

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	2
2	Análise de Requisitos	3
3	DER	6
4	Esquema Relacional da BD	7
5	SQL DDL	8
6	SQL DML	9
7	Normalização	9
8	Índices	9
9	Triggers	10
10	Stored Procedures	11
11	UDF	14
12	Transações	15
13	Conclusão	15
14	Bibliografia	16

1 Introdução

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 N° 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

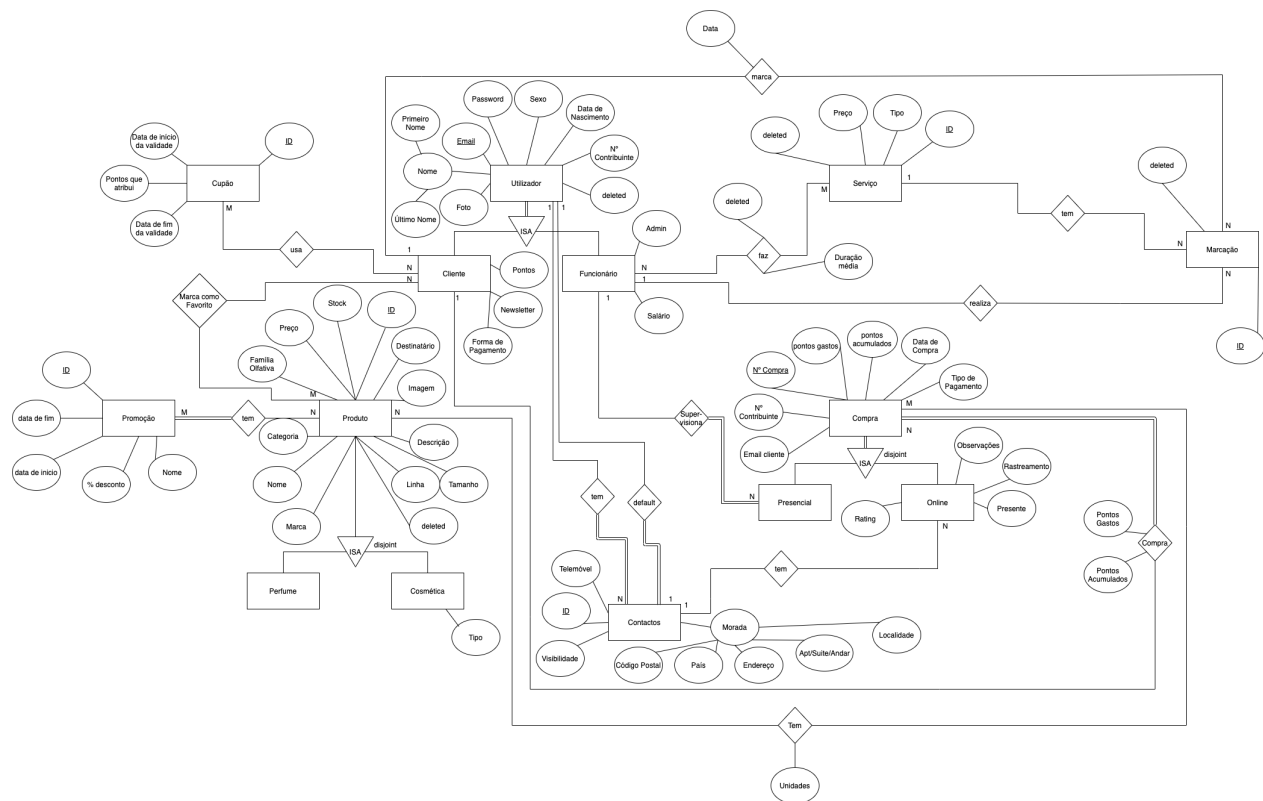


Figura 1: Diagrama Entidade/Relação da proposta de base de dados apresentada pelo grupo

4 Esquema Relacional da BD

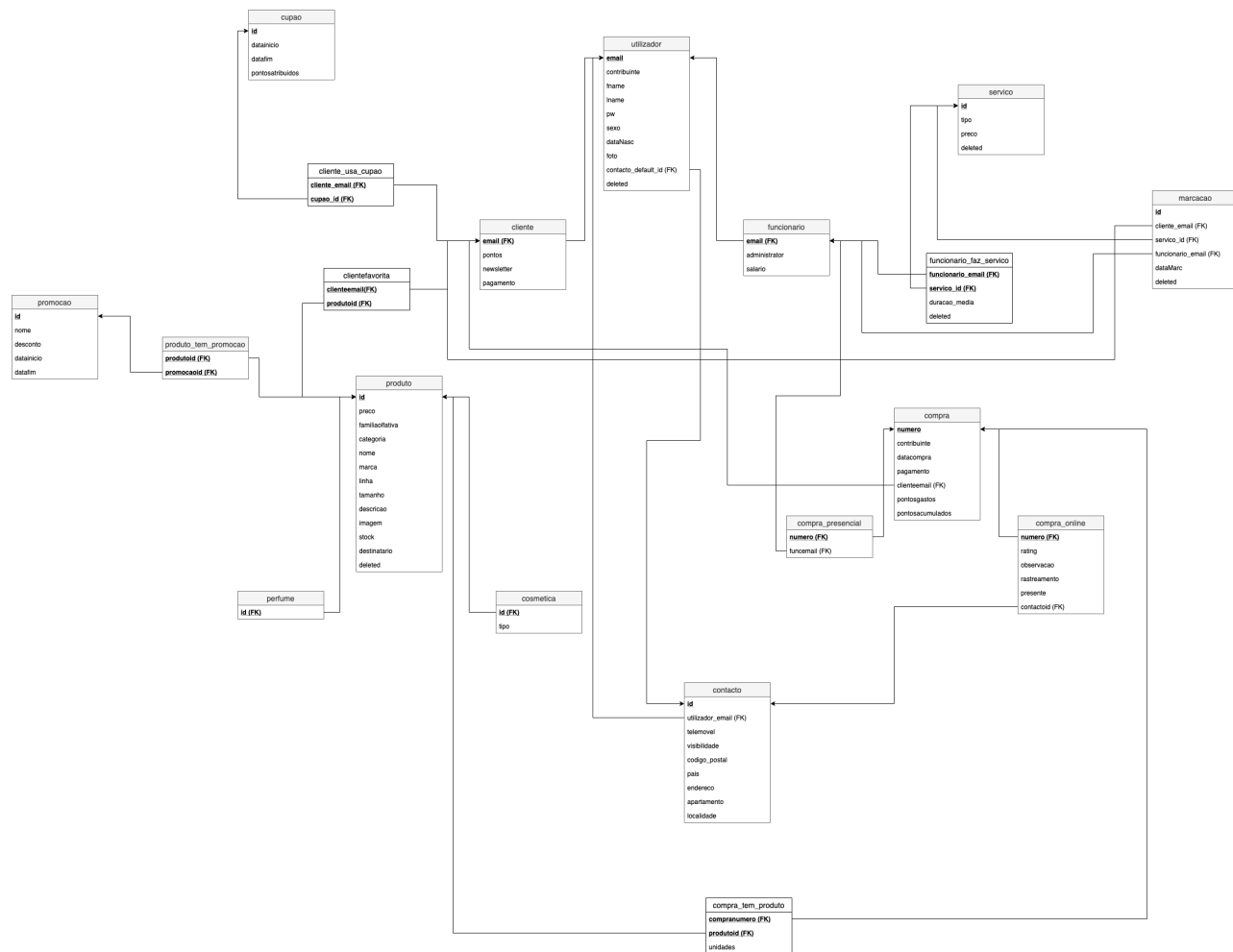


Figura 2: Modelo Entidade/Relação da proposta de base de dados apresentada pelo grupo

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 visível 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 GO
274
```

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 quando 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* (preço, 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 da *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 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 BD 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.

```
1 DROP PROCEDURE perf.addMarc;
2 GO
3
4
5 CREATE PROCEDURE perf.addMarc
6     @cliente_email VARCHAR(255),
7     @servico_id INT,
8     @funcionario_email VARCHAR(255),
9     @dataMarc SMALLDATETIME,
10    @responseMessage VARCHAR(258) = 'Erro! Tente noutra hora.' OUTPUT
11 AS
12 BEGIN
13     SET NOCOUNT ON
14     BEGIN TRY
15         DECLARE @duracao INT
16         SELECT @duracao = duracao_media FROM Perfumaria.perf.funcionario_faz_servico WHERE funcionario_email=@funcionario_email AND deleted = 0 AND servico_id = @servico_id
17         IF (@duracao IS NOT NULL AND @dataMarc > GETDATE())
18             BEGIN
19                 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)
20                     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))
21                     SET @responseMessage = 'Hora não disponível!'
22                 ELSE
23                     BEGIN
24                         INSERT INTO Perfumaria.perf.marcacao
25                         (cliente_email, servico_id, funcionario_email, dataMarc)
26                         VALUES(@cliente_email, @servico_id, @funcionario_email, @dataMarc)
27                     END
28                     SET @responseMessage='Marcação efetuado com sucesso!'
29             END
30         END TRY
31     END TRY
32     BEGIN CATCH
33         SET @responseMessage='Failure'
34     END CATCH
35
36
37 END
38 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.

```
1 DROP PROCEDURE perf.updateMarc;
2 GO
3
4 CREATE PROCEDURE perf.updateMarc
5     @cliente_email VARCHAR(255),
6     @idMarc INT,
7     @funcionario_email VARCHAR(255),
8     @dataMarc DATETIME = NULL,
9     @responseMessage NVARCHAR(258) = 'Erro! Tente noutra hora.' OUTPUT
10 AS
11 BEGIN
12     BEGIN TRY
13         SET NOCOUNT ON
14         IF EXISTS (SELECT email=@funcionario_email from Perfumaria.perf.funcionario)
15             BEGIN
16                 IF (@dataMarc IS NOT NULL)
17                     BEGIN TRY
18                         DECLARE @duracao INT
19                         SELECT @duracao=duracao_media FROM Perfumaria.perf.funcionario_faz_servico JOIN Perfumaria.perf.marcacao ON funcionario_faz_servico.servico_id=maracao.servico_id
20                         WHERE maracao.funcionario_email=@funcionario_email AND maracao.deleted = 0 AND maracao.id=@idMarc AND funcionario_faz_servico.funcionario_email=@funcionario_email
21                         IF (@duracao IS NOT NULL AND @dataMarc > GETDATE())
22                             BEGIN
23                                 IF (EXISTS(SELECT 1 FROM Perfumaria.perf.marcacao WHERE (dataMarc BETWEEN @dataMarc AND DATEADD(mi, @duracao, @dataMarc)) AND @funcionario_email=funcionario_email)
24                                     OR EXISTS(SELECT 1 FROM Perfumaria.perf.marcacao WHERE dataMarc BETWEEN @dataMarc AND DATEADD(mi, @duracao, @dataMarc) AND cliente_email=@cliente_email))
25                                     SET @responseMessage = 'Hora não disponível!'
26                                 ELSE
27                                     BEGIN
28                                         UPDATE Perfumaria.perf.marcacao
29                                         SET dataMarc=@dataMarc
30                                         WHERE id=@idMarc
31                                         SET @responseMessage='Marcação efetuado com sucesso!'
32                                     END
33                             END
34                         END TRY
35                     END TRY
36                 BEGIN CATCH
37                     SET @responseMessage='Failure'
38                 END CATCH
39             END
40         END TRY
41     END TRY
42     BEGIN CATCH
43         SET @responseMessage='Failed'
44     END CATCH
45
46 END
47 GO
```

Figura 7: Exemplo de uma Stored Procedure de update

Tem também alguns gets mais complexos, como por exemplo o getDetailsFromBuy, getDetailsFromSell, get-ProductFilters, etc.

```
1 DROP PROCEDURE perf.getProductFilters;
2 GO
3
4 CREATE PROCEDURE perf.getProductFilters
5     @deleted BIT = NULL,
6     @nome VARCHAR(30) = NULL,
7     @marca VARCHAR(30) = NULL,
8     @categoria VARCHAR(30) = NULL,
9     @destinatario VARCHAR(10) = NULL,
10    @orderby VARCHAR(50) = NULL,
11    @ordem VARCHAR(30) = NULL
12 AS
13 BEGIN
14     SET NOCOUNT ON
15     IF (@ordem = 'Ascendente')
16     BEGIN
17         SELECT id, preco, familiaolfativa, categoria, nome, marca, linha, tamanho, descricao, imagem, stock, destinatario, deleted
18         FROM Perfumaria.perf.produto
19         WHERE stock > 0 AND
20             deleted = 0 AND
21             nome LIKE ('%'+ISNULL(@nome, nome)+'%') AND
22             marca = ISNULL(@marca,marca) AND
23             categoria = ISNULL(@categoria,categoria)
24         ORDER BY CASE WHEN @orderby='Nome' THEN nome END,
25             CASE WHEN @orderby='Marca' THEN marca END,
26             CASE WHEN @orderby='Categoria' THEN categoria END,
27             CASE WHEN @orderby='Preço' THEN preco END
28     END
29
30     ELSE IF (@ordem = 'Descendente')
31     BEGIN
32         SELECT id, preco, familiaolfativa, categoria, nome, marca, linha, tamanho, descricao, imagem, stock, destinatario, deleted
33         FROM Perfumaria.perf.produto
34         WHERE stock > 0 AND
35             deleted = 0 AND
36             nome LIKE ('%'+ISNULL(@nome, nome)+'%') AND
37             marca = ISNULL(@marca,marca) AND
38             categoria = ISNULL(@categoria,categoria)
39         ORDER BY CASE WHEN @orderby='Nome' THEN nome END DESC,
40             CASE WHEN @orderby='Marca' THEN marca END DESC,
41             CASE WHEN @orderby='Categoria' THEN categoria END DESC,
42             CASE WHEN @orderby='Preço' THEN preco END DESC
43     END
44 END
45 GO
```

Figura 8: Exemplo de uma Stored Procedure de um get

Finalmente, tem 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 BD.

```
1  DROP PROCEDURE perf.RegisterFunc
2  GO
3
4
5  CREATE PROCEDURE perf.RegisterFunc
6      @email VARCHAR(255),
7      @password VARCHAR(25),
8      @contribuinte CHAR(9),
9      @fname VARCHAR(20),
10     @lname VARCHAR(20),
11     @sexo BIT,
12     @dataNasc DATE,
13     @foto VARCHAR(100),
14     @salario INT,
15     @administrator TINYINT,
16     @responseMessage NVARCHAR(250) OUTPUT
17 AS
18 BEGIN
19     BEGIN TRANSACTION
20     SET NOCOUNT ON
21
22     BEGIN TRY
23
24         INSERT INTO Perfumaria.perf.utilizador
25             (email, contribuinte, fname, lname, pw, sexo, dataNasc, foto)
26         VALUES(@email, @contribuinte, @fname, @lname, HASHBYTES('SHA2_512', @password), @sexo, @dataNasc, @foto)
27
28         INSERT INTO Perfumaria.perf.funcionario
29             (email, administrator, salario)
30         VALUES(@email, @administrator, @salario)
31
32         SET @responseMessage='Success'
33         COMMIT TRANSACTION
34     END TRY
35     BEGIN CATCH
36         SET @responseMessage=ERROR_MESSAGE()
37         ROLLBACK
38     END CATCH
39
40 END
41 GO
```

Figura 9: SP 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. Usamos UDFs Inline Table-Valued, como por exemplo clientBuyHistory, funcFutureMarc, etc.

```
1 DROP FUNCTION perf.funcFutureMarc;
2 GO
3
4 CREATE FUNCTION perf.funcFutureMarc (@email VARCHAR(255)) RETURNS TABLE
5 AS
6 RETURN (SELECT cliente_email,dataMarc, tipo, preco, fname, lname, marcacao.id
7         FROM ((Perfumaria.perf.marcacao JOIN Perfumaria.perf.servico ON servico_id=servico.id) JOIN Perfumaria.perf.utilizador ON cliente_email = email)
8         WHERE (funcionario_email=@email AND DATEDIFF(mi, GETDATE(), dataMarc) > 0)
9         ORDER BY dataMarc ASC OFFSET 0 ROWS)
10 GO
```

Figura 10: Exemplo de UDF Inline Table-Valued

Alem das Inline, utilizamos também as Multi-Statement Table-Valued para queries simples mas que necessitam de verificações iniciais, como por exemplo getAllFuncs, getAllProducts, etc.

```
1 DROP FUNCTION perf.getAllFuncs;
2 GO
3
4 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,
5                                         lname VARCHAR(20) NOT NULL, sexo BIT NOT NULL, dataNasc DATE NOT NULL,
6                                         foto VARCHAR(100) NOT NULL, contacto_default_id INT,
7                                         administrator TINYINT NOT NULL, salario INT NOT NULL, deleted BIT NOT NULL DEFAULT 0)
8 AS
9 BEGIN
10     DECLARE @email AS VARCHAR(255), @contribuinte AS CHAR(9), @fname AS VARCHAR(20),
11             @lname AS VARCHAR(20), @sexo AS BIT, @dataNasc AS DATE,
12             @foto AS VARCHAR(100), @contacto_default_id AS INT, @deleted AS BIT,
13             @administrator AS TINYINT, @salario AS INT;
14     IF EXISTS(SELECT email FROM Perfumaria.perf.funcionario WHERE email=@emailFunc AND administrator=2)
15         INSERT @table SELECT utilizador.email, contribuinte, fname, lname, sexo, dataNasc, foto, contacto_default_id, administrator, salario, deleted
16                         FROM Perfumaria.perf.utilizador JOIN Perfumaria.perf.funcionario ON utilizador.email=funcionario.email
17     RETURN;
18 END;
19 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.

```
1 DROP PROCEDURE perf.addNewFunc;
2 GO
3
4 CREATE PROCEDURE perf.addNewFunc
5     @email VARCHAR(255),
6     @contribuinte CHAR(9),
7     @fname VARCHAR(20),
8     @lname VARCHAR(20),
9     @pw VARCHAR(25),
10    @sexo BIT,
11    @dataNasc DATETIME,
12    @foto VARCHAR(100),
13    @contacto_default_id INT = NULL,
14    @administrator TINYINT,
15    @salario INT,
16    @emailFunc VARCHAR(255),
17    @responseMessage NVARCHAR(250) OUTPUT
18 AS
19 BEGIN
20     BEGIN TRANSACTION
21     SET NOCOUNT ON
22     BEGIN TRY
23         IF EXISTS(SELECT email FROM Perfumaria.perf.funcionario WHERE email=@emailFunc AND administrator=2)
24             BEGIN
25                 INSERT INTO Perfumaria.perf.utilizador
26                     (email, contribuinte, fname, lname, pw, sexo, dataNasc, foto, contacto_default_id)
27                     VALUES(@email, @contribuinte, @fname, @lname, HASHBYTES('SHA2_512', @pw), @sexo, @dataNasc, @foto, @contacto_default_id)
28
29                 INSERT INTO Perfumaria.perf.funcionario
30                     (email, administrator, salario)
31                     VALUES(@email, @administrator, @salario)
32
33                 SET @responseMessage='Success'
34             END
35         ELSE
36             SET @responseMessage='Permission denied'
37         COMMIT TRANSACTION
38     END TRY
39     BEGIN CATCH
40         SET @responseMessage='Failed'
41         ROLLBACK
42     END CATCH
43
44 END
45 GO
```

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

13 Conclusão

14 Bibliografia

[1] Material fornecido pelo Docente de Base de Dados