

ლაბორატორია 8

პროცედურები

SQL Server-ში შენახული პროცედურა (Stored Procedure) არის T-SQL ბრძანებების ნაკრები, რომელიც ინახება მონაცემთა ბაზაში. შენახული პროცედურა იღებს შემავალ და გამომავალ პარამეტრებს, ასრულებს შესაბამის SQL ოპერატორებს და აბრუნებს შედეგების კომპლექტს, ასეთის არსებობის შემთხვევაში.

ნაგულისხმევად, პირველი შესრულების შემდეგ ითვლება რომ შენახული პროცედურა შედგა. თუმცა იგი ინახება ბაზაში და წარმოადგენს შესრულების გეგმას, რომელიც გამოიყენება იგივე მოქმედების განმეორებით უფრო სწრაფი შესრულებისთვის.

შენახული პროცედურები ორი ტიპისაა:

მომხმარებლის მიერ განსაზღვრული პროცედურები (User-defined Procedures): მომხმარებლის მიერ განსაზღვრული შენახული პროცედურა იქმნება მომხმარებლის მიერ განსაზღვრულ მონაცემთა ბაზაში ან ნებისმიერი სისტემურ მონაცემთა ბაზაში, გარდა რესურსების მონაცემთა ბაზისა.

სისტემის პროცედურები (System procedures): სისტემის პროცედურები წარმოადგენს SQL Server-ის ნაწილს. ისინი ფიზიკურად ინახება რესურსების შიდა, ფარულ მონაცემთა ბაზაში და ლოგიკურად გამოტანილია ყველა მონაცემთა ბაზის **sys** სქემაში. სისტემაში შენახული პროცედურები იწყება პრეფიქსით **sp_**.

შენახული პროცედურის შექმნა

გამოიყენეთ CREATE განცხადება შენახული პროცედურის შესაქმნელად.

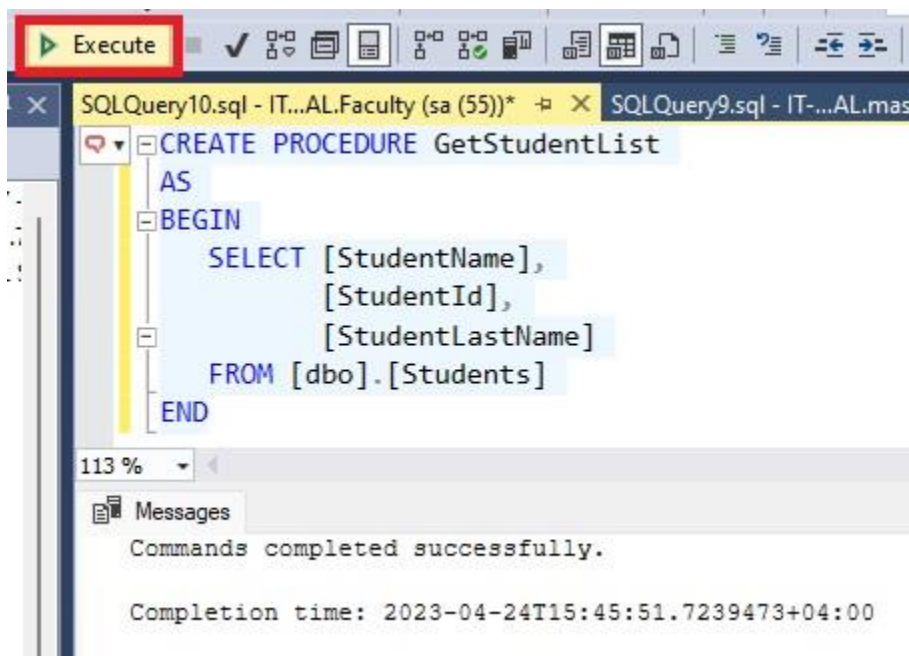
სინტაქსი:

```
CREATE [OR ALTER] {PROC | PROCEDURE} [schema_name.]
procedure_name ([@parameter data_type [ OUT | OUTPUT |
[READONLY]]
[ WITH <procedure_option> ]
[ FOR REPLICATION ]
AS
BEGIN
    sql_statements
END
```

შენახული პროცედურა შეიძლება შეიცავდეს ერთ ან მეტ ბრძანებას, როგორებიცაა მოთხოვნა, ჩასმა, განახლება ან წაშლა. ქვემოთ მოცემულია მარტივი შენახული პროცედურის მაგალითი, რომელიც აბრუნებს ჩანაწერებს ცხრილიდან SELECT მოთხოვნის გამოყენებით .

```
CREATE PROCEDURE GetStudentList
AS
BEGIN
    SELECT [StudentName],
           [StudentId],
           [StudentLastName]
    FROM [dbo].[Students]
END
```

გავუშვათ ზემოთ მოყვანილი T-SQL სკრიპტი შეკითხვის (Query) რედაქტორში, რათა შეადგინოთ და შექმნათ თქვენი პირველი შენახული პროცედურა მონაცემთა ბაზაში:



ზემოთ შექმნილი პროცედურა შეიძლება გამოვიდახოთ EXEC საბრძანებო სიტყვის გამოყენებით, როგორც ეს ნაჩვენებია ქვემოთ:

SQLQuery10.sql - IT...AL.Faculty (sa (55))* X SQLC

EXEC GetStudentList

113 %

Results Messages

	StudentName	StudentId	StudentLastName
1	ერეკლე	1000	ბურჯაძე
2	ნინო	1001	ქურდაძე
3	ანნა	1002	გაგუა
4	გიგო	1003	პაიჭიძე
5	ნუცა	1004	ურიდია
6	მაკა	1005	ჯოჯუა
7	დავითი	1007	ზვიადაძე
8	ნოკა	1008	ცინცაძე
9	ლელა	1009	დანელია

იმისთვის რომ პროცედურებმა შეასრულონ სხვადასხვა მოქმედებები მათ სჭირდებათ პარამეტრები, ზემოთ განხილული პროცედურა იყო უპარამეტრო, შესაბამისად იგი გამოძახების დროს არ ითხოვდა ჩვენგან რაიმე მონაცემს, თუმცა თუ მოვინდომებთ შევქმნათ პროცედურა რომელიც მაგალითად ჩაწერს მონაცემებს ცხრილში, აუცილებლად მოგვიწევს პარამეტრების სახით გავადავცეთ მონაცემები რომელთა ჩაწერაც გვინდა ცხრილში. პროცედურას შეიძლება ჰქონდეს ერთი, რამოდენიმე ან სულაც არცერთი პარამეტრი(როგორც უკვე ვნახეთ) თითოეულ პარამეტრს აქვს საკუთარი სახელი და პრეფიქსი @, ასევე აუცილებელია თითოეულ პარამეტრს ჰქონდეს საკუთარი ტიპი (ჩვენთვის უკვე ნაცნობი მონაცემთა ტიპებიდან). ასეთი პარამეტრები თავსდება პროცედურის თავში (ქუდში) და გამოყოფილია მძიმით.

შენახული პროცედურის პარამეტრები: შემავალი გამომავალი, გაჩუმების პრინციპით:

- შენახულ პროცედურას შეიძლება ჰქონდეს არცერთი, ერთი ან მეტი INPUT (შემავალი) და OUTPUT(გამომავალი) პარამეტრი.
- შენახულ პროცედურას შეიძლება ჰქონდეს მითითებული მაქსიმუმ 2100 პარამეტრი.
- თითოეულ პარამეტრს ენიჭება სახელი, მონაცემთა ტიპი და მიმართულება, (შემავალია თუ გამომავალი) თუ მიმართულება არ არის მითითებული, მაშინ ნაგულისხმევად პარამეტრი არის არის შემავალი.
- თქვენ შეგიძლიათ მიუთითოთ ნაგულისხმევი მნიშვნელობა პარამეტრებისთვის.
- შენახულ პროცედურებს შეუძლიათ დააბრუნონ მნიშვნელობა პროგრამაში, თუ პარამეტრი მითითებულია როგორც OUTPUT.

- პარამეტრის მნიშვნელობები უნდა იყოს მუდმივი ან ცვლადი. არ შეიძლება მნიშვნელობა იყოს ფუნქციის სახელი.
- პარამეტრის ცვლადები შეიძლება იყოს მომხმარებლის მიერ განსაზღვრული ან სისტემის ცვლადები, როგორცაა @spid

თუ გვსურს პროცედურის შექმნა რომელიც ჩვენს ნაცნობ Students ცხრილში შექმნის ახალ ჩანაწერს, ანუ გვსურს ახალი სტუდენტის დამატების პროცესის ავტომატიზაცია, აუცილებელია ასეთ პროცედურას როგორც მინიმუმ გადავცეთ აღნიშნული ცხრილის სავალდებულო ველების შესავსებად საჭირო ინფორმაცია: ჩვენს შემთხვევაში დაგვჭირდება პარამეტრები:

```
@StudentName NVARCHAR(30),
@StudentLastName NVARCHAR(30),
@email VARCHAR(30),
@DirectionId INT
```

სტუდენტის სახელი, გვარი, მეილი და მიმართულების ვალიდური (ბაზაში არსებული) ნომერი. გაითვალისწინეთ რომ ცვლადების სახელები შეიძლება შეარჩიოთ ნებისმიერი სასურველი წესით, ჩვენს შემთხვევაში ვირჩევთ შესავსები ველების იდენტურ დასახელებებს რათა შემდგომ პროცედურის ტანში ადვილად მოხდეს დიფერენცირება რომელი ცვლადი რომელი პარამეტრისთვისაა.

როგორც უკვე აღვნიშნეთ სახელები შეიძლება იყოს ნებისმიერი (გარდა დუბლირებულია) მაგრამ დამეთანხმებით ქვემოთ მოყვანილი სკრიპტის პარამეტრების იდენტიფიცირება შემდეგ კოდში ბევრად უფრო რთული იქნება:

```
@A NVARCHAR(30),
@B NVARCHAR(30),
@C VARCHAR(30),
@D INT
```

პარამეტრის სახელები

- შენახული პროცედურის პარამეტრების სახელები უნდა დაიწყოს ერთი @-ით.
- სახელი უნდა იყოს უნიკალური შენახული პროცედურის ფარგლებში.
- თუ პარამეტრის მნიშვნელობები გადაეცემა როგორც @Param1 = value1, @Param2 = value2, როგორც ნაჩვენებია ზემოთ მოცემულ მაგალითში, მაშინ პარამეტრების გადაცემა შესაძლებელია ნებისმიერი თანმიმდევრობით.
- თუ ერთი პარამეტრი მოწოდებულია როგორც @param1 = მნიშვნელობა, მაშინ ყველა პარამეტრი უნდა იყოს მიწოდებული იმავე წესით.

სახელებისგან განსხვავებით ცვლადების მონაცემთა ტიპის არჩევა სურვილისამებრ არ ხდება. აუცილებელია მონაცემთა ტიპი ემთხვეოდეს ცხრილში არსებულ ტიპს, ვინაიდან თუ ცხრილში მონაცემი მთელი რიცხვია, პარამეტრი რომელიც მას ენიჭება ასევე უნდა იყოს მთელი რიცხვი.

პარამეტრების დასრულების შემდგომ სულდება პროცედურის თავი და იწყება ტანი **AS** საბრძანებო სიტყვის შემდგომ. აქვე ვახსენოთ **BEGIN** და **END** ოპერატორები რომლებიც SQL-ში წარმოადგენენ {} ფიგურულ ფრჩხილებს შესაბამისად ქმნიან დირექტივების ბლოკს. როგორც ვიცით, ფიგურული ფრჩხილების მსგავსად თუ პროცედურის ტანში გვაქვს მხოლოდ ერთი ოპერატორი სავალდებულო არაა **BEGIN** და **END** ოპერატორების გამოყენება, მაგრამ თუ პროცედურა შედგება რამოდენიმე მოქმედებისგან აუცილებელია მათი **BEGIN** და **END** ოპერატორებით შემოსაზღვრა.

შემდეგი შენახული პროცედურა სვავს (Insert) მნიშვნელობებს ცხრილში Students:

```
CREATE PROCEDURE [dbo].[AddStudent]
```

```
    @StudentName NVARCHAR(30),  
    @StudentLastName NVARCHAR(30),  
    @Email VARCHAR(30),  
    @DirectionId INT
```

```
AS
```

```
BEGIN
```

```
    INSERT INTO [dbo].[Students]
```

```
    (  
        [StudentName],  
        [StudentLastName],  
        [Email],  
        [DirectionId]  
    )
```

```
VALUES
```

```
    (@StudentName,  
    @StudentLastName,  
    @Email,  
    @DirectionId);
```

```
END;
```

აღნიშნული შენახული პროცედურა შეიძლება გამოყენებულ იქნას ცხრილში მნიშვნელობების ჩასასმელად **INSERT** ბრძანების ნაცვლად. მნიშვნელობები გადაეცემა შენახულ პროცედურას პარამეტრების სახით. სიმბოლო **@** როგორც უკვე ვთქვით გამოიყენება როგორც პარამეტრის ცვლადების პრეფიქსი.

შეგვიძლია გამოვიძახოთ AddStudent შენახული პროცედურა კვლავ EXEC საკვანძო სიტყვის გამოყენებით, იმ განსხვავებით რომ ამჯერად ჩვენი პროცედურა შეიცავს პარამეტრებს და შესაბამისად მისი ყოველი გამოძახების დროს უნდა მოვახდინოთ აღნიშნული პარამეტრების ინიციალიზაცია:

```
EXEC [AddStudent]
  @StudentName =N'თეა'
, @StudentLastName = N'პაპავა'
, @Email = 'T.papava@gmail.com'
, @DirectionId = 1
```

პროცედურის შესრულებისას მიუთითეთ თითოეული პარამეტრი:



```
EXEC [AddStudent]
  @StudentName =N'თეა'
, @StudentLastName = N'პაპავა'
, @Email = 'T.papava@gmail.com'
, @DirectionId = 1
```

3 %

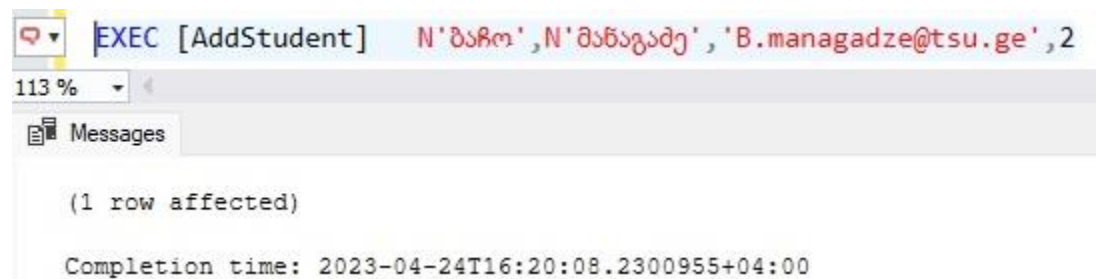
Messages

(1 row affected)

Completion time: 2023-04-24T16:17:05.2356284+04:00

თუ პროცედურის გამოძახებისას გადავცემთ ყველა პარამეტრს, და მათი გადაცემის თანმიმდევრობაც დაცულია, შეგვიძლია ცვლადი პარამეტრების სახელები აღარ მივუთითოთ:

```
EXEC [AddStudent] N'ბაზო',N'მანაგაძე', 'B.managadze@tsu.ge',2
```



```
EXEC [AddStudent] N'ბაზო',N'მანაგაძე', 'B.managadze@tsu.ge',2
```

113 %

Messages

(1 row affected)

Completion time: 2023-04-24T16:20:08.2300955+04:00

შენახული პროცედურის ნახვა

პროცედურების დასათვალირებლად გამოიყენეთ `sp_help` ან `sp_helptext` ბრძანებები:

sp_helptext AddStudent

```
SQLQuery10.sql - IT...AL.Faculty (sa (55))* -p X
sp_helptext AddStudent

113 %
Results Messages
Text
1 CREATE PROCEDURE [dbo].[AddStudent]
2 @StudentName NVARCHAR(30),
3 @StudentLastName NVARCHAR(30),
4 @Email VARCHAR(30),
5 @DirectionId INT
6 AS
7 BEGIN
8 INSERT INTO [dbo].[Students]
9 (
10 [StudentName],
11 [StudentLastName],
12 [Email],
13 [DirectionId]
14 )
15 VALUES
16 (@StudentName,
17 @StudentLastName,
18 @Email,
19 @DirectionId);
20 END;
```

sp_help AddStudent

SQLQuery10.sql - IT...AL.Faculty (sa (55))* -p X SQLQuery9.sql - IT...AL.master (sa (54)) SQLQuery189

sp_help AddStudent

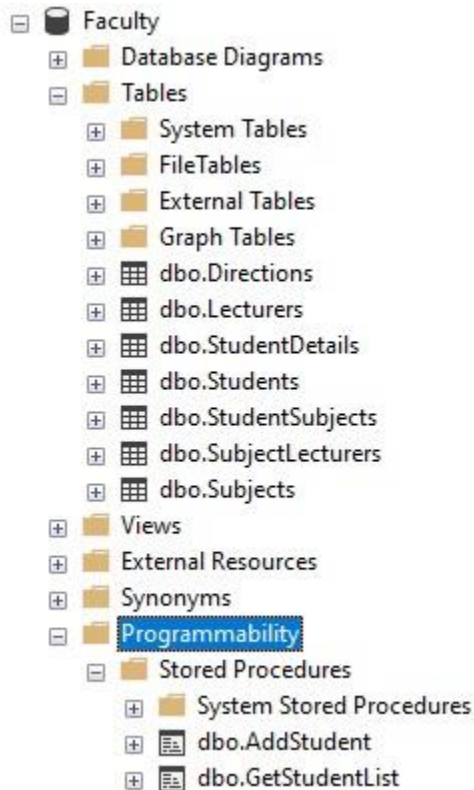
113 %

Name	Owner	Type	Created_datetime
AddStudent	dbo	stored procedure	2023-04-24 16:15:56.780

Parameter_name	Type	Length	Prec	Scale	Param_order	Collation
@StudentName	nvarchar	60	30	NULL	1	SQL_Latin1_General_CP1_CI_AS
@StudentLastName	nvarchar	60	30	NULL	2	SQL_Latin1_General_CP1_CI_AS
@Email	varchar	30	30	NULL	3	SQL_Latin1_General_CP1_CI_AS
@DirectionId	int	4	10	0	4	NULL

შედეგად შესაძლებელია დაათვალიეროთ თავად პროცედურის სკრიპტი ნახოთ როდის შეიქმნა, რა პარამეტრები აქვს და ა.შ

ყველა შენახული პროცედურა განთავსებულია მონაცემთა ბაზის Programmability > Stored Procedures საქალაქდეში.



შენახული პროცედურის შეცვლა

ALTER PROCEDURE ბრძანება გამოიყენება შენახული პროცედურის შესაცვლელად.

```
ALTER PROCEDURE [dbo].[GetStudentList]
AS
BEGIN
    SELECT [StudentName],
           [StudentId]+' '+[StudentLastName] AS FullName,
           [Email]
    FROM [dbo].[Students]
END
```

ALTER PROCEDURE ბრძანებით ხდება პროცედურის ტანის ნებისმიერი ცვლილება. გაითვალისწინეთ რომ უნდა დაწეროთ არა მხოლოდ შესაცვლელი ნაწილი არამედ პროცედურის სრული ტანი ხელახლა (ცვლილების ჩათვლით). ხოლო პროცედურის

ისევე როგორ ბაზის სხვა ობიექტების სახელის გადასარქმევად გამოიყენება სისტემური პროცედურა:

```
sp_rename 'GetStudentList', 'GetStudents'
```

შენახული პროცედურის წაშლა

DROP PROCEDURE ბრძანება გამოიყენება შენახული პროცედურის წასაშლელად.

```
DROP PROCEDURE dbo.GetStudents;
```

OUTPUT პარამეტრები

OUTPUT პარამეტრი გამოიყენება, როდესაც გსურთ დააბრუნოთ გარკვეული მნიშვნელობა შენახული პროცედურისგან. პროცედურის გამოძახებისას პროგრამამ ასევე უნდა გამოიყენოს OUTPUT საკვანძო სიტყვა.

შემდეგი შენახული პროცედურა შეიცავს INPUT და OUTPUT პარამეტრს:

```
ALTER PROCEDURE [dbo].[AddStudent]
```

```
    @StudentName NVARCHAR(30),  
    @StudentLastName NVARCHAR(30),  
    @Email VARCHAR(30),  
    @DirectionId INT,  
    @IsAdded BIT output
```

```
AS
```

```
BEGIN
```

```
    INSERT INTO [dbo].[Students]
```

```
    (  
        [StudentName],  
        [StudentLastName],  
        [Email],  
        [DirectionId]
```

```
)
```

```
VALUES
```

```
    (@StudentName,  
     @StudentLastName,  
     @Email,  
     @DirectionId);
```

```
    SET @IsAdded=1
```

```
END;
```

ზემოთ შენახულ პროცედურაში @IsAdded არის OUTPUT პარამეტრი. მას მნიშვნელობა მიენიჭება პროცედურის შესრულებისას და დაბრუნდება გამოძახების განცხადებაში. შემდეგი კოდი გამოიძახებს პროცედურას OUTPUT პარამეტრით:

```
DECLARE @IsAdded bit
```

```
EXEC [AddStudent]
```

```
  @StudentName = N'გიორგი'  
,@StudentLastName = N'მაისურაძე'  
,@Email = 'G.maisuradze@gmail.com'  
,@DirectionId = 2
```

```
,@IsAdded=@IsAdded OUTPUT
```

```
PRINT @IsAdded
```

თუ დავაკვირდებით პროცედურის გამოძახების ტანს, მასში გამოყენებული ახალი დირექტივა: **DECLARE @IsAdded int**

ვინდაინ ჩვენ გვჭირდება ცვლადი რომელიც არ წარმოადგენს პროცედურის პარამეტრს, და დამატებით უნდა შევქმნათ ახალი ცვლადი ვიყენებთ საბრძანებო სიტყვას **DECLARE**.

ცვლადზე მნიშვნელობის მინიჭება შესაძლებელია SET და SELECT ბრძანებებით. სიტყვა **DECLARE** -ის საშუალებით შექმნილი ცვლადის სიცოხლისუნარიანობა განისაზღვრება ერთი ტრანზაქციით, შესაბამისად ერთი exec-ით და იგი პროცედურის გაშვების ყოველ ჯერზე თავიდან იქმნება და ქრება პროცედურის დასრულებისთანავე.

ჩვენს შემთხვევაში შეიქმნა ცვლადი რომელიც ინახავს პროცედურიდან დაბრუნებულ პარამეტრს. ხოლო **PRINT @IsAdded** ბრძანებით ხდება მისი გამოტანა ეკრანზე.

ნაგულისმები პარამეტრები

SQL Server გაძლევთ საშუალებას მიუთითოთ პარამეტრების ნაგულისხმევი მნიშვნელობები. შესაბამისად გამოტოვოთ სასურველი პარამეტრები, რომლებსაც აქვთ ნაგულისხმევი მნიშვნელობები შენახული პროცედურის გამოძახებისას. ნაგულისხმევი მნიშვნელობა გამოიყენება მაშინ, როდესაც პარამეტრს არ გადაეცემა მნიშვნელობა ან როდესაც DEFAULT საკვანძო სიტყვა მითითებულია, როგორც მნიშვნელობა პროცედურის გამოძახებაში.

მიუთითეთ ნაგულისხმევი მნიშვნელობა, როდესაც აცხადებთ პარამეტრებს, როგორც ეს ნაჩვენებია ქვემოთ:

```
ALTER PROCEDURE [dbo].[AddStudent]
```

```

@StudentName NVARCHAR(30),
@StudentLastName NVARCHAR(30),
@email VARCHAR(30),
@DirectionId INT=1,
@IsAdded BIT output

AS
BEGIN
    INSERT INTO [dbo].[Students]
    (
        [StudentName],
        [StudentLastName],
        [Email],
        [DirectionId]
    )
    VALUES
    (@StudentName,
    @StudentLastName,
    @Email,
    @DirectionId);

    SET @IsAdded=1
END;

```

როგორც ხედავთ პარამეტრ@DirectionId-ს გაჩუმების პრინციპით მითითებული აქვს მნიშვნელობა 1, შესაბამისად თუ მას არ გადავცემთ პროცედურის გამოძახებისას, ავტომატურად მიენიჭება მიმართულების ნომერი -1.

```
DECLARE @IsAdded bit
```

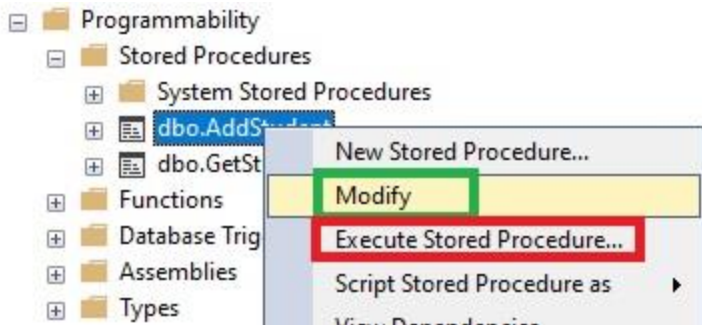
```

EXEC [AddStudent]
    @StudentName =N'გიორგი'
    ,@StudentLastName = N'მაისურაძე'
    ,@Email = 'G.maisuradze@gmail.com'
    ,@IsAdded=@IsAdded OUTPUT

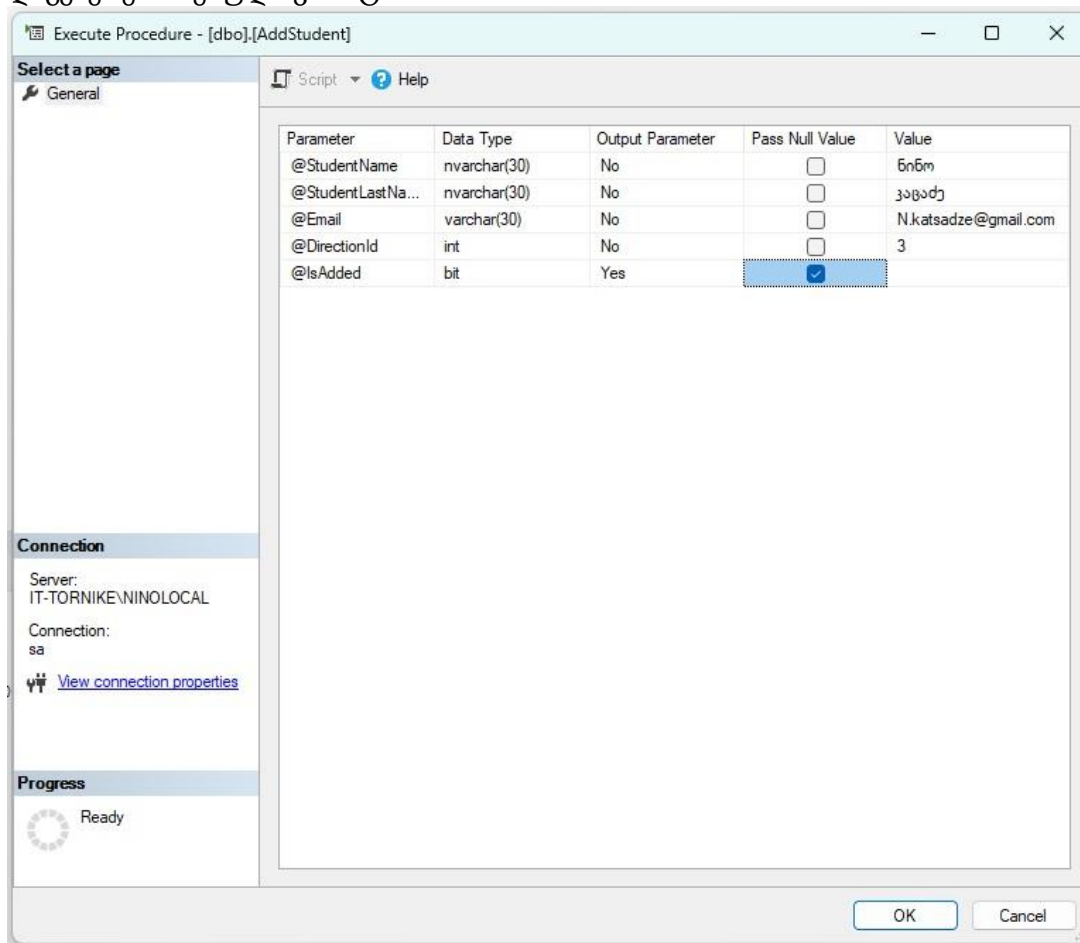
```

```
PRINT @IsAdded
```

პროცედურების გამოძახება ასევე შესაძლებელია დიზაინ რეჟიმში პროცედურაზე მარჯვენა ღილაკის და Execute ბრძანების გამოყენებით, ხოლო უკვე არსებული პროცედურის მოდიფიკაცია ასევე შესაძლებელია მასზე მარჯვენა ღილაკის დაჭერით და Modify ბრძანების ამორჩევით.



Execute ის გაშვების შემთხვევაში მიიღებთ ფანჯარას რომელშიც უნდა ჩაწეროთ თითოეული პარამეტრი გარდა გამოძახებისა, ასევე შესრულების შემდგომ მიიღებთ გამოძახების დაგენერირებულ სკრიპტს.



```
SQLQuery15.sql - IT...AL.Faculty (sa (63)) -> X SQLQuery13.sql - IT...AL.Faculty
USE [Faculty]
GO

--DECLARE @return_value int,
--        @IsAdded bit

--EXEC    @return_value = [dbo].[AddStudent]
--        @StudentName = N'ნინო',
--        @StudentLastName = N'კაცაძე',
--        @Email = N'N.katsadze@gmail.com',
--        @DirectionId = 3,
--        @IsAdded = @IsAdded OUTPUT

--SELECT  @IsAdded as N'@IsAdded'

--SELECT  'Return Value' = @return_value

GO
```

113 %

Results Messages

	@IsAdded
1	1

შეცდომების დამუშავება შენახულ პროცედურებში

როგორც უკვე აღვნიშნეთ პროცედურა წარმოადგენს ავტომატიზირებულ სკრიპტს, რომელიც უნდა მუშაობდეს ჩვენგან დამოუკიდებლად მაგალითად აპლიკაციაში ლილაკის დაჭერის ან საიტზე სარეგისტრაციო ფორმის შევსების შემდგომ, შესაბამისად აღნიშნული სკრიპტები უნდა დავეწეროთ ძალიან ფრთხილად რომ მაქსიმალურად ავირიდოთ თავიდან შეცდომები, და მათი არსებობის შემთხვევაში სწორად გავუმკლავდეთ მათ.

იმისთვის რომ სწორად დავეწეროთ კოდი საჭიროა გავთვალოთ ყველა შესაძლო შემთხვევა, იდეალურ შემთხვევაში ყველა არავალიდურ მონაცემებს დავხვდებით გამზადებული შესაბამისი „შეცდომის შეტყობინებით“ (Error Message) თუმცა ყოველთვის იარსებებს ალბათობა გუთვალისწინებელი შეცდომის, რა დროსაც შეცდომას დაგვიბრუნებს სისტემა სპეციალური ფუნქციებით, როგორიცაა ERROR_MESSAGE(), ERROR_NUMBER(), ERROR_STATE(), ERROR_SEVERITY().

აღნიშნული პრობლემის საპასუხოდ SQL Server-ში არსებობს TRY..CATCH ბლოკი რომელიც გამოიყენება შეცდომების დასამუშავებლად. ძირითადი T-SQL ბრძანებების ჯგუფი იწერება TRY ბლოკში. თუ TRY ბლოკში მოხდა შეცდომა, კონტროლი გადაეცემა CATCH ბლოკს, რომელსაც ექნება SQL განცხადებების სხვა ნაკრები შეცდომის დასამუშავებლად.

მაგალითისთვის განვიხილოთ ისევ სტუდენტის დამატების მოქმედება, რომელიც შეიძლება სინამდვილეში სულაც არ შედგებოდეს ერთი Insert ოპერაციისაგან, მაგალითად სისტემაში დავამატოთ სტუდენტის ცხრილის სრული ლოგირების ცხრილი StudentsLog რათა ყოველთვის ვიცოდეთ რომელი ჩანაწერი ვის მიერ დაემატა, განახლდა თუ წაიშალა და როდის. თუ სტუდენტის ცხრილი გამოიყურება ასე:

```
CREATE TABLE [dbo].[Students](
    [StudentId] [INT] IDENTITY(1000,1) NOT NULL PRIMARY key,
    [StudentName] [NVARCHAR](30) NOT NULL,
    [StudentLastName] [NVARCHAR](30) NOT NULL,
    [Email] [VARCHAR](30) NULL,
    [DirectionId] [INT] NOT NULL,
    [Password] [VARBINARY](250) NULL)
```

მისი ლოგის ცხრილი შეიძლება გამოიყურებოდეს შემდეგნაირად:

```
CREATE TABLE [dbo].[StudentsLog](
    [StudentsLogId] [int] IDENTITY(1,1) NOT NULL PRIMARY KEY,
    [LogOperation] [varchar](50) NULL,
    [UserId] [int] NULL,
    [RecordDate] [datetime] NULL,
    [StudentId] [int] NULL,
    [StudentName] [nvarchar](30) NULL,
    [StudentLastName] [nvarchar](30) NULL,
    [Email] [varchar](30) NULL,
    [DirectionId] [int] NULL,
    [Password] [varbinary](250) NULL)
```

go

ვინაიდან მასში დამატებული იქნება ლოგის ჩანაწერის ნომერი (StudentsLogId), ლოგის ოპერაციის დასახელება (LogOperation), ოპერაციის შემსრულებელი მომხმარებლის ნომერი (UserId) (რომელიც სტანდარტულად ყოველთვის ვიცით სისტემაში ავტორიზაციის შემდგომ მუშაობის პროცესში) და ოპერაციის შესრულების თარიღი (RecordDate).

აღნიშნულ შემთხვევაში პროცედურის ტანი შეიცვლება ვინაიდან მასში უნდა მოხდეს არა ერთი არამედ ორი ცხრილის შევსება:

```
ALTER procedure [dbo].[AddStudent]
```

```
@StudentName nvarchar(30),
@StudentLastName nvarchar(30),
```



```
@Email varchar(30),
@DirectionId int,
@UserId int
as
begin

DECLARE @StudentId int
```

```
insert into [dbo].[Students]
([StudentName],
[StudentLastName],
[Email],
[DirectionId],
[Password])
values
(@StudentName,
@StudentLastName,
@Email,
@DirectionId,
PWDENCRYPT('123456'))
)

set @StudentId=SCOPE_IDENTITY()
```

```
insert into [dbo].[StudentsLog]
(
[LogOperation],
[UserId],
[RecordDate],
[StudentId],
[StudentName],
[StudentLastName],
[Email],
[DirectionId],
[Password])
values
(
'Insert',
@UserId,
getdate(),
@StudentId,
@StudentName,
@StudentLastName,
@Email,
@DirectionId,
PWDENCRYPT('123456'))
)
```

```
end  
go
```

რაც შეეხება **SCOPE_IDENTITY()** სისტემურ ფუნქციას, იწერება ჩასმის Insert ოპერაციის შემდგომ და თუ კი მოხდა ავტომატური გადანომვრის მქონე ცვლადის (Identity) მნიშვნელობის დაგენერირება, დაგვიბრუნებს მას. მარტივად რომ ვთქვათ სტუდენტის დამატების შემდეგ მას მიენიჭა სისტემაში თავისუფალი მომდევნო ნომერი (რომელიც ჩვენ არ ვიცით) ხოლო ბრძანება **set @StudentId=SCOPE_IDENTITY()** წინასწარ გამზადებულ ცვლადში ჩაწერს ამ ნომერს.

ასევე სისტემური ფუნქცია **PWDENCRYPT('123456')** დასკრიპტავს გადაცემულ სტრიქონს და ჩაწერს პაროლის ველში. შესაბამისად რეგისტრაციის დროს ყველას ენიჭება ერთი და იგივე პაროლი რომლის შეცვლაც შეეძლება შემდგომ. გავითვალისწინოთ რომ პაროლების შენახვა დაუსკრიპტავი ფორმით დაუშვებელია!

რომ დავუბრუნდეთ მთავარ კოდს, როგორც ვთქვით მიშვნელოვანია წინასწარ გავითვალისწინოთ ყველა შეცდომა და მომხმარებელს დავუბრუნოთ შესაბამისი შეტყობინება, მაგრამ თუ მაინც მოხდა რაიმე გაუთვალისწინებელი საჭიროა გვქონდეს TRY..CATCH ბლოკი რათა ძირითად კოდში რაიმე შეცდომის მოხდენის შემთხვევაში დავიჭიროთ ეს შეცდომები, შევწყვიტოთ მოქმედები TRY ბლოკში და გადავიდეთ CATCH-ზე სადაც გვიწერია კოდი რომელიც გვინდა რომ შესრულდეს შეცდომების შემთხვევაში.

დავამატოთ ლოგის ცხრილში Error nvarchar(max) ველი, შეცდომების შესანახად:

```
ALTER TABLE [dbo].[StudentsLog]
```

```
ADD Error nvarchar(max)
```

ჩავასწოროთ ჩვენი პროცედურა დავამატოთ რამოდენიმე დასაბრუნებელი პარამეტრი, მათ შორის შეტყობინება შეცდომაზე :

```
ALTER procedure [dbo].[AddStudent]
```

```
@StudentName nvarchar(30),  
@StudentLastName nvarchar(30),  
@Email varchar(30),  
@DirectionId int,  
@StudentId INT OUTPUT,  
@IsAdded BIT OUTPUT,  
@Error NVARCHAR (MAX) OUTPUT
```

```
as
```

```

BEGIN TRY

insert into [dbo].[Students]
([StudentName],
[StudentLastName],
[Email],
[DirectionId],
[Password])
values
(@StudentName,
@StudentLastName,
@Email,
@DirectionId,
PWDENCRYPT('123456'))

SET @StudentId=SCOPE_IDENTITY()
SET @IsAdded=1

insert into [dbo].[StudentsLog]
(
[LogOperation],
[UserId],
[RecordDate],
[StudentId],
[StudentName],
[StudentLastName],
[Email],
[DirectionId],
[Password])
values
(
'Insert',
@UserId,
getdate(),
@StudentId,
@StudentName,
@StudentLastName,
@Email,
@DirectionId,
PWDENCRYPT('123456'))

)

END TRY

BEGIN CATCH
SET @IsAdded=0
SET @Error=ERROR_MESSAGE()

insert into [dbo].[StudentsLog]
(
[LogOperation],
[UserId],
[RecordDate],
[StudentId],
[StudentName],

```

```

[StudentLastName],
[Email],
[DirectionId],
[Password],
[Error])
values
(
'Insert Fail',
@UserId,
getdate(),
@StudentId,
@StudentName,
@StudentLastName,
@Email,
@DirectionId,
PWDENCRYPT('123456')
,@Error)

END CATCH

go

```

ძირითად კოდში შეცდომის შემთხვევაში CATCH ბლოკში მაინც მოხდება შეცდომის შესახებ ინფორმაციის ლოგირება ასევე დასაბრუნებელი ცვლადები პროგრამას დაუბრუნებენ პასუხს შეცდომის შესახებ. გამოვიძახოთ პროცედურა გადავცეთ არარსებული მიმართულების ნომერი და ვნახოთ შედეგი:

```

DECLARE
    @StudentId int,
    @IsAdded bit,
    @Error nvarchar(max)

EXEC [dbo].[AddStudent]
    @StudentName = N'ნიკა',
    @StudentLastName = N'რუხაძე',
    @Email = N'N.rukhadze@gmail.com',
    @DirectionId = 12,
    @UserId = 1,
    @StudentId = @StudentId OUTPUT,
    @IsAdded = @IsAdded OUTPUT,
    @Error = @Error OUTPUT

SELECT
    @StudentId as N'@StudentId',
    @IsAdded as N'@IsAdded',
    @Error as N'@Error'

```

აღნიშნული შეცდომის შემდეგ დავათვალიეროთ ძირითადი და ლოგირების ცხრილები:

```

SELECT * FROM [dbo].[Students];

```

```
SELECT * FROM [dbo].[StudentsLog];
```

როგორც უკვე აღვნიშნეთ მიშენელოვანია წინასწარ გავითვალისწინოთ ყველა შეცდომა და მომხმარებელს დავუბრუნოთ შესაბამისი შეტყობინება, მაგალითად თუ შეუძლებელია ერთი და იმავე მეილით ორი სტუდენტის რეგისტრაცია სასურველია ჩამატებამდე შემოწმდეს მეილის უნიკალურობა და მომხმარებელმა მიიღოს შესაბამისი შეცდომის შესახებ შეტყობინება. ჩავამატოთ If ლოგიკური შემოწმების პირობა პროცედურაში:

```
ALTER PROCEDURE [dbo].[AddStudent]
    @StudentName NVARCHAR(30),
    @StudentLastName NVARCHAR(30),
    @Email VARCHAR(30),
    @DirectionId INT = 1,
    @UserId INT,
    @StudentId INT OUTPUT,
    @IsAdded BIT OUTPUT,
    @Error NVARCHAR(MAX) OUTPUT
AS
BEGIN TRY

    IF EXISTS (SELECT [Email] FROM [dbo].[Students] WHERE [Email] = @Email)
    BEGIN
        SET @IsAdded = 0;
        SET @Error = N'სტუდენტი ასეთი მეილით უკვე დარეგისტრირებულია სისტემაში.';
        RAISERROR(@Error, 16, 1);

    END;

    INSERT INTO [dbo].[Students]
    (
        [StudentName],
        [StudentLastName],
        [Email],
        [DirectionId],
        [Password]
    )
    VALUES
    (@StudentName, @StudentLastName, @Email, @DirectionId, PWDENCRYPT('123456'));

    SET @StudentId = SCOPE_IDENTITY();
    SET @IsAdded = 1;

    INSERT INTO [dbo].[StudentsLog]
    (
        [LogOperation],
        [UserId],
        [RecordDate],
        [StudentId],
        [StudentName],
        [StudentLastName],
        [Email],
        [DirectionId],
    )
```

```

        [Password]
    )
    VALUES
    ('Insert', @UserId, GETDATE(), @StudentId, @StudentName, @StudentLastName, @Email,
@DirectionId,
    PWDENCRYPT('123456'));

END TRY
BEGIN CATCH
    SET @IsAdded = 0;
    SET @Error = ERROR_MESSAGE();

    INSERT INTO [dbo].[StudentsLog]
    (
        [LogOperation],
        [UserId],
        [RecordDate],
        [StudentId],
        [StudentName],
        [StudentLastName],
        [Email],
        [DirectionId],
        [Password],
        [Error]
    )
    VALUES
    ('Insert Fail', @UserId, GETDATE(), @StudentId, @StudentName, @StudentLastName,
@Email, @DirectionId,
    PWDENCRYPT('123456'), @Error);

END CATCH;

GO

```

გავნიხილოთ **IF EXISTS** პირობა, თუ პირობის მერე მომავალი მოთხოვნა (select) დაბრუნდება არაცარიელი, ჩვენს შემთხვევაში იპოვის ჩანაწერს იმავე მეილით რა მეილის მინიჭებასაც ახლა ვაპირებთ ახალი სტუდენტისთვის, გამოდის რომ მეილი დუბლირდება, უნდა შეწყდეს დამატების ოპერაცია და გამოვიდეს შეცდომა. ასეც მოვიქცევით, თუ შემოწმების პირობა ჭეშმარიტია **IF**-ის ტანში, რომელი შემოსაზღვრულია **BEGIN** და **END** ოპერატორებით ვწერთ შეცდომის ტექსტს შესაბამის **@Error** ცვლადში და ხელოვნურად წარმოვქმნით შეცდომას **RAISERROR** ფუნქციით. შეცდომის წარმოქმნისთანავე **TRY** ბლოკი „ავარიულად“ წყვეტს მოქმედებას და იწყება **CATCH** ბლოკის შესრულება სადაც ვახდენთ შეცდომის შესახებ ინფორმაციის შენახვას ლოგირების ცხრილში.თუ დავაკვირდებით კოდს, არ დაგვჭირდა განშტოების **IF**-ოპერატორისთვის **ELSE** პირობის დაწერა, **TRY..CATCH** ბლოკის თავისებურებიდან გამომდინარე ავტომატურად შეწყდა შეცდომის წარმოქმნისთანავე მოქმედება და აღარ შესრულდა დანარჩენი შევსების ოპერატორები.

ხელახლა გამვიდახოთ პროცედურა და გადავცეთ ერთი და იგივე მეილი ორჯერ:

```
DECLARE
    @StudentId int,
    @IsAdded bit,
    @Error nvarchar(max)

EXEC [dbo].[AddStudent]
    @StudentName = N'ნიკა',
    @StudentLastName = N'რუხაძე',
    @Email = N'N.rukhadze@gmail.com',
    @DirectionId = 2,
    @UserId = 1,
    @StudentId = @StudentId OUTPUT,
    @IsAdded = @IsAdded OUTPUT,
    @Error = @Error OUTPUT

SELECT    @StudentId as N'@StudentId',
          @IsAdded as N'@IsAdded',
          @Error as N'@Error'
```

შემდეგ კვლავ დავათვალიეროთ ძირითადი და ლოგირების ცხრილები:

```
SELECT * FROM [dbo].[Students];
SELECT * FROM [dbo].[StudentsLog];
```

ვინაიდან TRY ბლოკის შესრულება წყდება შეცდომის მოხდენისას, შესაძლებელია რომ ოპერაციათა ნაწილი შესრულდეს ხოლო ნაწილი არა. მაგალითად ჩვენი პროცედურის შემთხვევაში ერთი ჩასმა შესრულდეს ხოლო მეორე ვერა, რაც გამოიწვევს პროგრამის ხარვეზულ მუშაობას ვინაიდან თავიდანვე პროცედურის შექმნისას მთავარი მიზანია მთლიანი სკრიპტის ერთად შესრულება და არა ნაწილ-ნაწილ.

მაგალითად განვიხილოთ რაიმე რეგისტრაციის ფორმა სადაც შეიძლება ერთ ფორმაში ივსებოდეს რამოდენიმე ცხრილი, ძირითადი ცხრილი, დეტალურის ცხრილი და ასევე ფაილი იტვირთებოდეს და ამ დროს ამ სამიდან რომელიმე შევსება დასრულდა წარუმატებლად, რა გამოდის, რეგისტრაცია გავიარეთ ნაწილობრივად? ვერც თავიდან ვრეგისტრირებით იმიტომ რომ ერთ ცხრილში უკვე არის ჩვენზე ინფორმაცია და დუბლირდება ხელახლა ცდისას, ხოლო დანარჩენ ორ ცხრილში ვერ მოხდა შევსება და ავტორიზაციასაც ვეღარ გავდივარ? აღნიშნული პრობლემის თავიდან ასაცილებლად საჭიროა რომ პროცედურის ტანი ან შესრულდეს მთლიანად ან საერთოდ არ შესრულდეს და დაგვიბრუნდეს შეცდომა. ერთი შეხედვით ბოლო სკრიპტში გადავჭერთ სტანდარტული შეცდომების შემთხვევები მაგრამ როგორ გავიგოთ სად მოხდა შეცდომა? რეალურად არ არის საჭრო ვიცოდეთ რომელი ცხრილი შეივსო/შეიცვალა და რომელი არა, უბრალოდ

უნდა დავამატოთ ახალი ლოგიკა. რომლის მიხედვითაც თუკი პროცედურაში მოხდა შეცდომა მთლიანი სკრიპტი „დაბრუნდება უკან“ და წაიშლება რაც კი ცვლილებები შევიდა ცხრილებში კონკრეტული პროცედურის შესრულებისას. გამოდის თუკი რომელიმე ცხრილში რამე ჩაიწერა ან შეიცვალა უნდა დაბრუნდეს იმ მდგომარეობაზე რაც პროცედურის დაწყებამდე იყო. ამისათვის უნდა დავიჭიროთ საიდან დაიწყო პროცედურის ტანი და სად დამთავრდა, გავაერთიანოთ რამოდენიმე მოქმედება ერთ მოქმედებაში(ტრანზაქციაში) და შემდეგ დავაბრუნოთ აღნიშნული ტრანზაქცია „უკან“ (ROLLBACK).

SQL Server Transaction

ტრანზაქცია SQL Server-ში არის მონაცემთა ბაზაში ერთი ან რამდენიმე ამოცანის შესასრულებლად გათვალისწინებული ბრძანებების ან/და მოთხოვნების თანმიმდევრული ჯგუფი . თითოეული ტრანზაქციას შეიძლება შედგებოდეს წაკითხვის, ჩაწერის, განახლების ან წაშლის ერთი ოპერაციის ან ყველა ამ ოპერაციების კომბინაციისაგან. ყოველი ტრანზაქციისთვის აუცილებელია:

- ყველა მოქმედება წარმატებულია და ხდება ტრანზაქციის შესრულება/დასრულება (Commit).
- შეცდომის შემთხვევაში ყველა ცვლილება უქმდება და ტრანზაქცია უკან დაბრუნდება (Rollback).

სტანდარტულად Insert, Update ან Delete -ის დროს სერვერი იწყებს ავტოტრანზაქციებს, სადაც თითო მოქმედება წარმოადგენს თითო ტრანზაქციას. ტრანზაქციის მექანიკურად შესაქმნელად უნდა გამოვიყენოთ ბრძანებები BEGIN TRANSACTION ან BEGIN TRAN და დავასრულოთ ბრძანებებით COMMIT ან ROLLBACK. თუკი ტრანზაქციის დასაწყისიდან COMMIT მდე შეცდომა არ დაფიქსირდა ითვლება რომ ტრანზაქცია წარმატებით დასრულდა.

განვიხილოთ ჩვენი პროცედურის სკრიპტი ტრანზაქციით:

```
ALTER PROCEDURE [dbo].[AddStudent]
    @StudentName NVARCHAR(30),
    @StudentLastName NVARCHAR(30),
    @Email VARCHAR(30),
    @DirectionId INT = 1,
    @UserId INT,
    @StudentId INT OUTPUT,
    @IsAdded BIT OUTPUT,
    @Error NVARCHAR(MAX) OUTPUT
AS
BEGIN TRY
    BEGIN TRANSACTION

    IF EXISTS (SELECT [Email] FROM [dbo].[Students] WHERE [Email] = @Email)
    BEGIN
        SET @IsAdded = 0;
```

```

SET @Error = N'სტუდენტი ასეთი მეილით უკვე დარეგისტრირებულია სისტემაში.';
RAISERROR(@Error, 16, 1);

END;

INSERT INTO [dbo].[Students]
(
    [StudentName],
    [StudentLastName],
    [Email],
    [DirectionId],
    [Password]
)
VALUES
(@StudentName, @StudentLastName, @Email, @DirectionId, PWDENCRYPT('123456'));

SET @StudentId = SCOPE_IDENTITY();
SET @IsAdded = 1;

INSERT INTO [dbo].[StudentsLog]
(
    [LogOperation],
    [UserId],
    [RecordDate],
    [StudentId],
    [StudentName],
    [StudentLastName],
    [Email],
    [DirectionId],
    [Password]
)
VALUES
('Insert', @UserId, GETDATE(), @StudentId, @StudentName, @StudentLastName, @Email,
@DirectionId,
PWDENCRYPT('123456'));
COMMIT
END TRY
BEGIN CATCH
    ROLLBACK TRANSACTION
    SET @IsAdded = 0;
    SET @Error = ERROR_MESSAGE();

    INSERT INTO [dbo].[StudentsLog]
    (
        [LogOperation],
        [UserId],
        [RecordDate],
        [StudentId],
        [StudentName],
        [StudentLastName],
        [Email],
        [DirectionId],
        [Password],
        [Error]
    )
VALUES

```

```
    ('Insert Fail', @UserId, GETDATE(), @StudentId, @StudentName, @StudentLastName,  
@Email, @DirectionId,  
    PWDENCRYPT('123456'), @Error);
```

```
END CATCH;
```

```
GO
```

TRY ბლოკის დასაწყისშივე იხსნება ტრანზაქცია BEGIN TRANSACTION ბრძანებით და იხურება END TRY ბლოკის დასასრულში, ბრძანებით COMMIT. ეს ხდება რათა TRY -ს ტანი სერვერმა აღიქვას როგორც ერთი მთლიანი მოქმედება და იცოდეს რომ BEGIN TRANSACTION -დან COMMIT სიტყვამდე ყველა მოქმედება უნდა დასრულდეს წარმატებით რათა ჩაითვალოს რომ ტრანზაქცია შესრულდა. ახლა უკვე ჩვენი შევსების ოპერაციები სერვერისთვის აღარაა ცალცლაკე დამოუკიდებელი მოქმედებები. როგორც ვიცით თუ TRY ბლოკი დასრულდა უშეცდომოთ (გამოდის რომ BEGIN TRANSACTION -დან COMMIT სიტყვამდე არ მოხდა შეცდომა) ოპერაცია ითვლება წარმატებულად და აღარ სრულდება CATCH ბლოკი. მაგრამ ნებისმიერ შეცდომის შემთხვევაში წყდება TRY ბლოკი და შესაბამისად ტრანზაქცია, მართვა გადაეცემა CATCH ბლოკს , რომელშიც ვხედავ ბრძანებას ROLLBACK TRANSACTION.

ROLLBACK TRANSACTION ის შესრულებისას ტრანზაქციის შემადგენელი ყველა მოქმედება დაბრუნდება უკან. ზემოთ აღნიშნული პრობლემაც ნაწილობრივი რეგისტრაციის თაობაზე მოგვარდება ავტომატურად რადგან თუ ერთი ცხრილი შეივსო ხოლო დანარჩენებში მოხდა შეცდომა იმ ერთიდანაც ამოიშლება ჩანაწერი და დავიწყებთ რეგისტრაციის ხელახალ მცდელობას.