



universidade de aveiro

DEPARTAMENTO DE ELETRÓNICA, TELECOMUNICAÇÕES E INFORMÁTICA

BASES DE DADOS

2015/2016

Relatório Trabalho Prático Final

Grupo P4G9:

- Pedro Miguel André Coelho – Mec. 68803
- Vítor Silva Rodrigues – Mec. 72971

Engenharia de Computadores e Telemática



Índice

Índice

.....	
..... 2	
Introdução	
.....	
. 3	
Análise de Requisitos	
..... 4	
Diagrama	
Entidade-Relação.....	6
Modelo Relacional	
..... 7	
Normalização	
..... 8	
Criação de Tabelas	
..... 9	
Stored Procedures	
..... 11	
User-Defined Functions	
..... 14	
Indexes	
.....	
..... 16	
Triggers	
.....	
17	
Interface	
.....	
..... 18	



Conclusão

.....
.... 20

Referências

.....
21

Introdução

Este trabalho foi realizado no âmbito da unidade curricular Base de Dados do Mestrado Integrado em Engenharia de Computadores e Telemática da Universidade de Aveiro. O objetivo deste trabalho prático é o desenvolvimento de uma aplicação com uma interface gráfica que permita uma interação do utilizador com uma base de dados SQL Server para manipulação de dados.

Para os efeitos desejados, foram utilizadas as ferramentas Microsoft Management Studio 2012, para o desenvolvimento da base de dados no



servidor Microsoft SQL Server 2012, e Microsoft Visual Studio 2015, para o desenvolvimento da interface gráfica em C# e Visual Basic.

O relatório pretende incidir sobre o desenvolvimento da base de dados em SQL, bem como o de toda a interface a ela conectada desenvolvida no Visual Studio 2015, dando especial atenção à normalização e à utilização dos Stored Procedures, das User-Defined Functions, dos Indices e dos Triggers.

Para a criação deste projeto foi seguido o processo lecionado nas aulas, sendo estas as seguintes fases do processo: análise de requisitos, desenho conceptual, desenho do esquema lógico, desenho do esquema físico e administração.

Análise de Requisitos

É apresentada a seguir uma especificação de todas as entidades e relações existentes para a construção desta aplicação:

- Uma cadeia de cinemas é constituída por diversos cinemas em diferentes localizações;
- Um cinema tem associada a si um nome e um gerente do mesmo;
- Um cinema tem um conjunto de salas capazes de exibir filmes de diferentes tecnologias (IMAX, 3D, Digital);
- Um cinema tem em qualquer momento diversos filmes em exibição;

- Um filme tem associado a si um título, uma duração, uma descrição, uma restrição de idade, uma data de estreia, uma tecnologia, géneros, bem como qualquer outra informação necessária para a sua representação e uma distribuidora associada;
- Existem diferentes acordos com distribuidoras sendo que pode ser feito um pagamento inicial a este, bem como o pagamento de uma comissão por bilhete vendido;
- Os filmes poderão ter sessões atribuídas, em cada cinema, com repetição semanal, sendo que deverá poder ser aplicado um desconto a sessões específicas;
- Assim sendo, uma sessão tem associada a si uma hora, dia da semana, desconto e o cinema onde ocorre;
- O filme será exibido numa sala que poderá variar de sessão para sessão, bem como nos diferentes dias de cada sessão;
- Uma sala tem associada a si um número pré-determinado de lugares, distribuídos em filas e numerados por fila, que poderão ser normais ou aptos para clientes com necessidades especiais;
- Um cliente pode comprar bilhetes referentes à exibição de uma sessão num determinado dia, sendo que os lugares são escolhidos aquando a compra;

- Deverão existir diferentes tipos de bilhetes, com custo variável, cuja aquisição necessita que diversos requisitos sejam cumpridos pelo cliente (Normal, Estudante, Sénior, Criança, Funcionário, Cartão Cinema, Especial, ...);
- O bilhete é válido apenas para uma instância específica de visualização do filme;
- Deverá ser passada uma fatura/recibo associada à venda de um ou mais bilhetes, sendo registada a data e hora da compra, bem como o cliente e operador associado;
- Um funcionário trabalha para um cinema;
- Um funcionário terá a si associado os seus dados pessoais e um salário;
- Um Cliente poderá não fornecer NIF, sendo este designado por “Consumidor Final”;



- Um Cliente poderá ter um cartão de cinema associado que lhe permitirá adquirir bilhetes a preço reduzido, sendo que para isso deverá facultar os seus dados pessoais;
- Um funcionário poderá ser também um cliente, usufruindo da possibilidade de adquirir bilhetes de funcionário, com um custo reduzido.

A aplicação permite gerir:

- Cinemas:
 - Empregados;
 - Sessões;
 - Salas;
 - Bilhetes;
- Filmes;
- Tipos de Bilhete;
- Tecnologias de Filmes;
- Distribuidoras.

Diagrama Entidade-Relação

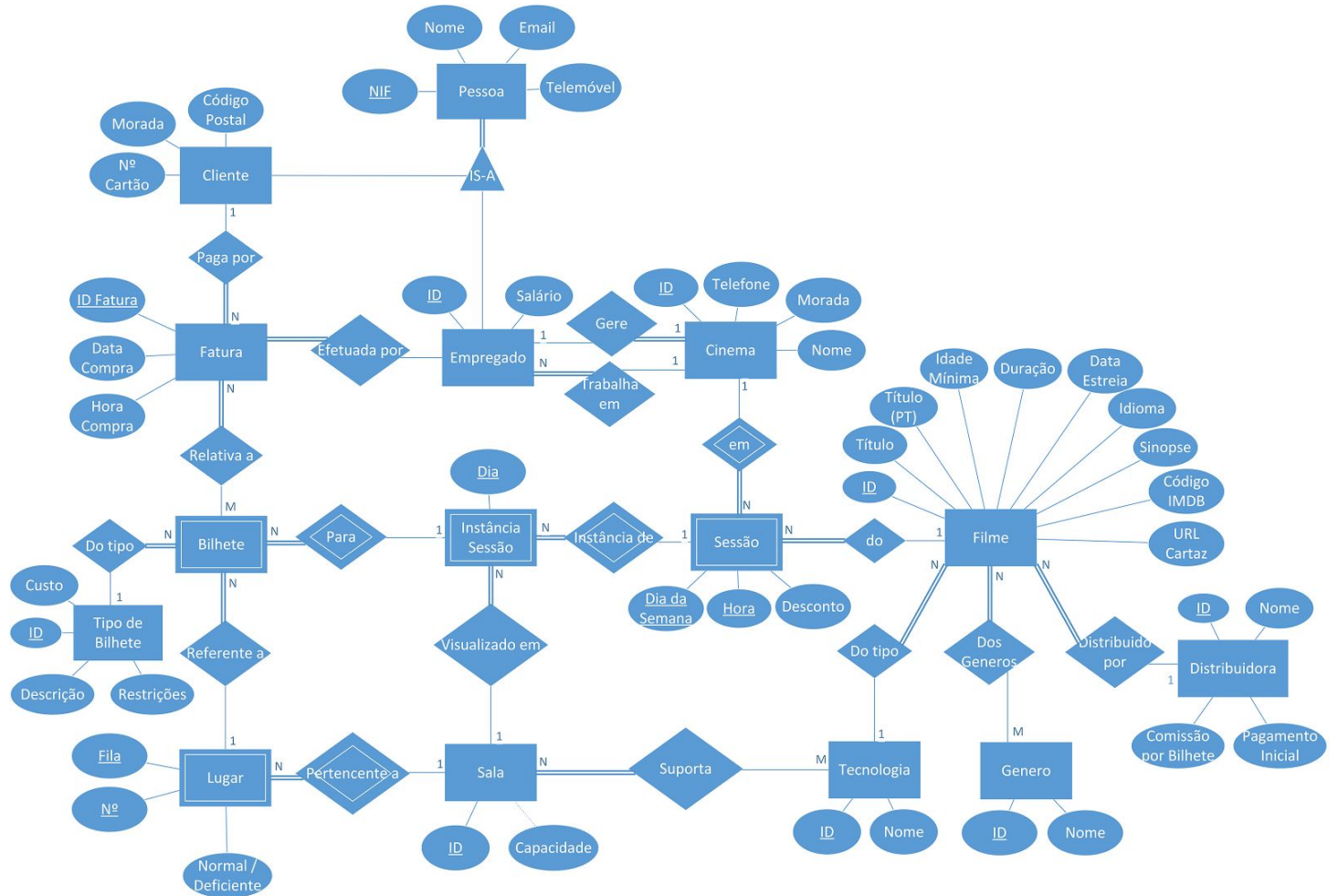


Figura 1 – Diagrama Entidade-Relação

Modelo Relacional



(meter o novo com as alterações)

Figura 2 – Modelo Relacional

Normalização



A normalização permite a integridade referencial entre relações e a não existência de redundâncias.

Podemos dizer que as relações da nossa base de dados encontram-se na 3ª Forma Normal e, sendo assim, também se encontram e satisfazem as condições da 1ª e 2ª Forma Normal.

Criação de Tabelas



A linguagem T-SQL DDL foi utilizada em todas as tabelas do trabalho final.

É possível observar uma demonstração da utilização desta mesma linguagem em algumas das tabelas:

```
-- CREATE SCHEMA and TABLES:
CREATE SCHEMA CinemasZ0Z;
GO

CREATE TABLE CinemasZ0Z.Distribuidora(
    id_dist INT PRIMARY KEY IDENTITY(1,1),
    nome_dist VARCHAR(30) UNIQUE NOT NULL,
    preco_inicial MONEY NOT NULL CHECK(preco_inicial > 0),
    comissao_bilhete SMALLMONEY NOT NULL CHECK(comissao_bilhete >= 0)
);

CREATE TABLE CinemasZ0Z.Tecnologia(
    id_tec INT PRIMARY KEY IDENTITY(1,1),
    nome_tec VARCHAR(7) UNIQUE NOT NULL,
    custo_extra SMALLMONEY NOT NULL
);

CREATE TABLE CinemasZ0Z.Cinema(
    id_cinema INT PRIMARY KEY IDENTITY(1,1),
    nome_cinema VARCHAR(30) UNIQUE NOT NULL,
    morada VARCHAR(100) NOT NULL,
    telefone INT NOT NULL CHECK(telefone LIKE REPLICATE('[0-9]', 9)),
    gerente INT
);

ALTER TABLE CinemasZ0Z.Cinema ADD CONSTRAINT FK_Cinema_Gerente FOREIGN KEY (gerente) REFERENCES CinemasZ0Z.Empregado(id_empregado);
ALTER TABLE CinemasZ0Z.Cinema ADD CONSTRAINT CK_Cinema_Gerente CHECK(CinemasZ0Z.verifyEmpregadoCinema(gerente, id_cinema) = 1);
```

Figura 4 – Tabelas Distribuidora, Tecnologia e Cinema



```
CREATE TABLE CinemasZ0Z.Empregado(  
    id_empregado INT PRIMARY KEY IDENTITY(1,1),  
    nome_empregado VARCHAR(50) NOT NULL,  
    nif INT CHECK(nif LIKE ('[1-9]' + REPLICATE('[0-9]', 8))),  
    email VARCHAR(50) UNIQUE NOT NULL CHECK(email LIKE '[a-z,0-9,-]@[a-z,0-9,-]%.[a-z][a-z]%.[a-z]'),  
    telemovel INT NOT NULL CHECK(telemovel LIKE ('[1-9]' + REPLICATE('[0-9]', 8))),  
    salario SMALLMONEY NOT NULL CHECK(salario > 0),  
    cinema INT REFERENCES CinemasZ0Z.Cinema(id_cinema)  
);  
  
CREATE TABLE CinemasZ0Z.Filme(  
    id_filme INT PRIMARY KEY IDENTITY(1,1),  
    id_dist INT NOT NULL REFERENCES CinemasZ0Z.Distribuidora(id_dist),  
    titulo VARCHAR(70) UNIQUE NOT NULL,  
    idade_min INT NOT NULL CHECK(idade_min >= 6 AND idade_min <= 18),  
    duracao INT NOT NULL CHECK(duracao > 0 AND duracao <= 300),  
    estreia DATE NOT NULL,  
    idioma CHAR(2) NOT NULL CHECK(Idioma LIKE '[A-Z][A-Z]')  
);  
  
CREATE TABLE CinemasZ0Z.Sala(  
    id_cinema INT NOT NULL REFERENCES CINEMASZ0Z.Cinema(id_cinema),  
    num_sala INT NOT NULL,  
    num_filas INT NOT NULL CHECK(num_filas > 0 AND num_filas < 10),  
    num_lugares_fila INT NOT NULL CHECK(num_lugares_fila > 0 AND num_lugares_fila < 20),  
    PRIMARY KEY (id_cinema, num_sala)  
);  
  
CREATE TABLE CinemasZ0Z.Sessao(  
    id_cinema INT REFERENCES CinemasZ0Z.Cinema(ID_Cinema),  
    dia_semana SMALLINT NOT NULL CHECK(dia_semana >= 1 AND dia_semana <= 7),  
    hora TIME NOT NULL,  
    desconto SMALLMONEY CHECK(desconto >= 0),  
    PRIMARY KEY(id_cinema, dia_semana, hora)  
);  
  
CREATE TABLE CinemasZ0Z.TipoBilhete(  
    id_tipo_bilhete INT PRIMARY KEY IDENTITY(1,1),  
    nome_bilhete VARCHAR(30) UNIQUE NOT NULL,  
    restricoes VARCHAR(30) NOT NULL,  
    custo SMALLMONEY NOT NULL CHECK(Custo > 0)  
);  
  
CREATE TABLE CinemasZ0Z.TecnologiasSala(  
    id_cinema INT NOT NULL,  
    num_sala INT NOT NULL,  
    id_tec INT NOT NULL REFERENCES CinemasZ0Z.Tecnologia(id_tec),  
    PRIMARY KEY (id_cinema, num_sala, id_tec),  
    FOREIGN KEY (id_cinema, num_sala) REFERENCES CinemasZ0Z.Sala(id_cinema, num_sala),  
);
```

Figura 5 – Tabelas Empregado, Filme, Sala, Sessão, TipoBilhete e TecnologiasSala



Stored Procedures

Stored Procedures são bastante importantes numa base de dados, pois permitem um maior desempenho, ajudam a prevenir SQL Injection e podem ser utilizados de forma a controlar permissões. Uma das outras vantagens é o facto de serem compilados apenas uma vez no momento de criação e carregados para memória na primeira utilização, em vez de serem compilados cada vez que fossem executados.

No nosso trabalho prático usámos Stored Procedures para inserções, updates e remoções. Desta forma, foi possível criar uma maior abstração de armazenamento de informação na base de dados.

Podemos ver alguns exemplos utilizados no nosso trabalho:

```
CREATE PROC CinemasZOZ.insertCinema (@NomeCinema VARCHAR(30), @Morada VARCHAR(100), @Telefone INT)
WITH ENCRYPTION, EXECUTE AS OWNER AS
BEGIN
    INSERT INTO CinemasZOZ.Cinema (nome_cinema, morada, telefone) VALUES (@NomeCinema, @Morada, @Telefone);
    RETURN SCOPE_IDENTITY();
END;
GO
GRANT EXECUTE ON CinemasZOZ.insertCinema TO cinemaAdmin;
GO

CREATE PROC CinemasZOZ.updateCinema (@IdCinema INT, @NomeCinema VARCHAR(30), @Morada VARCHAR(100),
                                     @Telefone INT, @Gerente INT = NULL)
WITH ENCRYPTION, EXECUTE AS OWNER AS
BEGIN
    UPDATE CinemasZOZ.Cinema SET nome_cinema = @NomeCinema, morada = @Morada, telefone = @Telefone,
                                gerente = @Gerente WHERE id_cinema = @IdCinema;
END;
GO
GRANT EXECUTE ON CinemasZOZ.updateCinema TO cinemaAdmin;
GO

CREATE PROC CinemasZOZ.removeCinema (@IdCinema INT)
WITH ENCRYPTION, EXECUTE AS OWNER AS
BEGIN
    DELETE FROM CinemasZOZ.Cinema WHERE id_cinema = @IdCinema;
END;
GO
GRANT EXECUTE ON CinemasZOZ.removeCinema TO cinemaAdmin;
GO
GRANT EXECUTE ON CinemasZOZ.removeCinema TO cinemaAdmin;
GO
```

Figura 6 – Código T-SQL de inserção, update e remoção de um cinema na base de dados

Estes três Stored Procedures funcionam como inserção, update e remoção de um cinema, respetivamente.

Por exemplo, para inserir um filme, o utilizador tem de chamar o Stored Procedure “CinemasZOZ.insertCinema” com os parâmetros definidos. No final, dá-se permissões das três funcionalidades apenas para o administrador da base de dados.

```
CREATE PROC CinemasZOZ.removeEmpregado (@IdEmpregado INT)
WITH ENCRYPTION, EXECUTE AS OWNER AS
BEGIN
    DELETE FROM CinemasZOZ.Empregado WHERE id_empregado = @IdEmpregado;
    DECLARE @LoginName sysname = 'ZOZ' + CAST(@IdEmpregado AS VARCHAR(10));
    EXEC sp_dropuser @LoginName;
    EXEC sp_droplogin @LoginName;
END;
GO
GRANT EXECUTE ON CinemasZOZ.removeEmpregado TO cinemaAdmin;
GO
```

Figura 7 – Código T-SQL de remoção de um empregado na base de dados

Este Stored Procedure funciona como remoção de um empregado.

Para o utilizador o remover tem de chamar o Stored Procedure “CinemasZOZ.removeEmpregado” com o seu respetivo id. Para remover também a sua entrada de login, tem de ser executada a instrução “sp_dropuser” com o seu login. No final, dá-se permissão da funcionalidade apenas para o administrador da base de dados.



```
CREATE PROC CinemasZOZ.updateSessao (@UserId INT, @Password VARCHAR(128), @DiaSemana SMALLINT, @Hora TIME,
@Desconto SMALLMONEY, @IdCinema INT = NULL, @NewDiaSemana SMALLINT = NULL, @NewHora TIME = NULL)
WITH ENCRYPTION, EXECUTE AS OWNER AS
BEGIN
    DECLARE @UserRole INT;
    SET @UserRole = CinemasZOZ.getUserRole(@UserId, @Password);
    IF @UserRole < 10 OR (@UserRole < 100 AND @IdCinema IS NOT NULL) OR (@UserRole >= 100 AND @IdCinema IS NULL)
        RETURN -1;
    IF @UserRole < 100
        SET @IdCinema = (SELECT cinema FROM CinemasZOZ.Empregado WHERE id_empregado = @UserId);
    IF @NewDiaSemana IS NULL
        SET @NewDiaSemana = @DiaSemana;
    IF @NewHora IS NULL
        SET @NewHora = @Hora;
    UPDATE CinemasZOZ.Sessao SET dia_semana = @NewDiaSemana, hora = @NewHora, desconto = @Desconto
    WHERE id_cinema = @IdCinema AND dia_semana = @DiaSemana AND hora = @Hora;
END;
GO
```

Figura 8 – Código T-SQL de update de uma sessão na base de dados

Este Stored Procedure funciona como update de uma sessão.

Para o utilizador alterar uma sessão tem de chamar o Stored Procedure “CinemasZOZ.updateSessao” com os parâmetros definidos para um novo dia da semana e uma nova hora.



User-Defined Functions

As User-Defined Functions (UDF) desempenham um papel importante na base de dados, pois para além de serem utilizadas como views parametrizáveis, também são mais rápidas e permitem mais “Check’s”. Tal como os Stored Procedures, também ajudam a prevenir SQL Injection.

Podemos ver algumas UDF’s utilizadas no nosso trabalho:

```
CREATE FUNCTION CinemasZOZ.getOwnUserRole()  
RETURNS INT WITH SCHEMABINDING, ENCRYPTION, EXECUTE AS CALLER AS  
BEGIN  
    IF (IS_ROLEMEMBER('cinemaAdmin') = 1 OR IS_ROLEMEMBER('db_owner') = 1)  
        RETURN 0;  
    IF (IS_ROLEMEMBER('cinemaGerente') = 1)  
        RETURN 1;  
    IF (IS_ROLEMEMBER('cinemaEmpregado') = 1)  
        RETURN 2;  
    RETURN -1;  
END;  
GO  
GRANT EXECUTE ON CinemasZOZ.getOwnUserRole TO cinemaEmpregado;  
GO
```

Figura 9 – Código da UDF para a verificação dos dados de login na base de dados

Esta UDF verifica os dados de login, para os três diferentes tipos de utilizadores que existem na base de dados (administrador, gerente e empregado).



```
CREATE FUNCTION CinemasZOZ.verifyEmpregadoCinema(@UserID INT, @CinemaID INT)
RETURNS BIT WITH SCHEMABINDING, ENCRYPTION AS
BEGIN
    IF @UserID IS NULL OR EXISTS(SELECT id_empregado, cinema
                                FROM CinemasZOZ.Empregado
                                WHERE id_empregado = @UserID AND cinema = @CinemaID)
        RETURN 1;
    RETURN 0;
END;
GO
```

Figura 10 – Código da UDF para a verificação de um empregado num cinema na base de dados

Esta UDF verifica se um funcionário pertence a um cinema, com o id do utilizador e o id do cinema nos parâmetros definidos.

```
CREATE FUNCTION CinemasZOZ.getListFilmes ()
RETURNS @table TABLE (id_filme INT, id_dist INT, titulo VARCHAR(70), idade_min INT, duracao INT, estreia DATE, idioma CHAR(2))
WITH SCHEMABINDING, ENCRYPTION, EXECUTE AS OWNER AS
BEGIN
    INSERT @table SELECT id_filme, id_dist, titulo, idade_min, duracao, estreia, idioma FROM CinemasZOZ.Filme ORDER BY titulo;
    RETURN;
END;
GO
GRANT SELECT ON CinemasZOZ.getListFilmes TO cinemaAdmin;
GO
```

Figura 11 – Código da UDF para a consulta dos filmes na base de dados

Esta UDF retorna os filmes da base de dados, acrescentando “WITH SCHEMABINDING”, para que a tabela não possa ser alterada.

No final, dá-se permissão da funcionalidade apenas para o administrador da base de dados.



Indexes

É necessário ter cuidados especiais sobre que índices escolher para a base de dados em questão. Estes são componentes essenciais quando se pretende fazer pesquisas em tempo útil num grande volume de dados. Por outro lado, podem também causar overhead na base de dados e tornar os tempos de inserção mais lentos.

Para a nossa base de dados optámos por índices em atributos que seriam os mais utilizados em consultas, como as chaves primárias.

Podemos ver alguns índices utilizados no nosso trabalho:

Triggers

Os Triggers asseguram verificações de integridade referencial definidas pelo utilizador. Eles garantem que, por exemplo, se um funcionário for inserido à base de dados, este possa ter as permissões definidas para um funcionário ou quando um funcionário muda de funções (passa, por exemplo, para gerente de um cinema) terão de lhe ser atribuídas mais permissões. Com isto, impede-se que haja possíveis inconsistências provocadas por utilizadores.

Podemos ver um trigger utilizado no nosso trabalho:

```
CREATE TRIGGER tg_addFuncionarioPermissions ON CinemasZOZ.Empregado AFTER INSERT
AS
    DECLARE @IdEmpregado INT;

    DECLARE c CURSOR FAST_FORWARD
    FOR SELECT id_empregado FROM inserted;

    OPEN c;
    FETCH c INTO @IdEmpregado;

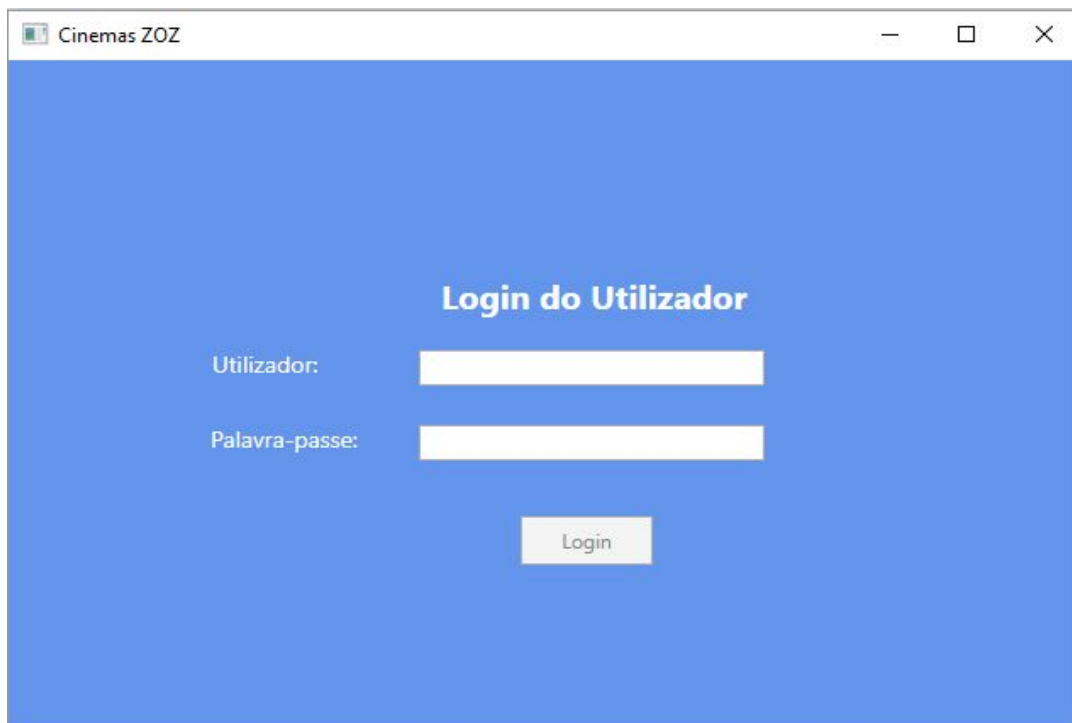
    WHILE @@FETCH_STATUS = 0
    BEGIN
        FETCH c INTO @IdEmpregado;
    END;

    CLOSE c;
    DEALLOCATE c;

GO
```

Figura 12 – Código T-SQL de criação do Trigger

Interface



Cinemas ZOZ

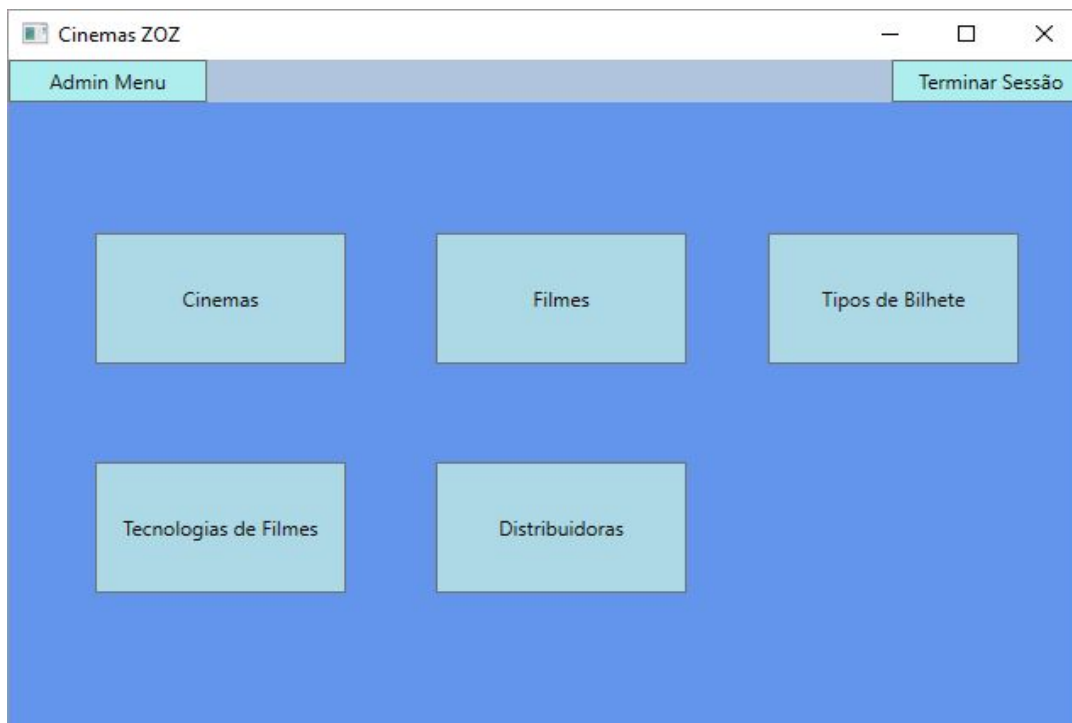
Login do Utilizador

Utilizador:

Palavra-passe:

Login

Figura 13 – Login de Administrador



Cinemas ZOZ

Admin Menu Terminar Sessão

Cinemas Filmes Tipos de Bilhete

Tecnologias de Filmes Distribuidoras

Figura 14 – Menu de Administrador





Conclusão

Este trabalho prático permitiu uma aplicação dos conhecimentos adquiridos ao longo das aulas teóricas e práticas da unidade curricular de Base de Dados, sendo que os objetivos propostos foram alcançados.

Ao longo do trabalho foram efetuadas algumas alterações tanto no diagrama entidade-relação como no esquema relacional da Base de Dados, no sentido de os melhorar.