

# Documentație Proiect Management Lant de Spitale

## 1. Descrierea Bazei de Date

Proiectul constă în crearea unei baze de date pentru gestionarea spitalelor, inclusiv informații despre spitale, manageri, secții, doctori, pacienți, programări și diagnostice. Structura bazei de date cuprinde 8 tabele interconectate prin relații de tip 1-n și m-n, cu constrângeri de integritate și indecși pentru optimizarea performanței.

Baza de date asigură operații CRUD prin proceduri stocate, triggere pentru automatizare și utilizatori cu drepturi specifice. De asemenea, include strategii de backup/restore și loguri de audit pentru o administrare eficientă și securizată a datelor din cadrul sistemului de management al spitalelor.

## 2. Creare Tabele

În această secțiune, se definesc tabelele necesare pentru a stoca datele despre managementul spitalelor. Tabelele au fost create pentru a reprezenta informațiile esențiale despre spitale, manageri, secții, doctori, pacienți, diagnostice și programări. Fiecare tabel este configurat cu câmpuri care reflectă informațiile relevante, dar, în această etapă, nu au fost adăugate constrângeri. Acestea vor fi implementate în pași următori pentru a asigura integritatea datelor și performanța bazei de date.

```
CREATE TABLE Manageri (
    id_manager INT PRIMARY KEY IDENTITY(1,1),
    nume VARCHAR(45) NOT NULL,
    salariu FLOAT NOT NULL,
    nr_telefon VARCHAR(45),
    email VARCHAR(45)
);

CREATE TABLE Spitale (
    id_spital INT PRIMARY KEY IDENTITY(1,1),
    denumire VARCHAR(45) NOT NULL,
    adresa VARCHAR(45) NOT NULL,
    id_manager INT
);

CREATE TABLE Sectii (
    id_sectie INT PRIMARY KEY IDENTITY(1,1),
    id_spital INT,
    denumire VARCHAR(45) NOT NULL
);

CREATE TABLE Doctori (
    id_doctor INT PRIMARY KEY IDENTITY(1,1),
    id_sectie INT,
    nume VARCHAR(45) NOT NULL,
    salariu FLOAT NOT NULL,
    sex VARCHAR(45),
    nr_telefon VARCHAR(45)
);

CREATE TABLE Pacienti (
    id_pacient INT PRIMARY KEY IDENTITY(1,1),
    nume VARCHAR(45) NOT NULL,
    adresa VARCHAR(45),
    sex VARCHAR(45),
    nr_telefon VARCHAR(45)
);
```

```

CREATE TABLE Diagnostice (
    id_diagnostic INT PRIMARY KEY IDENTITY(1,1),
    denumire VARCHAR(45) NOT NULL,
    detalii VARCHAR(45)
);

CREATE TABLE Programari (
    id_programare INT PRIMARY KEY IDENTITY(1,1),
    id_doctor INT,
    id_pacient INT,
    data_programare DATETIME NOT NULL
);

CREATE TABLE Pacienti_Diagnostice (
    id_pacient INT NOT NULL,
    id_diagnostic INT NOT NULL,
);

```

Fiecare tabel reprezintă o entitate distinctă din sistemul de management al spitalelor:

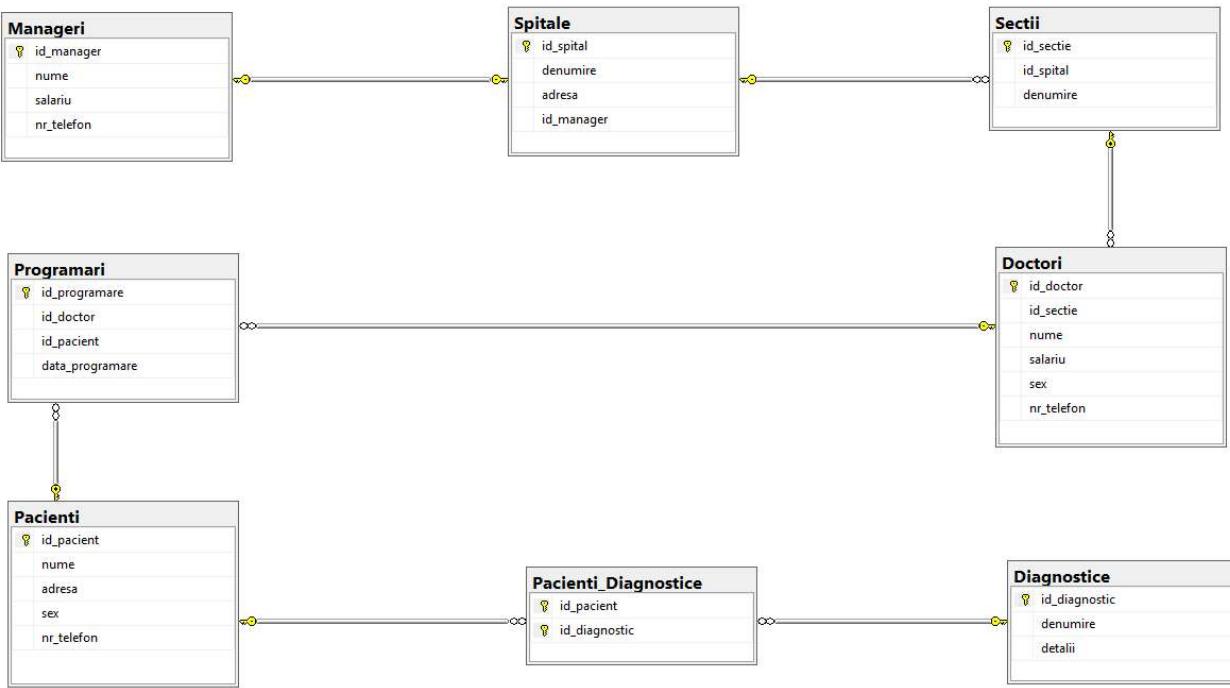
- **Manageri:** Conține informații despre managerii spitalelor, inclusiv numele, salariul și numărul de telefon al acestora.
- **Spitale:** Stochează informații despre spitale, cum ar fi denumirea, adresa și identificatorul managerului asociat.
- **Sectii:** Reprezintă secțiile din fiecare spital, cu informații despre numele secției și spitalul la care sunt asociate.
- **Doctori:** Stochează informații despre doctori, inclusiv numele, salariul, numărul de telefon și secția în care lucrează.
- **Pacienti:** Conține informații despre pacienți, cum ar fi numele, adresa, sexul și numărul de telefon.
- **Diagnostice:** Reprezintă diagnosticele medicale disponibile, stocând denumirea și detalii despre fiecare diagnostic în parte.
- **Programari:** Stochează informații despre programările pacienților cu medicii, inclusiv data și ora programării.
- **Pacienti\_Diagnostice:** Funcționează ca tabel intermedier pentru a crea o relație n:m între pacienți și diagnostice, indicând astfel diagnosticele asociate fiecărui pacient.

### 3. Relații între Tabele

Relațiile între tabele sunt esențiale pentru a modela interdependențele din sistem. După cum se observă în schemă, baza de date include 8 tabele: **manageri**, **spitale**, **secții**, **doctori**, **programări**, **pacienți**, **diagnostice** și un tabel intermedier **pacienți\_diagnostice**, care modelează relația n:m între pacienți și diagnostice.

Detalii despre relații:

- **1:1:** Între tabelul *Manageri* și *Spitale*. Fiecare spital are un singur manager, iar fiecare manager aparține unui singur spital.
- **1:n:** Între *Spitale* și *Secții*. Un spital poate avea mai multe secții.
- **1:n:** Între *Secții* și *Doctori*. O secție poate avea unul sau mai mulți doctori.
- **n:m:** Între *Doctori* și *Pacienți*. Un doctor poate consulta mai mulți pacienți, iar un pacient poate fi consultat de mai mulți doctori. Această relație este modelată prin tabelul *Programări*.
- **n:m:** Între *Pacienți* și *Diagnostice*. Un pacient poate avea mai multe diagnostice, iar un diagnostic poate fi asociat cu mai mulți pacienți. Această relație este modelată prin tabelul *Pacienți\_Diagnostice*.



## 4. Constrângerile și Indecși

Constrângerile implementate în baza de date includ:

- **FOREIGN KEY cu ON DELETE CASCADE**: Relațiile dintre tabele (de exemplu, *Manageri* și *Spitale*, *Sectii* și *Spitale*, *Doctori* și *Sectii*, *Programări* și *Pacienți*) sunt implementate prin chei externe. Opțiunea **ON DELETE CASCADE** asigură ștergerea automată a înregistrărilor dependente atunci când un rând asociat este șters.
- **CHECK**: Constrângerile care validează datele. De exemplu:
  - Salariul doctorilor trebuie să fie mai mare decât 0.
  - Sexul pacienților trebuie să fie unul dintre valorile permise: "Masculin", "Feminin" sau "Altul".
- **NOT NULL**: Câmpurile obligatorii, cum ar fi numele, numărul de telefon și datele de identificare (e.g., id-uri), sunt marcate cu **NOT NULL**.
- **Indecși**:
  - Un index unic este aplicat pe câmpul *email* din tabelul *Manageri* pentru a preveni duplicatele.
  - Un index unic este aplicat pe câmpul *id\_manager* din tabelul *Spitale* pentru a garanta că fiecare manager este asociat unui singur spital.
  - Indexarea câmpului *nume* din tabelul *Pacienți* pentru optimizarea interogărilor frecvente.
  - Indexarea câmpului *denumire* din tabelul *Diagnostice* pentru optimizarea căutărilor frecvente.
  - Indexarea câmpului *id\_sectie* din tabelul *Doctori* și a câmpului *data\_programare* din tabelul *Programări* pentru optimizarea interogărilor.
  - Indexuri suplimentare sunt aplicate pe câmpurile *id\_spital*, *id\_doctor* și *id\_patient* din tabelul *Programări*.
  - Indexuri suplimentare sunt aplicate pe câmpul *id\_patient* și *id\_diagnostic* din tabelul *Pacienti\_Diagnostice* pentru a optimiza interogările de asociere între pacienți și diagnostice.

### Cod SQL pentru Adăugarea Constrângerilor și Indecșilor

```

-- Relație 1:1 între Spatiale și Manageri
ALTER TABLE Spatiale
ADD CONSTRAINT FK_Spatiale_Manageri FOREIGN KEY (id_manager) REFERENCES Manageri(id_manager) ON DELETE CASCADE,
CONSTRAINT UQ_Spatiale_Manager UNIQUE (id_manager);
  
```

```
-- Relație 1:n intre Spitale și Sectii
ALTER TABLE Sectii
ADD CONSTRAINT FK_Sectii_Spitale FOREIGN KEY (id_spital) REFERENCES Spitale(id_spital) ON DELETE CASCADE;

-- Relație 1:n intre Sectii și Doctori
ALTER TABLE Doctori
ADD CONSTRAINT FK_Doctori_Sectii FOREIGN KEY (id_sectie) REFERENCES Sectii(id_sectie) ON DELETE CASCADE;

-- Relație n:m intre Doctori și Pacienti rezolvată prin adăugarea unei tabele de programări și relațiile:
-- Relație 1:n intre Doctori și Programari
ALTER TABLE Programari
ADD CONSTRAINT FK_Programari_Doctori FOREIGN KEY (id_doctor) REFERENCES Doctori(id_doctor) ON DELETE CASCADE;
-- Relație 1:n intre Pacienti și Programari
ALTER TABLE Programari
ADD CONSTRAINT FK_Programari_Pacienti FOREIGN KEY (id_pacient) REFERENCES Pacienti(id_pacient) ON DELETE CASCADE;

-- Relație n:m intre Pacienti și Diagnostice
ALTER TABLE Pacienti_Diagnostice
ADD CONSTRAINT PK_Pacienti_Diagnostice PRIMARY KEY (id_pacient, id_diagnostic),
CONSTRAINT FK_Pacienti_Diagnostice_Pacienti FOREIGN KEY (id_pacient) REFERENCES Pacienti(id_pacient) ON DELETE CASCADE,
CONSTRAINT FK_Pacienti_Diagnostice_Diagnostice FOREIGN KEY (id_diagnostic) REFERENCES Diagnostice(id_diagnostic) ON DELETE CASCADE;

-- Adăugarea indecșilor
CREATE UNIQUE INDEX IDX_Spitale_Manager ON Spitale(id_manager);
CREATE UNIQUE INDEX IDX_Email_Manager ON Manageri(email);
CREATE NONCLUSTERED INDEX IDX_Pacienti_Clustered ON Pacienti(nume);
CREATE NONCLUSTERED INDEX IDX_Diagnostice_Denumire ON Diagnostice(denumire);
CREATE INDEX IDX_Sectii_Spitale ON Sectii(id_spital);
CREATE INDEX IDX_Doctori_Sectii ON Doctori(id_sectie);
CREATE INDEX IDX_Programari_Doctori ON Programari(id_doctor);
CREATE INDEX IDX_Programari_Pacienti ON Programari(id_pacient);
CREATE INDEX IDX_Pacienti_Diagnostice_Pacienti ON Pacienti_Diagnostice(id_pacient);
CREATE INDEX IDX_Pacienti_Diagnostice_Diagnostice ON Pacienti_Diagnostice(id_diagnostic);
```

## 5. Vederi

Vederile sunt utilizate pentru a oferi informații agregate și pentru a simplifica interogările. În contextul bazei de date pentru spitale, acestea pot fi utilizate de diverse categorii de utilizatori, cum ar fi personalul administrativ, managerii de spitale sau doctorii, pentru a accesa informațiile relevante rapid și eficient. Exemple de vederi includ:

- **Vedere\_Doctori\_Sectii:** Vizualizarea informațiilor despre doctori, sectiile în care lucrează și spitalele aferente.
  - **Managerii spitalelor:** Utilizează această vedere pentru a avea o imagine clară asupra personalului medical, sectiilor și locațiilor acestora, ceea ce ajută la luarea deciziilor legate de resurse și gestionarea echipei.
  - **Personalul administrativ:** Se folosește de această vedere pentru a răspunde rapid întrebărilor despre personalul medical sau pentru a redirecționa pacienții către sectiile potrivite.
- **Vedere\_Programari:** Vizualizarea programărilor, inclusiv detaliilor despre pacienți și doctori.
  - **Pacienții:** Pot utiliza o interfață simplificată bazată pe această vedere pentru a verifica programările proprii și pentru a vedea detalii despre doctorul alocat.
  - **Personalul de programări/recepționerii:** Utilizează această vedere pentru a gestiona programările, a identifica eventualele conflicte sau a contacta pacienții pentru confirmări sau modificări.
  - **Managerii:** Pot folosi această vedere pentru a monitoriza eficiența doctorilor, analizând numărul de programări realizate.
- **Vedere\_Diagnostice\_Pacienti:** Vizualizarea diagnosticelor asociate pacienților, cu detalii relevante.
  - **Doctorii:** Utilizează această vedere pentru a accesa rapid istoricul diagnosticelor pacienților, ceea ce facilitează procesul decizional medical.
  - **Personalul administrativ:** Poate utiliza această vedere pentru a asista doctorii în pregătirea dosarelor medicale sau pentru a răspunde cererilor pacienților legate de istoricul lor medical.

```

CREATE VIEW Vedere_Doctori_Sectii AS
SELECT
    d.id_doctor,
    d.nume AS nume_doctor,
    d.salariu AS salariu_doctor,
    d.sex AS sex_doctor,
    d.nr_telefon AS telefon_doctor,
    s.denumire AS denumire_sectie,
    sp.denumire AS denumire_spital,
    sp.adresa AS adresa_spital
FROM
    Doctori d
JOIN
    Sectii s ON d.id_sectie = s.id_sectie
JOIN
    Spitale sp ON s.id_spital = sp.id_spital;

```

```

CREATE VIEW Vedere_Programari AS
SELECT
    pr.id_programare,
    pac.nume AS nume_pacient,
    pac.nr_telefon AS telefon_pacient,
    d.nume AS nume_doctor,
    d.nr_telefon AS telefon_doctor,
    d.salariu AS salariu_doctor,
    pr.data_programare
FROM
    Programari pr
JOIN
    Pacienti pac ON pr.id_pacient = pac.id_pacient
JOIN
    Doctori d ON pr.id_doctor = d.id_doctor;

```

```

CREATE VIEW Vedere_Diagnostice_Pacienti AS
SELECT
    pac.nume AS nume_pacient,
    pac.sex AS sex_pacient,
    pac.nr_telefon AS telefon_pacient,
    diag.denumire AS diagnostic,
    diag.detalii AS detalii_diagnostic
FROM
    Pacienti_Diagnostice pd
JOIN
    Pacienti pac ON pd.id_pacient = pac.id_pacient
JOIN
    Diagnostice diag ON pd.id_diagnostic = diag.id_diagnostic;

```

## Explicații și utilizatori principali:

- **Vedere\_Doctori\_Sectii:**

Utilizată de personalul administrativ și de managerii spitalelor pentru a avea o imagine clară a doctorilor și secțiilor în care activează.  
Este utilă pentru alocarea resurselor și pentru evaluări interne.

- **Vedere\_Programari:**

Utilizată de recepționeri sau de personalul de programări pentru a gestiona întâlnirile dintre pacienți și doctori. Poate fi folosită și pentru a contacta pacienții pentru confirmări sau modificări de programări.

- **Vedere\_Diagnostice\_Pacienti:**

Utilizată de doctori pentru a consulta rapid istoricul medical al pacienților și pentru a lua decizii medicale informate.

## 6. Proceduri Stocate (CRUD)

Procedurile stocate sunt folosite pentru a implementa operațiuni CRUD (Create, Read, Update, Delete). Ele oferă beneficii precum performanță îmbunătățită, securitate sporită și cod reutilizabil. Exemple de proceduri includ:

- **AddDoctor:** Procedură pentru adăugarea unui nou doctor în baza de date.
- **UpdatePatient:** Procedură pentru actualizarea informațiilor unui pacient.
- **DeleteAppointment:** Procedură pentru ștergerea unei programări.
- **GetDoctorsByHospital:** Procedură pentru afișarea doctorilor dintr-un spital.

```
-- Procedura pentru adăugarea unui doctor
CREATE PROCEDURE AddDoctor
    @id_sectie INT,
    @nume NVARCHAR(100),
    @salariu FLOAT,
    @sex NVARCHAR(10),
    @nr_telefon NVARCHAR(20)
AS
BEGIN
    IF NOT EXISTS (SELECT 1 FROM Sectii WHERE id_sectie = @id_sectie)
        BEGIN
            RAISERROR('Sectia specificata nu exista.', 16, 1);
            RETURN;
        END
    INSERT INTO Doctori (id_sectie, nume, salariu, sex, nr_telefon)
    VALUES (@id_sectie, @nume, @salariu, @sex, @nr_telefon);
    PRINT 'Doctor adaugat cu succes.';
END;

-- Procedura pentru actualizarea informațiilor unui pacient
CREATE PROCEDURE UpdatePatient
    @id_pacient INT,
    @nume NVARCHAR(100),
    @adresa NVARCHAR(100),
    @sex NVARCHAR(10),
    @nr_telefon NVARCHAR(20)
AS
BEGIN
    IF NOT EXISTS (SELECT 1 FROM Pacienti WHERE id_pacient = @id_pacient)
        BEGIN
            RAISERROR('Pacientul specificat nu exista.', 16, 1);
            RETURN;
        END
    UPDATE Pacienti
    SET nume = @nume, adresa = @adresa, sex = @sex, nr_telefon = @nr_telefon
    WHERE id_pacient = @id_pacient;
    PRINT 'Datele pacientului au fost actualizate.';
END;

-- Procedura pentru ștergerea unei programări
CREATE PROCEDURE DeleteAppointment
    @id_programare INT
AS
BEGIN
    IF NOT EXISTS (SELECT 1 FROM Programari WHERE id_programare = @id_programare)
        BEGIN
            RAISERROR('Programarea specificata nu exista.', 16, 1);
        END
    
```

```

        RETURN;
    END
    DELETE FROM Programari WHERE id_programare = @id_programare;
    PRINT 'Programarea a fost ștersă.';
END;

-- Procedura pentru afișarea doctorilor dintr-un spital
CREATE PROCEDURE GetDoctorsByHospital
    @id_spital INT
AS
BEGIN
    IF NOT EXISTS (SELECT 1 FROM Spitale WHERE id_spital = @id_spital)
    BEGIN
        RAISERROR('Spitalul specificat nu există.', 16, 1);
        RETURN;
    END
    SELECT
        d.nume AS DoctorName,
        d.salariu AS Salary,
        s.denumire AS SectionName
    FROM
        Doctori d
    JOIN
        Sectii s ON d.id_sectie = s.id_sectie
    WHERE
        s.id_spital = @id_spital;
END;

```

## Exemple de utilizare

- Adăugarea unui doctor:

```

EXEC AddDoctor
    @id_sectie = 1,
    @nume = 'Andrei Pop',
    @salariu = 4500,
    @sex = 'Masculin',
    @nr_telefon = '0787654321';

```

- Actualizarea unui pacient:

```

EXEC UpdatePatient
    @id_pacient = 1,
    @nume = 'Ana Ionescu',
    @adresa = 'Strada Păcii 12',
    @sex = 'Femin',
    @nr_telefon = '0723456789';

```

- Ștergerea unei programări:

```

EXEC DeleteAppointment
    @id_programare = 3;

```

- Afișarea doctorilor dintr-un spital:

```

EXEC GetDoctorsByHospital
    @id_spital = 1;

```

## Functii

Funcțiile sunt utilizate pentru a efectua calcule sau pentru a întoarce valori pe baza unor parametri. Spre deosebire de proceduri, acestea pot fi utilizate în interogări SELECT.

```
-- Funcție pentru calcularea salariului mediu al doctorilor dintr-un spital
CREATE FUNCTION GetAverageDoctorSalary(@id_spital INT)
RETURNS FLOAT
AS
BEGIN
    DECLARE @avgSalary FLOAT;
    SELECT @avgSalary = AVG(d.salariu)
    FROM Doctori d
    JOIN Sectii s ON d.id_sectie = s.id_sectie
    WHERE s.id_spital = @id_spital;
    RETURN @avgSalary;
END;

-- Funcție pentru întoarcerea numărului total de programări ale unui doctor
CREATE FUNCTION GetDoctorAppointmentCount(@id_doctor INT)
RETURNS INT
AS
BEGIN
    DECLARE @count INT;
    SELECT @count = COUNT(*)
    FROM Programari
    WHERE id_doctor = @id_doctor;
    RETURN @count;
END;
```

## Exemple de utilizare

- **Calcularea salariului mediu al doctorilor dintr-un spital:**

```
SELECT dbo.GetAverageDoctorSalary(1) AS AverageSalary;
```

- **Obținerea numărului total de programări ale unui doctor:**

```
SELECT dbo.GetDoctorAppointmentCount(3) AS AppointmentCount;
```

## Diferențe între proceduri și funcții

- **Returnare:** Procedurile nu returnează valori direct (în afară de OUTPUT sau coduri de eroare), în timp ce funcțiile returnează o valoare.
- **Utilizare:** Funcțiile pot fi utilizate în interogări SELECT, în timp ce procedurile sunt apelate direct cu EXEC.
- **Complexitate:** Procedurile permit operațiuni complexe, inclusiv DML (INSERT, UPDATE, DELETE), pe când funcțiile sunt limitate la operațiuni de citire.

## 7. Triggere DML și DDL

Triggerele sunt instrumente automatizate care sunt declanșate de acțiuni asupra bazei de date, cum ar fi operațiunile **INSERT**, **UPDATE** și **DELETE** pentru DML (Data Manipulation Language) și **CREATE**, **ALTER** și **DROP** pentru DDL (Data Definition Language). Acestea sunt folosite pentru a monitoriza, valida și preveni erori în manipularea datelor.

### Triggere DML

Triggerele DML sunt declanșate de operațiunile **INSERT**, **UPDATE** și **DELETE**.

- **trg\_ValidateInsertPatient:** Trigger care validează datele unui pacient înainte de inserare în tabelul *Pacienti*, asigurându-se că numele pacientului nu este gol și că numărul de telefon are cel puțin 10 caractere.
- **trg\_ValidateUpdateDoctorSalary:** Trigger care validează actualizarea salariului unui doctor, prevenind ca valoarea salariului să fie mai mică decât salariul minim permis.

```
-- 1. trg_ValidateInsertPatient
CREATE TRIGGER trg_ValidateInsertPatient
ON Pacienti
FOR INSERT
AS
BEGIN
    DECLARE @PacientName VARCHAR(45), @Phone VARCHAR(45);

    -- Obținem numele și numărul de telefon al pacientului introdus
    SELECT @PacientName = nume, @Phone = nr_telefon
    FROM inserted;

    -- Validăm că numele nu este gol și numărul de telefon este valid
    IF (@PacientName IS NULL OR @PacientName = '' OR LEN(@Phone) < 10)
        BEGIN
            RAISERROR('Date invalide! Numele nu poate fi gol și numărul de telefon trebuie să aibă cel puțin 10 caractere.', 16, 1);
            ROLLBACK TRANSACTION; -- Anulează inserarea
        END
    END;
END;

-- 2. trg_ValidateUpdateDoctorSalary
CREATE TRIGGER trg_ValidateUpdateDoctorSalary
ON Doctori
FOR UPDATE
AS
BEGIN
    DECLARE @OldSalary FLOAT, @NewSalary FLOAT, @DoctorID INT;

    -- Obținem vechiul salariu, noul salariu și ID-ul doctorului
    SELECT @OldSalary = salariu, @NewSalary = salariu, @DoctorID = id_doctor
    FROM inserted;

    -- Verificăm dacă salariul a fost redus la o valoare mai mică decât salariul minim permis
    IF (@NewSalary < 3000)
        BEGIN
            RAISERROR('Salariul doctorului nu poate fi mai mic de 3000!', 16, 1);
            ROLLBACK TRANSACTION; -- Anulează actualizarea
        END
    END;
END;
```

## Triggere DDL

Triggerele DDL sunt declanșate de modificările structurale ale bazei de date (CREATE, ALTER, DROP).

- **trg\_AuditDDLChanges:** Trigger care loghează toate acțiunile DDL (CREATE, ALTER, DROP) înregistrate în tabelul *Logs*, oferind un istoric al modificărilor structurale.
- **trg\_PreventDropSpitaleDoctori:** Trigger care previne ștergerea tabelelor sensibile, cum ar fi *Spitale* și *Doctori*, în cazul în care aceste tabele sunt implicate într-o acțiune de **DROP**.

```
-- 1. trg_AuditDDLChanges
CREATE TRIGGER trg_AuditDDLChanges
ON DATABASE
FOR CREATE_TABLE, ALTER_TABLE, DROP_TABLE
```

```

AS
BEGIN
    DECLARE @TableName NVARCHAR(100), @Action NVARCHAR(50), @Timestamp DATETIME;

    -- Obținem numele tabelului și tipul de acțiune
    SET @TableName = EVENTDATA().value('(/EVENT_INSTANCE/ObjectName)[1]', 'NVARCHAR(100)');
    SET @Action = EVENTDATA().value('(/EVENT_INSTANCE/EventType)[1]', 'NVARCHAR(50)');
    SET @Timestamp = GETDATE();

    -- Înregistram schimbările DDL într-un tabel de loguri
    INSERT INTO Logs (TableName, Action, PerformedBy, Timestamp)
    VALUES (@TableName, @Action, SYSTEM_USER, @Timestamp);

END;

```

-- 2. trg\_PreventDropSpitaleDoctori

```

CREATE TRIGGER trg_PreventDropSpitaleDoctori
ON DATABASE
FOR DROP_TABLE
AS
BEGIN
    DECLARE @TableName NVARCHAR(100);

    -- Obținem numele tabelului care se încearcă a fi șters
    SET @TableName = EVENTDATA().value('(/EVENT_INSTANCE/ObjectName)[1]', 'NVARCHAR(100)');

    -- Verificăm dacă tabelul care se încearcă ștergerea este unul dintre tabelele sensibile
    IF @TableName IN ('Spitale', 'Doctori')
        BEGIN
            RAISERROR('Nu se poate șterge tabelul %s! Este un tabel critic.', 16, 1, @TableName);
            ROLLBACK; -- Anulează acțiunea de ștergere
        END
    END;

```

## Explicație și Beneficii

**trg\_ValidateInsertPatient:** Acest trigger este folosit pentru a valida automat datele unui pacient înainte de a fi inserate în tabelul *Pacienti*. În cazul în care datele sunt invalide (de exemplu, numele este gol sau numărul de telefon nu are lungimea corectă), inserarea este anulată și o eroare este generată.

**trg\_ValidateUpdateDoctorSalary:** Acest trigger este folosit pentru a preveni actualizarea unui salar la o valoare mai mică decât salarul minim permis de 3000. Astfel, se garantează că salariile doctorilor nu vor scădea sub acest prag.

**trg\_AuditDDLChanges:** Acest trigger urmărește orice schimbare structurală a bazei de date (CREATE, ALTER, DROP) și înregistrează informațiile respective în tabelul *Logs*. Acesta este util pentru auditul modificărilor structurale ale bazei de date.

**trg\_PreventDropSpitaleDoctori:** Acest trigger previne ștergerea tabelelor sensibile, cum ar fi *Spitale* și *Doctori*, protejând astfel integritatea bazei de date. În cazul în care cineva încearcă să șteargă aceste tabele, o eroare este generată și acțiunea de ștergere este anulată.

## Testare

Pentru a testa aceste trigger, poți utiliza următoarele exemple:

```

-- Testare trg_ValidateInsertPatient
INSERT INTO Pacienti (nume, adresa, sex, nr_telefon)
VALUES ('Ion Popescu', 'Strada Exemplu, nr. 10', 'M', '0712345678'); -- Inserare validă

-- Testare trg_ValidateUpdateDoctorSalary
UPDATE Doctori
SET salar = 2500
WHERE id_doctor = 1; -- Va genera eroarea: "Salariul doctorului nu poate fi mai mic de 3000!"

```

```
-- Testare trg_AuditDDLChanges
CREATE TABLE TestTable (ID INT);

-- Testare trg_PreventDropSpitaleDoctori
DROP TABLE Spitale; -- Va genera eroarea: "Nu se poate șterge tabelul Spitale! Este un tabel critic."
```

## 8. Cursoare

În cadrul aplicației, am implementat trei proceduri stocate care utilizează cursoare pentru a efectua acțiuni secvențiale asupra bazei de date. Iată descrierea acestora:

- **sp\_UpdateDoctorSalaries**: Procedura stocată pentru mărirea salariilor doctorilor prin aplicarea unui procentaj pe salariul lor actual.
- **sp\_GetDoctorPhoneNumbers**: Procedura stocată pentru concatenarea numerelor de telefon ale doctorilor într-un singur sir de caractere.
- **sp\_CountFemalePatients**: Procedura stocată pentru numărarea pacientelor de sex feminin.

Aceste proceduri sunt implementate folosind cursoare, care permit procesarea secvențială a înregistrărilor din tabelele relevante.

### 1. Procedura pentru mărirea salariilor doctorilor

Procedura sp\_UpdateDoctorSalaries utilizează un cursor pentru a itera prin toți doctorii din tabelul Doctori și pentru a calcula și aplica o creștere procentuală asupra salariului lor actual. Această procedură este utilă atunci când dorim să actualizăm salariile pentru mai mulți doctori simultan.

```
-- 1. Procedura stocată pentru mărirea salariilor doctorilor
CREATE PROCEDURE sp_UpdateDoctorSalaries
    @Procent FLOAT
AS
BEGIN
    DECLARE @DoctorID INT, @SalariuActual FLOAT, @SalariuNou FLOAT;

    DECLARE SalaryCursor CURSOR FOR
        SELECT id_doctor, salariu
        FROM Doctori;

    OPEN SalaryCursor;
    FETCH NEXT FROM SalaryCursor INTO @DoctorID, @SalariuActual;

    WHILE @@FETCH_STATUS = 0
    BEGIN
        -- Calculăm noul salariu
        SET @SalariuNou = @SalariuActual + (@SalariuActual * @Procent);

        -- Actualizăm salariul în tabelul Doctori
        UPDATE Doctori
        SET salariu = @SalariuNou
        WHERE id_doctor = @DoctorID;

        FETCH NEXT FROM SalaryCursor INTO @DoctorID, @SalariuActual;
    END;

    CLOSE SalaryCursor;
    DEALLOCATE SalaryCursor;
END;
```

Testare:

```
-- Testare procedura pentru mărirea salariilor doctorilor cu 10%
EXEC sp_UpdateDoctorSalaries @Procent = 0.1;
```

## 2. Procedura pentru concatenarea numerelor de telefon ale doctorilor

Procedura sp\_GetDoctorPhoneNumbers folosește un cursor pentru a itera prin toate numerele de telefon ale doctorilor din tabelul Doctori și le concatenează într-un singur sir de caractere, separându-le prin virgulă. Această procedură este utilă pentru a obține rapid un sir cu toate numerele de telefon ale doctorilor.

```
-- 2. Procedura pentru concatenarea numerelor de telefon ale doctorilor
CREATE PROCEDURE sp_GetDoctorPhoneNumbers
AS
BEGIN
    DECLARE @PhoneNumber VARCHAR(255) = ''; -- Variabilă pentru stocarea rezultatului concatenat
    DECLARE @Phone VARCHAR(45);

    -- Cursorul selectează numerele de telefon ale doctorilor
    DECLARE PhoneCursor CURSOR FOR
        SELECT nr_telefon
        FROM Doctori
        WHERE nr_telefon IS NOT NULL;

    OPEN PhoneCursor;
    FETCH NEXT FROM PhoneCursor INTO @Phone;

    WHILE @@FETCH_STATUS = 0
    BEGIN
        -- Concatenează numărul de telefon la rezultatul existent
        SET @PhoneNumber = @PhoneNumber + @Phone + ', ';

        FETCH NEXT FROM PhoneCursor INTO @Phone;
    END;

    CLOSE PhoneCursor;
    DEALLOCATE PhoneCursor;

    -- Înlăturăm ultima virgulă și spațiu
    SET @PhoneNumber = LEFT(@PhoneNumber, LEN(@PhoneNumber) - 2);

    -- Returnăm rezultatul concatenat
    SELECT @PhoneNumber AS DoctorPhoneNumbers;
END;
```

Testare:

```
-- Testare procedura pentru a obține toate numerele de telefon ale doctorilor
EXEC sp_GetDoctorPhoneNumbers;
```

## 3. Procedura pentru numărarea pacientelor de sex feminin

Procedura sp\_CountFemalePatients folosește un cursor pentru a itera prin toți pacienții din tabelul Pacienti și pentru a număra câte paciente de sex feminin există. Această procedură este utilă pentru a obține rapid numărul pacientelor din baza de date.

```
-- 3. Procedura pentru numărarea pacientelor de sex feminin
CREATE PROCEDURE sp_CountFemalePatients
AS
BEGIN
    DECLARE @PatientCount INT = 0; -- Contor pentru numărul pacientelor
    DECLARE @Sex VARCHAR(45);

    -- Cursorul selectează sexul pacienților
    DECLARE PatientCursor CURSOR FOR
        SELECT sex
        FROM Pacienti;
```

```

SELECT sex
FROM Pacienti;

OPEN PatientCursor;
FETCH NEXT FROM PatientCursor INTO @Sex;

WHILE @@FETCH_STATUS = 0
BEGIN
    -- Incrementăm contorul dacă pacientul este de sex feminin
    IF @Sex = 'Feminin'
    BEGIN
        SET @PatientCount = @PatientCount + 1;
    END;

    FETCH NEXT FROM PatientCursor INTO @Sex;
END;

CLOSE PatientCursor;
DEALLOCATE PatientCursor;

-- Returnăm numărul pacientelor de sex feminin
SELECT @PatientCount AS FemalePatientCount;
END;

```

Testare:

```
-- Testare procedura pentru a număra pacientele de sex feminin
EXEC sp_CountFemalePatients;
```

### Descrierea cursurilor:

- **SalaryCursor:** Aceast cursor parcurge tabelul `Doctori` și actualizează salariile fiecărui doctor cu un procent specificat. Cursorul extrage `id_doctor` și salariu pentru fiecare înregistrare și actualizează salariul folosind formula calculată.
- **PhoneCursor:** Aceast cursor parcurge tabelul `Doctori` și concatenează numerele de telefon ale fiecărui doctor într-un sir de caractere, separate prin virgulă.
- **PatientCursor:** Aceast cursor parcurge tabelul `Pacienti` și numără pacientele de sex feminin. Dacă sexul pacientului este 'Feminin', contorul se incrementează.

## 9. Utilizatori și Roluri

În cadrul acestui proiect, utilizatorii și rolurile sunt definite pentru a controla accesul la baza de date `db_management_spitale`. Fiecare rol are permisiuni specifice pentru gestionarea și accesarea datelor.

- **AdminUser:** Permisii complete pentru administrarea bazei de date, inclusiv crearea, modificarea și stergerea obiectelor, precum și gestionarea utilizatorilor și permisiunilor.
- **PacientUser:** Permisii limitate, de doar **executare** a procedurilor. Poate face o programare, poate vizualiza istoricul programărilor și diagnosticelor sale.

### 1. Crearea utilizatorilor și a permisiunilor

Mai jos sunt pașii pentru crearea utilizatorilor și atribuirea permisiunilor specifice pentru fiecare rol:

```
-- Creare login pentru admin
CREATE LOGIN admin WITH PASSWORD = 'admin@123';
USE db_management_spitale;
CREATE USER admin FOR LOGIN admin;
GRANT CONTROL ON DATABASE::db_management_spitale TO admin;
```

```
-- Creare login pentru pacient
CREATE LOGIN pacient WITH PASSWORD = 'pacient@123';
USE db_management_spitale;
CREATE USER pacient FOR LOGIN pacient;
GRANT EXECUTE ON OBJECT::pacient_make_appointment TO pacient;
GRANT EXECUTE ON OBJECT::pacient_view_appointments TO pacient;
GRANT EXECUTE ON OBJECT::pacient_view_diagnostics TO pacient;
```

## 2. Proceduri stocate pentru pacient

Aceste proceduri permit pacientului să facă o programare, să vizualizeze istoricul programărilor și diagnosticelor sale.

### 2.1 pacient\_make\_appointment - Crearea unei programări

Această procedură permite pacientului să își facă o programare cu un doctor.

```
CREATE PROCEDURE pacient_make_appointment
    @patient_name VARCHAR(45),
    @doctor_name VARCHAR(45),
    @appointment_date DATETIME
AS
BEGIN
    -- Verifică dacă pacientul există
    IF NOT EXISTS (SELECT 1 FROM Pacienti WHERE nume = @patient_name)
    BEGIN
        RAISERROR('Pacientul specificat nu există.', 16, 1);
        RETURN;
    END

    -- Verifică dacă doctorul există
    IF NOT EXISTS (SELECT 1 FROM Doctori WHERE nume = @doctor_name)
    BEGIN
        RAISERROR('Doctorul specificat nu există.', 16, 1);
        RETURN;
    END

    -- Obține ID-ul pacientului și al doctorului
    DECLARE @patient_id INT, @doctor_id INT;
    SELECT @patient_id = id_pacient FROM Pacienti WHERE nume = @patient_name;
    SELECT @doctor_id = id_doctor FROM Doctori WHERE nume = @doctor_name;

    -- Introduce programarea în tabel
    INSERT INTO Programari (id_doctor, id_pacient, data_programare)
    VALUES (@doctor_id, @patient_id, @appointment_date);

    PRINT 'Programare realizată cu succes.';
END;
```

### 2.2 pacient\_view\_appointments - Vizualizarea programărilor pacientului

Această procedură permite pacientului să vizualizeze istoricul programărilor sale.

```
CREATE PROCEDURE pacient_view_appointments
    @patient_name VARCHAR(45)
AS
```

```

BEGIN
    -- Verifică dacă pacientul există
    IF NOT EXISTS (SELECT 1 FROM Pacienti WHERE nume = @patient_name)
    BEGIN
        RAISERROR('Pacientul specificat nu există.', 16, 1);
        RETURN;
    END

    -- Obține ID-ul pacientului
    DECLARE @patient_id INT;
    SELECT @patient_id = id_pacient FROM Pacienti WHERE nume = @patient_name;

    -- Afisează istoricul programărilor pacientului
    SELECT
        D.nume AS Doctor,
        P.data_programare AS DataProgramare
    FROM Programari P
    JOIN Doctori D ON P.id_doctor = D.id_doctor
    WHERE P.id_pacient = @patient_id;
END;

```

### **2.3 pacient\_view\_diagnostic - Vizualizarea diagnosticelor pacientului**

Această procedură permite pacientului să vizualizeze diagnosticele sale.

```

CREATE PROCEDURE pacient_view_diagnostic
    @patient_name VARCHAR(45)
AS
BEGIN
    -- Verifică dacă pacientul există
    IF NOT EXISTS (SELECT 1 FROM Pacienti WHERE nume = @patient_name)
    BEGIN
        RAISERROR('Pacientul specificat nu există.', 16, 1);
        RETURN;
    END

    -- Obține ID-ul pacientului
    DECLARE @patient_id INT;
    SELECT @patient_id = id_pacient FROM Pacienti WHERE nume = @patient_name;

    -- Afisează diagnosticele pacientului
    SELECT
        D.denumire AS Diagnostic,
        D.detalii AS Detalii
    FROM Diagnostice D
    JOIN Pacienti_Diagnostice PD ON D.id_diagnostic = PD.id_diagnostic
    WHERE PD.id_pacient = @patient_id;
END;

```

## **4. Exemplu de utilizare a procedurilor**

Mai jos sunt exemple pentru a utiliza procedurile create:

**Exemplu pentru pacient\_make\_appointment:**

```
EXEC pacient_make_appointment
@patient_name = 'Ana Ionescu',
@doctor_name = 'Ion Popescu',
@appointment_date = '2024-12-20 10:00:00';
```

**Exemplu pentru pacient\_view\_appointments:**

```
EXEC pacient_view_appointments
@patient_name = 'Ana Ionescu';
```

**Exemplu pentru pacient\_view\_diagnostics:**

```
EXEC pacient_view_diagnostics
@patient_name = 'Ana Ionescu';
```

## 10. Job-uri TransactSQL

Job-urile sunt folosite pentru a automatiza sarcini recurante. Exemple:

- Calculate Daily Revenue:** Un job care calculează veniturile zilnice și le inserează în tabelă.
- Calculate Doctor and Patient Stats:** Un job care calculează numărul de doctori și pacienți tratați zilnic și le inserează într-o tabelă dedicată.

```
use showroomDB;
CREATE TABLE DailyRevenue (
    RevenueID INT IDENTITY PRIMARY KEY,
    RevenueDate DATE,
    TotalRevenue DECIMAL(10, 2)
);

USE msdb;
GO

EXEC sp_add_job
    @job_name = 'Calculate Daily Revenue',
    @enabled = 1,
    @description = 'Job care calculeaza veniturile zilnice si le insereaza in tabela.';

EXEC sp_add_jobstep
    @job_name = 'Calculate Daily Revenue',
    @step_name = 'Calculate Daily Revenue Step',
    @subsystem = 'TSQL',
    @command = '
        USE showroomDB;
        INSERT INTO DailyRevenue (RevenueDate, TotalRevenue)
        SELECT
            CAST(GETDATE() AS DATE) AS RevenueDate,
            SUM(t.TotalPrice) AS TotalRevenue
        FROM
            Transactions t
        WHERE
            t.TransactionType = 'Sales'';

-- Set the step to run every day at 10:00 AM
ALTER JOB [Calculate Daily Revenue]
    ADD STEP [Calculate Daily Revenue Step]
    WITH ON_FAILURE_STOP = 0;
```

```

        CAST(t.Date AS DATE) = CAST(GETDATE() AS DATE);
        ',
        @database_name = 'showroomDB';

EXEC sp_add_jobschedule
    @job_name = 'Calculate Daily Revenue',
    @name = 'Daily at 21:00',
    @enabled = 1,
    @freq_type = 4, -- Daily
    @freq_interval = 1,
    @active_start_time = 210000; -- 21:00:00
GO

-- Creare job pentru statistica doctori și pacienți
USE db_management_spitale;
GO

CREATE TABLE DailyDoctorPatientStats (
    StatID INT IDENTITY PRIMARY KEY,
    StatDate DATE,
    TotalDoctors INT,
    TotalPatients INT
);

USE msdb;
GO

-- Creare job
EXEC sp_add_job
    @job_name = 'Calculate Doctor and Patient Stats',
    @enabled = 1,
    @description = 'Job care calculează numărul de doctori și pacienți tratați zilnic și le inserează în tabelă.';

-- Creare pas de job
EXEC sp_add_jobstep
    @job_name = 'Calculate Doctor and Patient Stats',
    @step_name = 'Calculate Stats Step',
    @subsystem = 'TSQL',
    @command =
        USE db_management_spitale;

        DECLARE @TotalDoctors INT, @TotalPatients INT;

        -- Număr total de doctori
        SELECT @TotalDoctors = COUNT(*)
        FROM Doctori;

        -- Număr total de pacienți tratați (adăugați condițiile pentru a selecta doar pacienții tratați)
        SELECT @TotalPatients = COUNT(DISTINCT P.id_pacient)
        FROM Programari P
        JOIN Pacienti D ON P.id_pacient = D.id_pacient
        WHERE P.data_programare <= GETDATE(); -- Pacienți care au fost tratați până azi

        -- Inserare statistici în tabelă
        INSERT INTO DailyDoctorPatientStats (StatDate, TotalDoctors, TotalPatients)
        VALUES (CAST(GETDATE() AS DATE), @TotalDoctors, @TotalPatients);
        ',
        @database_name = 'db_management_spitale';

-- Creare programare job
EXEC sp_add_jobschedule
    @job_name = 'Calculate Doctor and Patient Stats',
    @name = 'Daily at 23:59',
    @enabled = 1,

```

```

@freq_type = 4, -- Daily
@freq_interval = 1,
@active_start_time = 235900; -- 23:59:00
GO

```

## 11. Backup-ul Bazei de Date

Pentru a asigura protecția și recuperarea datelor în caz de pierderi sau corupere, sunt implementate mai multe strategii de backup pentru baza de date *db\_management\_spitale*. Acestea includ:

- **Backup zilnic complet:** Creează un backup complet al bazei de date în fiecare zi.
- **Backup diferențial:** Efectuat la fiecare 6 ore, conține doar modificările efectuate de la ultimul backup complet.
- **Backup al jurnalului de tranzacții:** Salvează jurnalul de tranzacții pentru a permite recuperarea la un punct specific în timp.

```

USE master;
GO

-- Full Backup (zilnic)
BACKUP DATABASE db_management_spitale
TO DISK = 'C:\SQLBackups\db_management_spitale_FullBackup.bak'
WITH FORMAT,
      INIT,
      NAME = 'Full Backup of db_management_spitale',
      STATS = 10; -- Afisează progresul backup-ului

-- Backup Differential (la fiecare 6 ore)
USE master;
GO

BACKUP DATABASE db_management_spitale
TO DISK = 'C:\SQLBackups\db_management_spitale_DifferentialBackup.bak'
WITH DIFFERENTIAL,
      INIT,
      NAME = 'Differential Backup of db_management_spitale',
      STATS = 10;

-- Backup al jurnalului de tranzactii
USE master;
GO

BACKUP LOG db_management_spitale
TO DISK = 'C:\SQLBackups\db_management_spitale_LogBackup.trn'
WITH INIT,
      NAME = 'Transaction Log Backup of db_management_spitale',
      STATS = 10;

```