Transact SQL

1. Create a database with two tables: Persons(Id(PK), FirstName, LastName, SSN) and Accounts(Id(PK), PersonId(FK), Balance). Insert few records for testing. Write a stored procedure that selects the full names of all persons.

```
CREATE TABLE Persons (
        PersonID  INT PRIMARY KEY IDENTITY(1,1) NOT NULL,
        FirstName NVARCHAR(100) NOT NULL,
        LastName NVARCHAR(100) NOT NULL,
        SSN NVARCHAR(40) NULL
)

INSERT INTO Persons
VALUES
       ('Ivan', 'Georgiev', '091241231924124'),
       ('Ivancho', 'Georgiev', '091241231924124'),
       ('Penka', 'Georgieva', '091241231924124'),
       ('Gosho', 'Georgiev', '091241231924124'),
       ('Ivanka', 'Georgieva', '091241231924124')

CREATE TABLE Accounts (
        AccountID INT PRIMARY KEY IDENTITY(1,1) NOT NULL,
        PersonID INT
                    CONSTRAINT FK_PersonID
                    FOREIGN KEY(PersonID)
                    REFERENCES Persons(PersonID) NOT NULL,
        Balance money NULL
)

INSERT INTO Accounts
VALUES
       (1, 1000),
       (2, 2000),
       (3, 3000),
       (4, 4000),
       (5, 5000)
GO

CREATE PROC dbo.usp_SelectFullNamesFromPersons
AS
        SELECT p.FirstName + ' ' + p.LastName
        FROM Persons p
GO

EXEC dbo.usp_SelectFullNamesFromPersons
```

2. Create a stored procedure that accepts a number as a parameter and returns all persons who have more money in their accounts than the supplied number.

```
CREATE PROC dbo.usp_SelectPersonsWithMoney (@money int)
AS
        SELECT p.FirstName + ' ' + p.LastName
        FROM Persons p
                    JOIN Accounts a
                      ON p.PersonID = a.AccountID
              WHERE a.Balance >= @money
GO
EXEC dbo.usp_SelectPersonsWithMoney 2000
```

3. Create a function that accepts as parameters – sum, yearly interest rate and number of months. It should calculate and return the new sum. Write a SELECT to test whether the function works as expected.

```
CREATE PROC dbo.usp_FindMoneyByInterestRate (@sum int, @yearlyInterestPercent float,
@months int)
AS
        SELECT @sum + @sum * (((@yearlyInterestPercent / 12) / 100) * @months)
GO
EXEC dbo.usp_FindMoneyByInterestRate 2000, 12, 1
```

4. Create a stored procedure that uses the function from the previous example to give an interest to a person's account for one month. It should take the AccountId and the interest rate as parameters.

```
CREATE PROC dbo.usp_GiveOneMonthMoney (@accountID int, @yearlyInterestPercent float)
AS
              DECLARE @oldMoney money
              SET @oldMoney = (SELECT Balance FROM Accounts WHERE AccountID = @accountID)

              DECLARE @newMoney money
              CREATE TABLE #tmpTable
              (
                    OutputValue money
              )
              INSERT INTO #tmpTable (OutputValue)
              EXEC dbo.usp_FindMoneyByInterestRate 2000, @yearlyInterestPercent, 1;

              SELECT @newMoney = OutputValue
              FROM #tmpTable
              DROP TABLE #tmpTable

        UPDATE Accounts
              SET Balance = @newMoney
              FROM Accounts
              WHERE AccountID = @accountID
GO
EXEC dbo.usp_GiveOneMonthMoney 2, 36
```

5. Add two more stored procedures WithdrawMoney(AccountId, money) and DepositMoney (AccountId, money) that operate in transactions

```
CREATE PROC dbo.usp_WithdrawMoney (@accountID int, @amount money)
AS
	BEGIN TRAN
	UPDATE Accounts
	SET Balance = Balance - @amount
	FROM Accounts
	WHERE AccountID = @accountID
	COMMIT TRAN
GO

CREATE PROC dbo.usp_DepositMoney (@accountID int, @amount money)
AS
	BEGIN TRAN
	UPDATE Accounts
	SET Balance = Balance + @amount
	FROM Accounts
	WHERE AccountID = @accountID
	COMMIT TRAN
GO

EXEC dbo.usp_WithdrawMoney 2, 1000

EXEC dbo.usp_DepositMoney 3, 1000
```

6. Create another table – Logs(LogID, AccountID, OldSum, NewSum). Add a trigger to the Accounts table that enters a new entry into the Logs table every time the sum on an account changes.

```
CREATE Trigger TR_AccountUpdate ON dbo.Accounts FOR UPDATE
AS
	BEGIN
	INSERT INTO dbo.Logs(AccountID, OldSum, NewSum)
	SELECT inserted.AccountID, deleted.Balance, inserted.Balance
	FROM inserted, deleted
	END
```

7. Define a function in the database TelerikAcademy that returns all Employee's names (first or middle or last name) and all town's names that are comprised of given set of letters. Example 'oistmiahf' will return 'Sofia', 'Smith', … but not 'Rob' and 'Guy'.

```
CREATE FUNCTION fn_ListTownsPersonsWithLetters(@letters nvarchar(MAX))
RETURNS TABLE
AS
  RETURN
  (
  SELECT FirstName
  FROM Employees
  WHERE dbo.fn_StringContainsName(@letters, FirstName) = 1
  UNION
  SELECT MiddleName
  FROM Employees
  WHERE dbo.fn_StringContainsName(@letters, MiddleName) = 1
  UNION
  SELECT LastName
  FROM Employees
  WHERE dbo.fn_StringContainsName(@letters, LastName) = 1
  UNION
  SELECT Name
  FROM Towns
  WHERE dbo.fn_StringContainsName(@letters, Name) = 1
  )
GO

ALTER FUNCTION fn_StringContainsName (@string nvarchar(MAX), @name nvarchar(MAX))
RETURNS bit
AS
BEGIN
       DECLARE @counter int = 1
       WHILE (@counter <= LEN(@name))
              BEGIN
              IF (CHARINDEX(SUBSTRING(@name, @counter, 1), @string) = 0)
                     RETURN 0
              SET @counter = @counter + 1
              END
RETURN 1
END
GO

SELECT * FROM fn_ListTownsPersonsWithLetters('abcdefghijklmnopqrstuvwxyz')
```

8. Using database cursor write a T-SQL script that scans all employees and their addresses and prints all pairs of employees that live in the same town.

```
DECLARE empCursor CURSOR READ_ONLY FOR

SELECT a.FirstName, a.LastName, t1.Name, b.FirstName, b.LastName
FROM Employees a
JOIN Addresses adr
ON a.AddressID = adr.AddressID
JOIN Towns t1
ON adr.TownID = t1.TownID,
 Employees b
JOIN Addresses ad
ON b.AddressID = ad.AddressID
JOIN Towns t2
ON ad.TownID = t2.TownID
WHERE t1.Name = t2.Name
  AND a.EmployeeID <> b.EmployeeID
ORDER BY a.FirstName, b.FirstName

OPEN empCursor
DECLARE @firstName1 NVARCHAR(50)
DECLARE @lastName1 NVARCHAR(50)
DECLARE @town NVARCHAR(50)
DECLARE @firstName2 NVARCHAR(50)
DECLARE @lastName2 NVARCHAR(50)
FETCH NEXT FROM empCursor
        INTO @firstName1, @lastName1, @town, @firstName2, @lastName2

WHILE @@FETCH_STATUS = 0
        BEGIN
                PRINT @firstName1 + ' ' + @lastName1 +
                        '     ' + @town + '      ' + @firstName2 + ' ' + @lastName2
                FETCH NEXT FROM empCursor
                        INTO @firstName1, @lastName1, @town, @firstName2, @lastName2
        END

CLOSE empCursor
DEALLOCATE empCursor
```

9. * Write a T-SQL script that shows for each town a list of all employees that live in it. Sample output:

```
USE TelerikAcademy
DECLARE empCursor CURSOR READ_ONLY FOR
SELECT Name FROM Towns
OPEN empCursor
DECLARE @townName VARCHAR(50), @userNames VARCHAR(MAX)
FETCH NEXT FROM empCursor INTO @townName


WHILE @@FETCH_STATUS = 0
  BEGIN
                BEGIN
                DECLARE nameCursor CURSOR READ_ONLY FOR
                SELECT a.FirstName, a.LastName
                FROM Employees a
                JOIN Addresses adr
                ON a.AddressID = adr.AddressID
                JOIN Towns t1
                ON adr.TownID = t1.TownID
                WHERE t1.Name = @townName
                OPEN nameCursor

                DECLARE @firstName VARCHAR(50), @lastName VARCHAR(50)
                FETCH NEXT FROM nameCursor INTO @firstName,  @lastName
                WHILE @@FETCH_STATUS = 0
                        BEGIN
                                SET @userNames = CONCAT(@userNames, @firstName, ' ',
@lastName, ', ')
                                FETCH NEXT FROM nameCursor
                                INTO @firstName,  @lastName
                        END
        CLOSE nameCursor
        DEALLOCATE nameCursor
                END
                SET @userNames = LEFT(@userNames, LEN(@userNames) - 1)
    PRINT @townName + ' -> ' + @userNames
    FETCH NEXT FROM empCursor
    INTO @townName
  END
CLOSE empCursor
DEALLOCATE empCursor

GO
```

10. Define a .NET aggregate function StrConcat that takes as input a sequence of strings and return a single string that consists of the input strings separated by ','. For example the following SQL statement should return a single string:

```
IF OBJECT_ID('dbo.concat') IS NOT NULL DROP Aggregate concat
GO

IF EXISTS (SELECT * FROM sys.assemblies WHERE name = 'concat_assembly')
       DROP assembly concat_assembly;
GO

CREATE Assembly concat_assembly
   AUTHORIZATION dbo
   FROM 'C:\Users\geri\Documents\SQL Server Management Studio\Projects\4. T-
SQL\Concatination.dll'
   WITH PERMISSION_SET = SAFE;
GO

CREATE AGGREGATE dbo.concat (

    @VALUE NVARCHAR(MAX)
  , @Delimiter NVARCHAR(4000)

) RETURNS NVARCHAR(MAX)
EXTERNAL Name concat_assembly.concat;
GO

--If execution of user code in the .NET Framework is disabled
sp_configure 'clr enabled', 1
GO
RECONFIGURE
GO

SELECT dbo.concat(FirstName + ' ' +LastName,', ')
   FROM Employees
GO
```