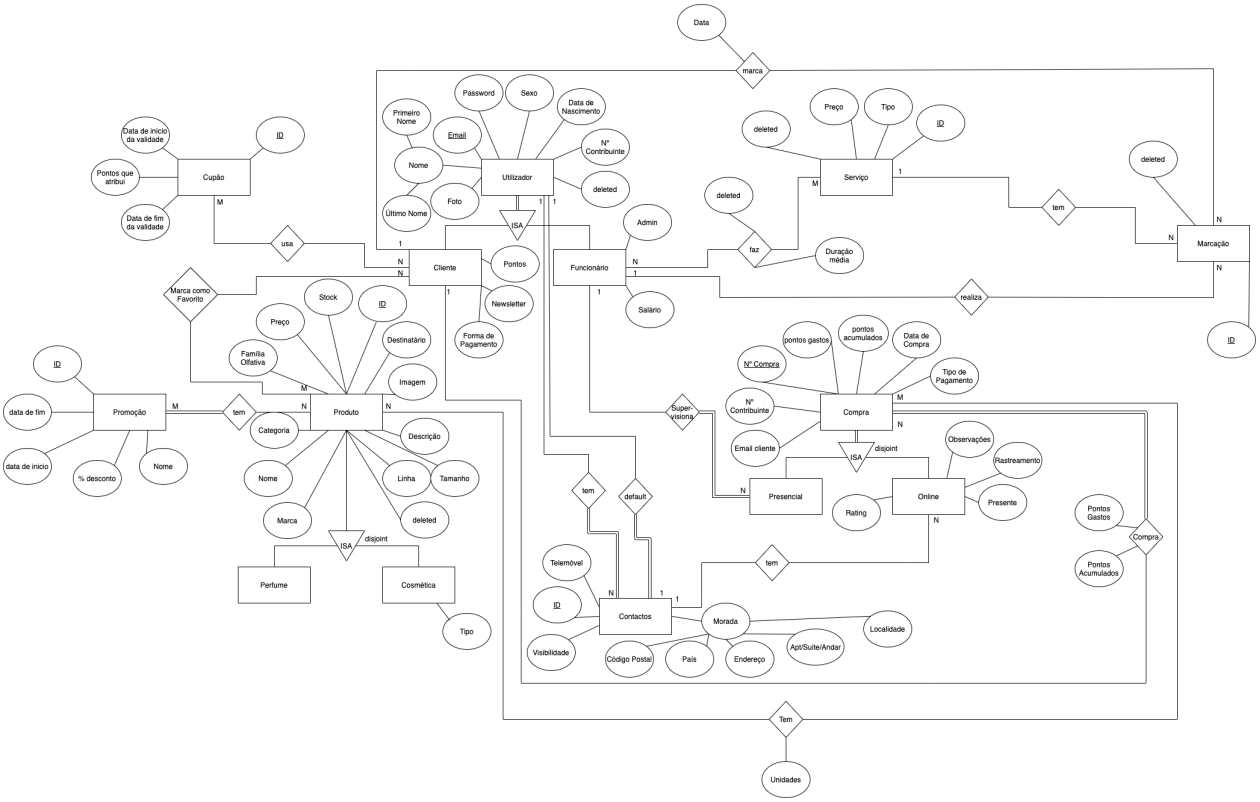

Figura 1: Diagrama Entidade/Relação da proposta de base de dados apresentada pelo grupo

4 Esquema Relacional da BD

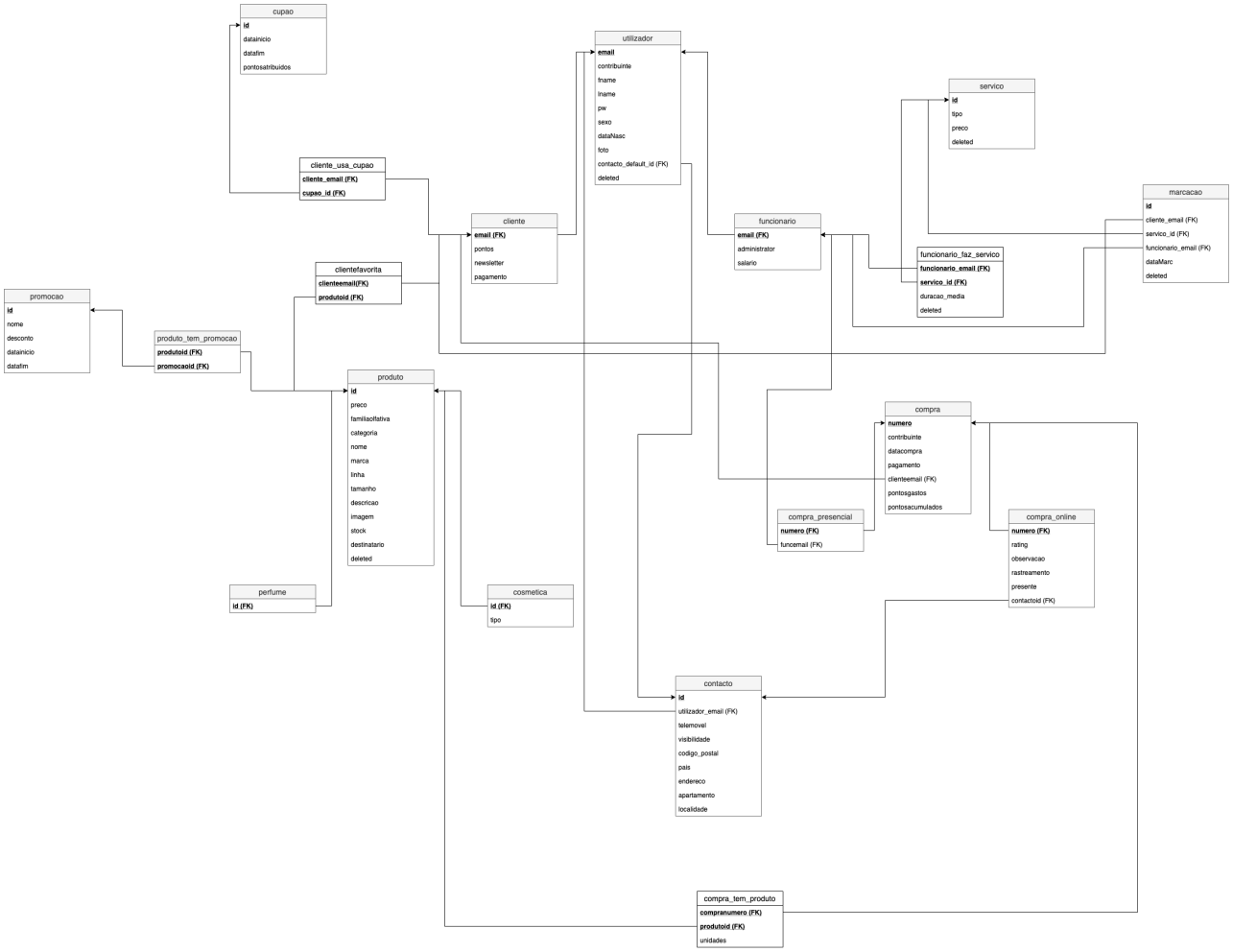

Figura 2: Modelo Entidade/Relação da proposta de base de dados apresentada pelo grupo

5 SQL DDL

Foi criada uma Base de dados local com o nome de *Perfumaria* e posteriormente criado o *Schema perf.* Assim, as tabelas têm como nome *Perfumaria.perf.[TableName]*.

```
1   IF DB_ID('Perfumaria') IS NULL
2   CREATE DATABASE Perfumaria;
3   GO
4
5   USE Perfumaria;
6   GO
7
8   IF NOT EXISTS (SELECT * FROM sys.schemas WHERE name = 'perf')
9   BEGIN
10  EXEC('CREATE SCHEMA perf')
11  END
```

Figura 3: Criação da Base de dados e *Schema*

A criação das tabelas é sempre feita da mesma forma, como demonstrado na Figura 4. A tabela *Perfumaria.perf.utilizador* tem a particularidade de, por questões de segurança, apenas guardar o valor do *Hash* da password que o utilizador coloca (criado através de um *Stored Procedure*). Assim, a password nunca chega a ser guardada na BD, apenas é guardado o *Hash*.

```
259  CREATE TABLE Perfumaria.perf.utilizador
260  (
261      email VARCHAR(255) NOT NULL,
262      contribuinte CHAR(9) NOT NULL,
263      fname VARCHAR(20) NOT NULL,
264      lname VARCHAR(20) NOT NULL,
265      pw BINARY(64) NOT NULL,
266      sexo BIT NOT NULL,
267      dataNasc DATE NOT NULL,
268      foto VARCHAR(100) NOT NULL,
269      contacto_default_id INT,
270      deleted BIT NOT NULL DEFAULT 0,
271      CONSTRAINT utilizador_pk PRIMARY KEY (email)
272
273  );
274  GO
```

Figura 4: Tabela Perfumaria.perf.utilizador

As *Foreign Keys* são adicionadas no fim do *script* para evitar conflitos, sendo que existem verificações iniciais para, se necessário, ser feito o *Drop* de *Constraints*.

6 SQL DML

Todos os comandos de *SQL DML* foram executados dentro de *Stored Procedures, UDFs e Triggers* de maneira a criar uma camada de abstração entre a interface e a Base de Dados, aumentando a segurança com uma série de verificações, evitando *SQL Injections* e permitindo o encapsulamento da mesma.

7 Normalização

Tendo em conta o objetivo do processo de desenho de uma base de dados relacional relativamente a minimizar a redundância dos dados, foram feitos diversos testes para verificar que formas normais esta respeitava (processo de normalização).

Após estas verificações, foi concluído que o modelo de dados se encontrava na Terceira Forma Normal (*3FN*), respeitando também as formas normais anteriores. Com isto em consideração, concluiu-se que as relações estavam normalizadas, não necessitando qualquer tipo de alterações.

8 Índices

Ao analisar o tipo de *queries* feitas pela aplicação e qual a sua frequência, foi concluído que faria sentido utilizar um índice composto *Non-clustered* para melhorar a performance das pesquisas e filtros da visualização dos produtos. Dito isto, foi criado o **índice composto *Non-clustered* (preco, marca, nome, categoria)** da tabela **produto**, permitindo melhoria da *performance* a *queries* que incluam filtros ou pesquisas com:

- Preço
- Preço e Marca
- Preço, Marca e Categoria
- Preço, Marca, Nome e Categoria

É expectável a utilização destes filtros, especialmente com preço, pois os produtos são carregados inicialmente na aplicação ordenados pelo preço. Futuramente com a utilização da aplicação por utilizadores, faria sentido verificar se o restante das presunções se mantêm. No entanto, nos testes que foram feitos à performance das *queries* antes e depois da utilização do índice, ao utilizá-lo, o *Estimated Subtree Cost* foi inferior.

Por outro lado, não foram utilizados outros índices pelo simples facto que a maior parte das restantes *queries* lidam com as entidades através de pesquisa com as *Primary Keys* que, por si só, já são por *default* um índice *Clustered*.

9 Triggers

Foram criados *Triggers After* para automaticamente serem executadas ações e algumas verificações após determinadas chamadas *DML*, com o objetivo de manter a integridade dos dados.

Um dos quatro exemplos utilizados é a adição de pontos à conta do cliente após a utilização de um cupão (validado anteriormente através de um *Stored Procedure*):

```
1   DROP TRIGGER perf.useCuponTrigger;
2   GO
3
4   CREATE TRIGGER perf.useCuponTrigger ON perf.[cliente_usa_cupao]
5   AFTER INSERT
6   AS
7   BEGIN
8       SET NOCOUNT ON;
9       DECLARE @cupao AS CHAR(10);
10      DECLARE @email AS VARCHAR(255);
11      DECLARE @pontos AS INT;
12      SELECT @cupao = cupao_id, @email = cliente_email FROM inserted;
13      SELECT @pontos = pontos_atribuidos FROM Perfumaria.perf.cupao WHERE id = @cupao;
14      BEGIN TRY
15          UPDATE perf.cliente
16          SET pontos += @pontos
17          WHERE email = @email
18      END TRY
19      BEGIN CATCH
20          RAISERROR ('Não foi possível atribuir os pontos', 16, 1);
21          ROLLBACK TRAN
22      END CATCH
23
24  END
25  GO
```

Figura 5: *Trigger After Insert* de um cupão na tabela *cliente_usa_cupao*

Os outros 3 *Triggers After, buyProduct, changeProduct* e *createContact* verificam se os produtos necessitam de passar para eliminados (através do atributo *deleted*) consoante o *stock*, ou se o utilizador passa a ter um contacto como *default*, após a sua criação, se ainda não tiver nenhum.

Foi escolhido o uso de *Stored Procedures* ao invés de *Triggers Instead Of* para verificações antes da execução de chamadas *DML* pois a aplicação utiliza-os sempre para fazer alterações de dados e nunca foi necessária a substituição de uma chamada por outra.

10 Stored Procedures

Com foi referido anteriormente, grande parte das chamadas à Base de Dados são feitas através de *Stored Procedures*, criando assim a camada de abstração. Resumindo, foram criadas *SPs* para adicionar elementos às tabelas, como por exemplo *addContact, addCupon, addFuncService, addMarc*, etc.

```
DROP PROCEDURE perf.addMarc;
GO


CREATE PROCEDURE perf.addMarc
    @cliente_email VARCHAR(255),
    @servico_id INT,
    @funcionario_email VARCHAR(255),
    @dataMarc SMALLDATETIME,
    @responseMessage VARCHAR(250) = 'Erro! Tente noutra hora.' OUTPUT
AS
BEGIN
    SET NOCOUNT ON
    BEGIN TRY
        DECLARE @duracao INT
        SELECT @duracao = duracao_media FROM Perfumaria.perf.funcionario_faz_servico WHERE funcionario_email=@funcionario_email AND deleted = 0 AND  servico_id = @servico_id
            IF (@duracao IS NOT NULL AND @dataMarc > GETDATE())
            BEGIN
                IF (EXISTS(SELECT 1 FROM Perfumaria.perf.marcacao WHERE (dataMarc BETWEEN @dataMarc AND DATEADD(mi, @duracao, @dataMarc)) AND funcionario_email=@funcionario_email AND deleted=0)
                OR EXISTS(SELECT 1 FROM Perfumaria.perf.marcacao WHERE dataMarc BETWEEN @dataMarc AND DATEADD(mi, @duracao, @dataMarc) AND cliente_email=@cliente_email AND deleted=0))
                SET @responseMessage = 'Hora não disponível!'
                ELSE
                BEGIN
                INSERT INTO Perfumaria.perf.marcacao
                (cliente_email, servico_id, funcionario_email, dataMarc)
                VALUES(@cliente_email, @servico_id, @funcionario_email, @dataMarc)

                SET @responseMessage='Marcação efetuado com sucesso!'
                END
            END
    END TRY
    BEGIN CATCH
        SET @responseMessage='Failure'
    END CATCH


END
GO
```

Figura 6: Exemplo de uma *Stored Procedure* de adição

Foram também criadas *SPs* para alterar valores de uma tabela, como por exemplo *changeDefaultContact, changeProduct, updateFunc, updateMarc*, etc.

```
DROP PROCEDURE perf.updateMarc;
GO

CREATE PROCEDURE perf.updateMarc
    @cliente_email VARCHAR(255),
    @idMarc INT,
    @funcionario_email VARCHAR(255),
    @dataMarc DATETIME = NULL,
    @responseMessage NVARCHAR(250) = 'Erro! Tente noutra hora.' OUTPUT
AS
BEGIN
    BEGIN TRY
        SET NOCOUNT ON
            IF EXISTS (SELECT email=@funcionario_email from Perfumaria.perf.funcionario)
            BEGIN
                IF (@dataMarc IS NOT NULL)
                BEGIN
                    BEGIN TRY
                        DECLARE @duracao INT
                        SELECT @duracao=duracao_media FROM Perfumaria.perf.funcionario_faz_servico JOIN Perfumaria.perf.marcacao ON funcionario_faz_servico.servico_id=marcacao.servico_id
                                WHERE marcacao.funcionario_email=@funcionario_email  AND marcacao.deleted = 0 AND marcacao.id=@idMarc AND funcionario_faz_servico.funcionario_email=@funcionario_email
                            IF (@duracao IS NOT NULL AND @dataMarc > GETDATE())
                            BEGIN
                                IF (EXISTS(SELECT 1 FROM Perfumaria.perf.marcacao WHERE (dataMarc BETWEEN @dataMarc AND DATEADD(mi, @duracao, @dataMarc)) AND @funcionario_email=funcionario_email)
                                OR EXISTS(SELECT 1 FROM Perfumaria.perf.marcacao WHERE dataMarc BETWEEN @dataMarc AND DATEADD(mi, @duracao, dataMarc) AND cliente_email=@cliente_email))
                                    SET @responseMessage = 'Hora não disponível!'
                                ELSE
                                BEGIN
                                    UPDATE Perfumaria.perf.marcacao
                                    SET dataMarc=@dataMarc
                                    WHERE id=@idMarc
                                    SET @responseMessage='Marcação efetuado com sucesso!'
                                END
                            END
                    END TRY
                    BEGIN CATCH
                        SET @responseMessage='Failure'
                    END CATCH
                END
            END
    END TRY
    BEGIN CATCH
        SET @responseMessage='Failed'
    END CATCH
END
GO
```

Figura 7: Exemplo de uma *Stored Procedure* de *update*

Existem também alguns *gets* mais complexos, como por exemplo o *getDetailsFromBuy*, *getDetailsFromSell*, *getProductFilters*, etc.

```
DROP PROCEDURE perf.getProductFilters;
GO

CREATE PROCEDURE perf.getProductFilters
    @deleted BIT = NULL,
    @nome VARCHAR(30) = NULL,
    @marca VARCHAR(30) = NULL,
    @categoria VARCHAR(30) = NULL,
    @destinatario VARCHAR(10) = NULL,
    @orderby VARCHAR(50) = NULL,
    @ordem VARCHAR(30) = NULL
AS
BEGIN
        SET NOCOUNT ON
        IF (@ordem = 'Ascendente')
        BEGIN
            SELECT id, preco, familiaolfativa, categoria, nome, marca, linha, tamanho, descricao, imagem, stock, destinatario, deleted
            FROM Perfumaria.perf.produto
            WHERE stock > 0 AND
                  deleted = 0 AND
                  nome LIKE ('%'+ISNULL(@nome, nome)+'%') AND
                  marca = ISNULL(@marca,marca) AND
                  categoria = ISNULL(@categoria,categoria)
            ORDER BY CASE WHEN @orderby='Nome' THEN nome END,
                     CASE WHEN @orderby='Marca' THEN marca END,
                     CASE WHEN @orderby='Categoria' THEN categoria END,
                     CASE WHEN @orderby='Preço' THEN preco END
        END

        ELSE IF (@ordem = 'Descendente')
        BEGIN
            SELECT id, preco, familiaolfativa, categoria, nome, marca, linha, tamanho, descricao, imagem, stock, destinatario, deleted
            FROM Perfumaria.perf.produto
            WHERE stock > 0 AND
                  deleted = 0 AND
                  nome LIKE ('%'+ISNULL(@nome, nome)+'%') AND
                  marca = ISNULL(@marca,marca) AND
                  categoria = ISNULL(@categoria,categoria)
            ORDER BY CASE WHEN @orderby='Nome' THEN nome END DESC,
                     CASE WHEN @orderby='Marca' THEN marca END DESC,
                     CASE WHEN @orderby='Categoria' THEN categoria END DESC,
                     CASE WHEN @orderby='Preço' THEN preco END DESC
        END

END
GO
```

Figura 8: Exemplo de uma *Stored Procedure* de um *get*

Finalmente, existem algumas *SPs* para ações mais específicas. Dentro destas, as mais importantes são *login*, *registerFunc* e *registerClient*, que permitem o sistema de *login* inicial da nossa aplicação.

```
DROP PROCEDURE perf.RegisterFunc
GO


CREATE PROCEDURE perf.RegisterFunc
    @email VARCHAR(255),
    @password VARCHAR(25),
    @contribuinte CHAR(9),
    @fname VARCHAR(20),
    @lname VARCHAR(20),
    @sexo BIT,
    @dataNasc DATE,
    @foto VARCHAR(100),
    @salario INT,
    @administrator TINYINT,
    @responseMessage NVARCHAR(250) OUTPUT
AS
BEGIN
    BEGIN TRANSACTION
    SET NOCOUNT ON

    BEGIN TRY

        INSERT INTO Perfumaria.perf.utilizador
        (email, contribuinte, fname, lname, pw, sexo, dataNasc, foto)
    VALUES(@email, @contribuinte, @fname, @lname, HASHBYTES('SHA2_512', @password), @sexo, @dataNasc, @foto)

    INSERT INTO Perfumaria.perf.funcionario
        (email, administrator, salario)
    VALUES(@email, @administrator, @salario)

    SET @responseMessage='Success'
    COMMIT TRANSACTION
    END TRY
    BEGIN CATCH
        SET @responseMessage=ERROR_MESSAGE()
        ROLLBACK
    END CATCH

END
GO
```

Figura 9: *Stored Procedure* de Registo de um funcionário

11 UDF

Foram também utilizadas *UDFs* para a execução de *queries* mais simples, isto é, para apenas ir buscar valores de certas tabelas, tendo em conta as suas restrições. Preferiram-se *UDFs* a *Views*, não só devido ao seu maior desempenho, mas também por permitir passar parâmetros. Usaram-se *UDFs Inline Table-Valued*, como por exemplo *clientBuyHistory*, *funcFutureMarc*, etc.

```
DROP FUNCTION perf.funcFutureMarc;
GO

CREATE FUNCTION perf.funcFutureMarc (@email VARCHAR(255)) RETURNS TABLE
AS
    RETURN (SELECT cliente_email,dataMarc, tipo, preco, fname, lname, marcacao.id
                   FROM ((Perfumaria.perf.marcacao JOIN Perfumaria.perf.servico ON servico_id=servico.id) JOIN Perfumaria.perf.utilizador ON cliente_email = email)
                   WHERE (funcionario_email=@email AND DATEDIFF(mi, GETDATE(), dataMarc) > 0)
                   ORDER BY dataMarc ASC OFFSET 0 ROWS)
GO
```

Figura 10: Exemplo de *UDF Inline Table-Valued*

Além das *Inline*, utilizámos também as *Multi-Statement Table-Valued* para *queries* simples mas que necessitavam de verificações iniciais, como por exemplo *getAllFuncs*, *getAllProducts*, etc.

```
DROP FUNCTION perf.getAllFuncs;
GO

CREATE FUNCTION perf.getAllFuncs ( @emailFunc VARCHAR(255)) RETURNS @table TABLE (email VARCHAR(255) NOT NULL, contribuinte CHAR(9) NOT NULL, fname VARCHAR(20) NOT NULL,
                                                                                  lname VARCHAR(20) NOT NULL, sexo BIT NOT NULL, dataNasc DATE NOT NULL,
                                                                                  foto VARCHAR(100) NOT NULL, contacto_default_id INT,
                                                                                  administrator TINYINT NOT NULL, salario INT NOT NULL, deleted BIT NOT NULL DEFAULT 0)
AS
BEGIN
        DECLARE @email AS VARCHAR(255), @contribuinte AS CHAR(9), @fname AS VARCHAR(20),
                @lname AS VARCHAR(20), @sexo AS BIT, @dataNasc AS DATE,
                @foto AS VARCHAR(100), @contacto_default_id AS INT, @deleted AS BIT,
                @administrator AS TINYINT, @salario AS INT;
        IF EXISTS(SELECT email FROM Perfumaria.perf.funcionario WHERE email=@emailFunc AND administrator=2)
            INSERT @table SELECT utilizador.email, contribuinte, fname, lname, sexo, dataNasc, foto, contacto_default_id, administrator, salario, deleted
                    FROM Perfumaria.perf.utilizador JOIN Perfumaria.perf.funcionario ON utilizador.email=funcionario.email
        RETURN;
      END;
GO
```

Figura 11: Exemplo de *UDF Multi-Statement Table-Valued*

12 Transações

Para garantir a consistência dos dados, foram utilizadas Transações que, com a sua atomicidade, garantem que, mesmo que uma das execuções de modificações de dados falhem, as outras todas são revertidas, mantendo coerência e integridade na Base de Dados. Estas foram utilizadas em casos onde existem múltiplas modificações de dados num *script* e é necessário garantir a sua execução total.

```
DROP PROCEDURE perf.addNewFunc;
GO

CREATE PROCEDURE perf.addNewFunc
    @email VARCHAR(255),
    @contribuinte CHAR(9),
    @fname VARCHAR(20),
    @lname VARCHAR(20),
    @pw VARCHAR(25),
    @sexo BIT,
    @dataNasc DATETIME,
    @foto VARCHAR(100),
    @contacto_default_id INT = NULL,
    @administrator TINYINT,
    @salario INT,
    @emailFunc VARCHAR(255),
    @responseMessage NVARCHAR(250) OUTPUT
AS
BEGIN
    BEGIN TRANSACTION
    SET NOCOUNT ON
    BEGIN TRY
        IF EXISTS(SELECT email FROM Perfumaria.perf.funcionario WHERE email=@emailFunc AND administrator=2)
            BEGIN
                INSERT INTO Perfumaria.perf.utilizador
                (email, contribuinte, fname, lname, pw, sexo, dataNasc, foto, contacto_default_id)
                VALUES(@email, @contribuinte, @fname, @lname, HASHBYTES('SHA2_512', @pw), @sexo, @dataNasc, @foto, @contacto_default_id)

                INSERT INTO Perfumaria.perf.funcionario
                (email, administrator, salario)
                VALUES(@email, @administrator, @salario)

                SET @responseMessage='Success'
            END
        ELSE
            SET @responseMessage='Permition denied'
        COMMIT TRANSACTION
    END TRY
    BEGIN CATCH
        SET @responseMessage='Failed'
        ROLLBACK
    END CATCH

END
GO
```

Figura 12: Exemplo de uma Transação, mantendo a consistência dos dados

13 Conclusão

Além das decisões já referidas e não havendo nenhuma *query* que álgebra relacional não conseguisse resolver, optou-se também por não utilizar Cursores devido à sua falta de desempenho.

De notar que a interface da aplicação não era o foco do projeto, pelo que a mesma poderá não estar visualmente bem concebida.

Concluindo, pensa-se que os objetivos do trabalho foram superados e que foi desenvolvido conhecimento na área do mesmo.

14 Bibliografia

[1] Material fornecido pelo docente da disciplina