

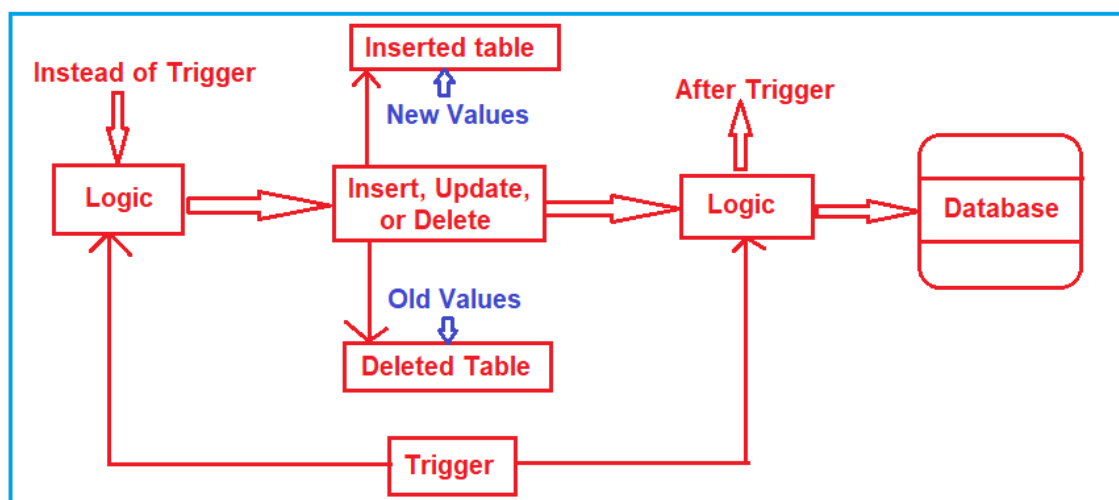
## Topic1: Triggers in SQL Server with Examples (from: [dotnettutorials.net/lesson/dml-triggers-in-sql-server/](https://dotnettutorials.net/lesson/dml-triggers-in-sql-server/))

Discuss the below topics:

1. What are Triggers in SQL Server?
2. Types of Triggers in SQL Server
3. What are DML Triggers in SQL Server?
4. Examples to understand DML Trigger
5. Where the Triggers are Created in SQL Server?
6. Why do we need DML Triggers in SQL Server?
7. Multiple examples to understand the above concepts

### What are Triggers in SQL Server?

Triggers are nothing but they are logic's like stored procedures that can be executed automatically before the Insert, Update or Delete happens in a table or after the Insert, Update, or Delete happens in a table. In simple words, we can say that, if you want to execute some pre-processing or post-processing logic before or after the Insert, Update, or Delete in a table then you need to use Triggers in SQL Server.



There are two types of triggers. They are as follows:

1. **Instead of Triggers:** The Instead Of triggers are going to be executed instead of the corresponding DML operations. That means instead of the DML operations such as Insert, Update, and Delete, the Instead Of triggers are going to be executed.
2. **After Triggers:** The After Triggers fires in SQL Server execute after the triggering action. That means once the DML statement (such as Insert, Update, and Delete) completes its execution, this trigger is going to be fired.

### Types of Triggers in SQL Server:

There are four types of triggers available in SQL Server. They are as follows:

1. DML Triggers – Data Manipulation Language Triggers.

2. [DDL Triggers – Data Definition Language Triggers](#)
3. CLR triggers – Common Language Runtime Triggers
4. [Logon triggers](#)

In this article, we are going to discuss the **DML triggers** and the rest are going to discuss in our upcoming articles.

### What are DML Triggers in SQL Server?

As we know DML Stands for Data Manipulation Language and it provides Insert, Update and Delete statements to perform the respective operation on the database tables or view which will modify the data. The triggers which are executed automatically in response to DML events (such as Insert, Update, and Delete statements) are called DML Triggers.

### The syntax for creating a DML Triggers in SQL Server:

```
CREATE/ALTER TRIGGER TriggerName
ON TableName/ViewName
[WITH TriggerAttributes]
FOR / AFTER/InsteadOf [INSERT, UPDATE, DELETE]
AS
BEGIN
    Trigger Body
END
```

### Let us understand the syntax in detail.

1. **ON TableName or ViewName:** It refers to the table or view name on which we are defining the trigger.
2. **For/After/InsteadOf:** The **For/After** option specifies that the trigger fires only after the SQL statements are executed whereas the **InsteadOf** option specifies that the trigger is executed on behalf of the triggering SQL statement. You cannot create After Trigger on views.
3. **INSERT, UPDATE, DELETE:** The INSERT, UPDATE, DELETE specify which SQL statement will fire this trigger and we need to use at least one option or the combination of multiple options is also accepted.

**Note:** The **Insert, Update and Delete** statements are also known as Triggering SQL statements as these statements are responsible for the trigger to fire.

### Examples to understand DML Trigger in SQL Server:

In this article, we are going to discuss some simple examples to understand the triggers and from our next article, we will see the real-time usages of Triggers. Here, we are going to use the following Employee table to understand the Triggers.

Id	Name	Salary	Gender	DepartmentId
1	Pranaya	5000	Male	3
2	Priyanka	5400	Female	2
3	Anurag	6500	male	1
4	sambit	4700	Male	2
5	Hina	6600	Female	3

### Example: For/After Insert DML Trigger in SQL Server

So, basically what we want is, we want to create a **For/After DML Trigger** which should fire after the **INSERT DML operation** when performing on the **Employee** table. The trigger should restrict the **INSERT** operation on the **Employee** table. To do so, please execute the below query. As you can see in the below query, this trigger is created for the Employee table. We also specify that is a FOR trigger for the INSERT DML operation and as part of the trigger body, we are simply rollbacking the transaction which will roll back the insert statement.

```
CREATE TRIGGER trInsertEmployee
ON Employee
FOR INSERT
AS
BEGIN
PRINT 'YOU CANNOT PERFORM INSERT OPERATION'
ROLLBACK TRANSACTION
END
```

Let's try to insert the following record into the employee table.

```
INSERT INTO Employee VALUES (6, 'Saroj', 7600, 'Male', 1)
```

When you try to execute the above Insert statement it gives you the below error. First, the INSERT statement is executed, and then immediately this trigger fired and roll back the INSERT operation as well as print the message.

```
YOU CANNOT PERFORM INSERT OPERATION
Msg 3609, Level 16, State 1, Line 12
The transaction ended in the trigger. The batch has been aborted.
```

### Example: For/After Update DML Trigger in SQL Server

So, basically what we want is, we want to create a **For/After DML Trigger** which should fire after the **UPDATE DML operation** when performing on the **Employee** table. The trigger should restrict the **UPDATE** operation on the Employee table. To do so, please execute the below query. As you can see in the below query, this trigger is created for the Employee table. We also specify that is a FOR trigger for the UPDATE DML operation and as part of the trigger body, we are simply rollbacking the transaction which will roll back the UPDATE statement and print a message.

```
CREATE TRIGGER trUpdateEmployee
ON Employee
FOR UPDATE
AS
BEGIN
PRINT 'YOU CANNOT PERFORM UPDATE OPERATION'
ROLLBACK TRANSACTION
```

**END**

Let's try to update one record in the Employee table

**UPDATE Employee SET Salary = 90000 WHERE Id = 1**

When you try to execute the above Update statement it will give you the following error. First, the UPDATE statement is executed, and then immediately this trigger fired and roll back the UPDATE operation as well as print the message.

---

**YOU CANNOT PERFORM UPDATE OPERATION**

**Msg 3609, Level 16, State 1, Line 13**

**The transaction ended in the trigger. The batch has been aborted.**

### **Example3: For/After Delete DML Triggers in SQL Server**

**CREATE TRIGGER** trDeleteEmployee

**ON** Employee

**FOR DELETE**

**AS**

**BEGIN**

**PRINT** 'YOU CANNOT PERFORM DELETE OPERATION'

**ROLLBACK TRANSACTION**

**END**

**DELETE FROM Employee WHERE Id = 1**

When we try to execute the above Delete statement, it gives us the below error. First, the DELETE statement is executed, and then immediately this trigger fired and roll back the DELETE operation as well as print the message.

---

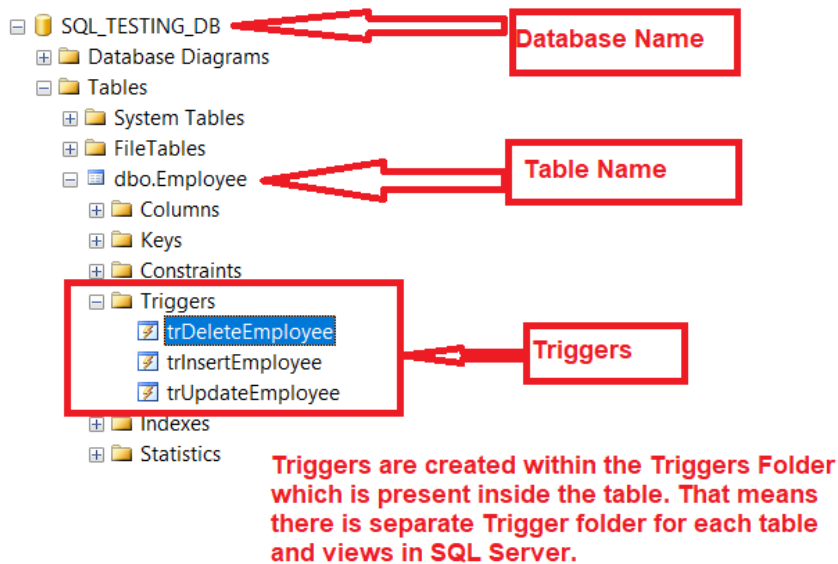
**YOU CANNOT PERFORM DELETE OPERATION**

**Msg 3609, Level 16, State 1, Line 13**

**The transaction ended in the trigger. The batch has been aborted.**

### **Where the Triggers are Created in SQL Server?**

In SQL Server, the Triggers are created within the Trigger folder which you can find when you expand the table as shown in the below image.



#### Example4: For Insert/Update/Delete DML Trigger in SQL Server

So, basically what we want is, we want to create a For/After DML Trigger which should fire after any DML operation (INSERT, UPDATE, and DELETE) when performing on the Employee table. The trigger should restrict all the DML operations on the Employee table. As you can see in the below query, this trigger is created for the Employee table. We also specify that is a FOR trigger for the INSERT, UPDATE, DELETE DML operation and as part of the trigger body, we are simply rolling back the transaction which will roll back all DML operation and print a message.

First, delete all the triggers that we already created on the Employee table. To delete you can use the below syntax.

**DROP Trigger TrggerName**

**Example:**

**DROP TRIGGER trDeleteEmployee**

**DROP TRIGGER trInsertEmployee**

**DROP TRIGGER trUpdateEmployee**

Now execute the below query to create a trigger that will restrict all the DML Operations on the Employee table.

```
CREATE TRIGGER trAllDMLOperationsOnEmployee
ON Employee
FOR INSERT, UPDATE, DELETE
AS
BEGIN
PRINT 'YOU CANNOT PERFORM DML OPERATION'
ROLLBACK TRANSACTION
END
```

Now, you cannot perform any DML operations on the Employee table because those operations are restricted by a trigger called trAllDMLOperationsOnEmployee.

**Example5:**

**Create a Trigger that will restrict all the DML operations on the Employee table on MONDAY only.**

1. SUN DAY = 1
2. MON DAY = 2
3. TUE DAY = 3
4. WED DAY = 4
5. THU DAY = 5
6. FRI DAY = 6
7. SAT DAY = 7

```
ALTER TRIGGER trAllDMLOperationsOnEmployee
ON Employee
FOR INSERT, UPDATE, DELETE
AS
BEGIN
IF DATEPART(DW,GETDATE())= 2
BEGIN
PRINT 'DML OPERATIONS ARE RESTRICTED ON MONDAY'
ROLLBACK TRANSACTION
END
END
```

**Example6:**

**Create a Trigger that will restrict all the DML operations on the Employee table before 1 pm.**

```
ALTER TRIGGER trAllDMLOperationsOnEmployee
ON Employee
FOR INSERT, UPDATE, DELETE
AS
BEGIN
IF DATEPART(HH,GETDATE()) < 13
BEGIN
```

```
PRINT 'INVALID TIME'
```

```
ROLLBACK TRANSACTION
```

```
END
```

```
END
```

### Why do we need DML Triggers in SQL Server?

DML Triggers are used to enforce business rules and data integrity. These triggers are very much similar to constraints in the way they enforce integrity. So, with the help of DML Triggers, we can enforce data integrity which cannot be done with the help of constraints that is comparing values with values of another table, etc.

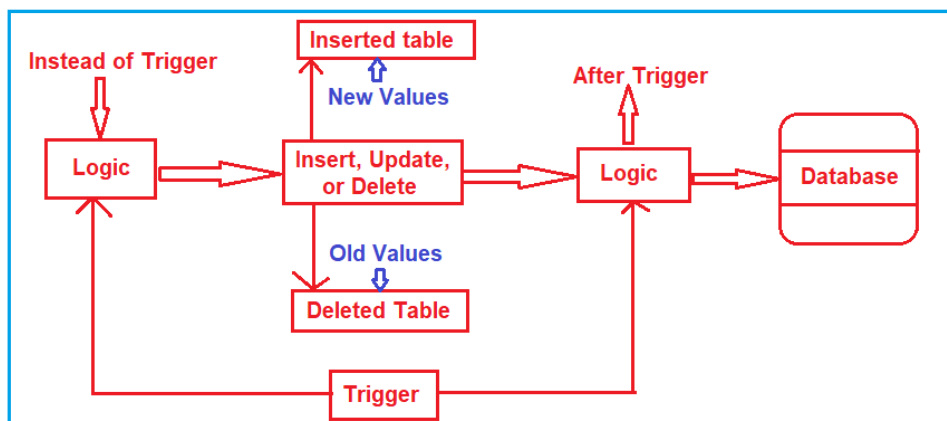
## Topic 2 :Inserted and Deleted Tables in SQL Server

1. What are Inserted and Deleted Tables in SQL Server?
2. Understanding the Inserted and Deleted Tables with Examples
3. What is Deleted Table in SQL Server?
4. How to view the updating data in a table?
5. What will happen if we update multiple records at a time?

### What are Inserted and Deleted Tables in SQL Server?

Inserted and Deleted tables are temporary tables that are created by SQL Server in the context of a trigger. That means these two tables can only be available as part of a trigger. If you try to access these tables outside of a trigger, then you will get an error. The table structure of both inserted and deleted tables will be exactly the same as the table structure of the table on which the trigger is created.

Whenever you fire any INSERT, UPDATE, and DELETE statement on a table, all the new records are actually going to the inserted table i.e. all the updated and new records are present in the inserted table. On the other hand, all the old values are present in the deleted table.



### Understanding the Inserted and Deleted Tables within a trigger with Examples in SQL Server.

Employee table to understand the Inserted and Deleted Tables in the SQL server.

Id	Name	Salary	Gender	DepartmentId
1	Pranaya	5000	Male	3
2	Priyanka	5400	Female	2
3	Anurag	6500	male	1
4	sambit	4700	Male	2
5	Hina	6600	Female	3

### Understanding the INSERTED Table in SQL Server:

The Inserted table is created by SQL Server when we perform an **INSERT** operation and this table has access to the values being inserted into the table. So, whenever we insert the values into a table those values we can see in the inserted magic table. Let us see an example for a better understanding. The following is an example Insert trigger. So, when we insert a record into the Employee table, this trigger is going to be fired after the record inserted. And the inserted data is also stored in the INSERTED magic table and those data stored in the INSERTED magic table are displaying as part of the trigger body.

```
CREATE TRIGGER trInsertEmployee
```

```
ON Employee
```

```
FOR INSERT
```

```
AS
```

```
BEGIN
```

```
SELECT * FROM INSERTED
```

```
END
```

```
INSERT INTO Employee VALUES (6, 'Saroj', 7700, 'Male', 2)
```

So when we execute the above insert statement, the data is inserted as expected in the Employee table along with a copy of the inserted new data also available in the Inserted table. So, we get the following output. Please note, the structure of the Inserted table is exactly the same as the structure of the Employee table.

Id	Name	Salary	Gen	DepartmentId
6	Saroj	7700	Male	2

When we add a new row into the Employee table a copy of the row will also be made into the INSERTED table which only a trigger can access. We cannot access this table outside the context of the trigger.

### What is Deleted Table in SQL Server?

The Deleted table is created by SQL Server when we perform a delete operation on the table and this table has access to the record being deleted. So, in simple words, we can say that, whenever we delete a record from a table the deleted record information we can view with the help of the deleted



table as part of a trigger in SQL Server. Let us understand this with an example. The following is an example delete trigger. So, when we delete a record from the Employee table, this trigger is going to be fired.

```
CREATE TRIGGER trDeleteEmployee  
ON Employee  
FOR DELETE  
AS  
BEGIN  
SELECT * FROM DELETED  
END
```

**DELETE FROM Employee WHERE Id = 6**

When we execute the above Delete statement, the data gets deleted from the Employee table whose Id is 6 along with it also displays the following deleted data as part of the deleted table.

Id	Name	Salary	Gen	DepartmentId
6	Saroj	7700	Male	2

This proves that the delete table holds the data that is being deleted from the table. When we delete a row from the Employee table, a copy of the deleted row will be made available in the DELETED table, which only a trigger can access. Just like the INSERTED table, the DELETED table cannot be accessed outside the context of a trigger and the structure of the DELETED table will be identical to the structure of the Employee table.

### How to view the updating data in a table?

When we perform an update operation, then we will be having both the inserted and deleted tables. The inserted table will hold the new values being inserted and the deleted table will hold the old values of the table. So in a simple word, we can say that, whenever we update a record in the table, then we can view the new data in the inserted table and the old data in the deleted table. Let us understand this with an example. The following is an example update trigger. So, when we update a record from the Employee table, this trigger is going to be fired.

```
CREATE TRIGGER trUpdateEmployee  
ON Employee  
FOR UPDATE  
AS  
BEGIN  
SELECT * FROM DELETED  
SELECT * FROM INSERTED  
END
```

**UPDATE Employee SET Name = 'Sharma', Salary = 8000 WHERE Id = 5**

So when we execute the above update statement, the data is updated in the table as expected along with it, it also gives us the following output. So this proves that whenever we perform an update operation then the Deleted Table will hold the OLD Data whereas the InsertedTable will hold the new Data.

Id	Name	Sal	Gender	Department
5	Hina	6600	Female	3

**OLD Data Deleted Magic Table**

Id	Name	Sal	Gender	Department
5	Sharma	8000	Female	3

**New Data Inserted Magic Table**

**What will happen if we update multiple records at a time?**

Id	Name	Salary	Gender	DepartmentId
1	Pranaya	5000	Male	3
2	Priyanka	5400	Female	2
3	Anurag	6500	male	1
4	sambit	4700	Male	2
5	Sharma	8000	Female	3

Here you can see that the Employee table having three employees whose Gender is Male. Let's update the Salary of All the Male Employees to 20000 by executing the following SQL query.

**UPDATE Employee SET Salary = 20000 WHERE Gender = 'Male'**

So when we execute the above code the data is updated in the Employee table as expected as well as we will also get the following output. As you can see in the below image, all the old records are stored in the Deleted Table whereas all the updated data are stored in the Inserted table.

Id	Name	Salary	Gen	DepartmentId
4	sambit	4700	Male	2
3	Anurag	6500	male	1
1	Pranaya	5000	Male	3

Id	Name	Salary	Gen	DepartmentId
4	sambit	20000	Male	2
3	Anurag	20000	male	1
1	Pranaya	20000	Male	3

---

## **Topic3: DML Trigger Real-Time Examples in SQL Server**

Here, we are going to implement the audit example using the DML Trigger in SQL Server. So, whenever we INSERT, UPDATE, or DELETE any data from a table, then we need to make an entry into the audit table as well. We are going to use the following Employee table to understand this concept.

Id	Name	Salary	Gender	DepartmentId
1	Pranaya	5000	Male	3
2	Priyanka	5400	Female	2
3	Anurag	6500	male	1
4	sambit	4700	Male	2
5	Hina	6600	Female	3

Along with the above Employee table, we are also going to use the following **EmployeeAudit** table.

```
CREATE TABLE EmployeeAudit
```

```
(
```

```
ID INT IDENTITY(1,1) PRIMARY KEY,
```

```
AuditData VARCHAR(MAX),
```

```
AuditDate DATETIME
```

```
)
```

Our business requirement is that whenever a new employee is added to the Employee table we want to capture the ID and name of the Employee and store the data into the **EmployeeAudit** table. The simplest way to achieve this is by using an **AFTER TRIGGER** for the **INSERT** event.

#### **Example: AFTER TRIGGER for INSERT Event in SQL Server**

The following is an example of **AFTER TRIGGER for INSERT** event on Employee table to store the inserted employee data on the **EmployeeAudit** table:

```
CREATE TRIGGER tr_Employee_For_Insert
```

```
ON Employee
```

```
FOR INSERT
```

```
AS
```

```
BEGIN
```

```
-- Declare a variable to hold the ID Value
```

```
DECLARE @ID INT
```

```
-- Declare a variable to hold the Name value
```

```
DECLARE @Name VARCHAR(100)
```

```
-- Declare a variable to hold the Audit data
```

```
DECLARE @AuditData VARCHAR(100)
```

```
-- Get the ID and Name from the INSERTED Magic table
```

```
SELECT @ID = ID, @Name = Name FROM INSERTED
```

```
-- Set the AuditData to be stored in the EmployeeAudit table
```

```

SET @AuditData = 'New employee Added with ID = ' + Cast(@ID AS VARCHAR(10)) + ' and Name ' +
@Name

-- Insert the data into the EmployeeAudit table

INSERT INTO EmployeeAudit (AuditData, AuditDate) VALUES(@AuditData, GETDATE())

END

```

Once you created the above DML AFTER INSERT Trigger. Now let us insert a record into the Employee table by executing the below INSERT statement.

**INSERT INTO Employee VALUES (6, 'Saroj', 3300, 'Male', 2)**

When we execute the above INSERT statement. Immediately after inserting the record into the Employee table, the insert trigger gets fired automatically, and a record is also inserted into the EmployeeAudit table. To verify this, issue a select query against the EmployeeAudit table as shown below

**SELECT \* FROM EmployeeAudit**

Once you execute the above query, it will give you the following output.

ID	AuditData	AuditDate
1	New employee Added with ID = 6 and Name Saroj	2018-09-22 17:30:05.640

### Example: AFTER TRIGGER for DELETE Event in SQL Server

Next, we are going to capture the audit information when an employee is deleted from the Employee table. The following is an example for **AFTER TRIGGER for DELETE** event on the Employee table in SQL Server to store the deleted employee data on the **EmployeeAudit** table.

```

CREATE TRIGGER tr_Employee_For_Delete

ON Employee

FOR DELETE

AS

BEGIN

-- Declare a variable to hold the ID Value

DECLARE @ID INT

-- Declare a variable to hold the Name value

DECLARE @Name VARCHAR(100)

-- Declare a variable to hold the Audit data

DECLARE @AuditData VARCHAR(100)

-- Get the ID and Name from the DELETED table

SELECT @ID = ID, @Name = Name FROM DELETED

-- Set the AuditData to be stored in the EmployeeAudit table

```

```

SET @AuditData = 'An employee is deleted with ID = ' + Cast(@ID AS VARCHAR(10)) + ' and Name = ' +
@Name

-- Insert the data into the EmployeeAudit table

INSERT INTO EmployeeAudit (AuditData, AuditDate)VALUES(@AuditData, GETDATE())

END

```

Once you created the above DML AFTER DELETE Trigger. Now let us DELETE a record from the Employee table by executing the below DELETE SQL statement.

**DELETE FROM Employee WHERE ID = 6**

So when we execute the above DELETE statement. Immediately after Deleting the row from the Employee table, the delete trigger gets fired automatically, and a row into EmployeeAudit is also inserted. To verify this, issue a select query against the EmployeeAudit table as shown below

**SELECT \* FROM EmployeeAudit**

Once you execute the above query, it will give you the following output.

ID	AuditData	AuditDate
1	New employee Added with ID = 6 and Name Saroj	2018-09-22 17:30:05.640
2	An employee is deleted with ID = 6 and Name = S...	2018-09-22 17:43:02.090

As we already see, triggers make use of 2 special tables INSERTED and DELETED. The inserted table contains the updated data or new data whereas the deleted table contains the old data or deleted data. The After trigger for the UPDATE event makes use of both the inserted and deleted magic tables.

#### Example: AFTER TRIGGER for UPDATE Event in SQL Server

Now we are going to capture the audit information when an employee is updated from the Employee table. The following is an example of **AFTER TRIGGER for UPDATE** event on Employee table to store the updated employee data on the **EmployeeAudit** table.

```

CREATE TRIGGER tr_Employee_For_Update
ON Employee
FOR Update
AS
BEGIN
-- Declare the variables to hold old and updated data

DECLARE @ID INT

DECLARE @Old_Name VARCHAR(200), @New_Name VARCHAR(200)

DECLARE @Old_Salary INT, @New_Salary INT

DECLARE @Old_Gender VARCHAR(200), @New_Gender VARCHAR(200)

DECLARE @Old_DepartmentId INT, @New_DepartmentId INT

-- Declare Variable to build the audit string

```

```

DECLARE @AuditData VARCHAR(MAX)

-- Store the updated data into a temporary table

SELECT *

INTO #UpdatedDataTempTable

FROM INSERTED

-- Loop thru the records in the UpdatedDataTempTable temp table

WHILE(Exists(SELECT ID FROM #UpdatedDataTempTable))

BEGIN

--Initialize the audit string to empty string

SET @AuditData = ''

-- Select first row data from temp table

SELECT TOP 1 @ID = ID,

@New_Name = Name,

@New_Gender = Gender,

@New_Salary = Salary,

@New_DepartmentId = DepartmentId

FROM #UpdatedDataTempTable

-- Select the corresponding row from deleted table

SELECT @Old_Name = Name,

@Old_Gender = Gender,

@Old_Salary = Salary,

@Old_DepartmentId = DepartmentId

FROM DELETED WHERE ID = @ID

-- Build the audit data dynamically

Set @AuditData = 'Employee with Id = ' + CAST(@ID AS VARCHAR(6)) + ' changed'

-- If old name and new name are not same, then its changed

IF(@Old_Name <> @New_Name)

BEGIN

Set @AuditData = @AuditData + ' Name from ' + @Old_Name + ' to ' + @New_Name

END

-- If old Gender and new gender are not same, then its changed

```

```

IF(@Old_Gender <> @New_Gender)

BEGIN

Set @AuditData = @AuditData + ' Gender from ' + @Old_Gender + ' to ' + @New_Gender

END

-- If old Salary and new Salary are not same, then its changed

IF(@Old_Salary <> @New_Salary)

BEGIN

Set @AuditData = @AuditData + ' Salary from ' + Cast(@Old_Salary AS VARCHAR(10))+ ' to '

+ Cast(@New_Salary AS VARCHAR(10))

END

-- If old Department ID and new Department ID are not same, then its changed

IF(@Old_DepartmentId <> @New_DepartmentId)

BEGIN

Set @AuditData = @AuditData + ' DepartmentId from ' + Cast(@Old_DepartmentId AS

VARCHAR(10))+ ' to '

+ Cast(@New_DepartmentId AS VARCHAR(10))

END

-- Then Insert the audit data into the EmployeeAudit table

INSERT INTO EmployeeAudit(AuditData, AuditDate) VALUES(@AuditData, GETDATE())

-- Delete the current row from temp table, so we can move to the next row

DELETE FROM #UpdatedDataTempTable WHERE ID = @ID

End

End

```

Now perform the update operation on the Employee table and you will see everything is working as expected.

## Topic4: Instead Of Trigger in SQL Server with Examples

1. What is Instead Of Triggers in SQL Server?
2. Example to understand Instead of Triggers in SQL Server
3. **INSTEAD OF INSERT** trigger in SQL Server
4. Instead Of Update Trigger in SQL Server
5. Instead Of Delete Trigger in SQL Server

## What is Instead Of Triggers in SQL Server?

The INSTEAD OF triggers are the DML triggers that are fired instead of the triggering event such as the INSERT, UPDATE or DELETE events. So, when you fire any DML statements such as Insert, Update, and Delete, then on behalf of the DML statement, the instead of trigger is going to execute. In real-time applications, Instead Of Triggers are used to correctly update a complex view.

### Example to understand Instead of Triggers in SQL Server:

We are going to use the following Department and Employee table to understand the complex views in SQL Server.

#### Department

ID	Name
1	IT
2	HR
3	Sales

#### Employee

ID	Name	Gender	DOB	Salary	DeptID
1	Pranaya	Male	1996-02-29 10:53:27.060	25000.00	1
2	Priyanka	Female	1995-05-25 10:53:27.060	30000.00	2
3	Anurag	Male	1995-04-19 10:53:27.060	40000.00	2
4	Preety	Female	1996-03-17 10:53:27.060	35000.00	3
5	Sambit	Male	1997-01-15 10:53:27.060	27000.00	1
6	Hina	Female	1995-07-12 10:53:27.060	33000.00	2

We want to retrieve the following data from the Employee and Department table.

ID	Name	Gender	Salary	Department
1	Pranaya	Male	25000.00	IT
2	Priyanka	Female	30000.00	HR
3	Anurag	Male	40000.00	HR
4	Preety	Female	35000.00	Sales
5	Sambit	Male	27000.00	IT
6	Hina	Female	33000.00	HR

So, let's create a view that will return the above results.

```
CREATE VIEW vwEmployeeDetails
```

```
AS
```

```
SELECT emp.ID, emp.Name, Gender, Salary, dept.Name AS Department
```

```
FROM Employee emp
```

```
INNER JOIN Department dept
```

```
ON emp.DeptID = dept.ID
```

Now let's try to insert a record into the view vwEmployeeDetails by executing the following query.

```
INSERT INTO vwEmployeeDetails VALUES(7, 'Saroj', 'Male', 50000, 'IT')
```

When we execute the above query it gives us the error as '**View or function vwEmployeeDetails is not updatable because the modification affects multiple base tables.**'



## How to overcome the above problem?

By using Instead of Insert Trigger.

### INSTEAD OF INSERT Trigger in SQL Server:

Here you can see that inserting a record into a view that is based on multiple tables gives us an error. Now let's understand how the INSTEAD OF TRIGGERS can help us in situations like this. As we are getting an error when we are trying to insert a record into the view, let's create an INSTEAD OF INSERT trigger on the view vwEmployeeDetails to correctly insert the records into the appropriate table.

```
CREATE TRIGGER tr_vwEmployeeDetails_InsteadOfInsert
ON vwEmployeeDetails
INSTEAD OF INSERT
AS
BEGIN
DECLARE @DepartmentId int
-- First Check if there is a valid DepartmentId in the Department Table
-- for the given Department Name
SELECT @DepartmentId = dept.ID
FROM Department dept
INNER JOIN INSERTED inst
on inst.Department = dept.Name
--If the DepartmentId is null then throw an error
IF(@DepartmentId is null)
BEGIN
RAISERROR('Invalid Department Name. Statement terminated', 16, 1)
RETURN
END
-- Finally insert the data into the Employee table
INSERT INTO Employee(ID, Name, Gender, Salary, DeptID)
SELECT ID, Name, Gender, Salary, @DepartmentId
FROM INSERTED
End
```

Now, let's execute the below Insert statement.

```
INSERT INTO vwEmployeeDetails VALUES(7, 'Saroj', 'Male', 50000, 'IT')
```

Instead Of Trigger inserts the row **correctly** into the Employee table as expected. Since we are inserting a row, the **inserted magic** table will contain the newly added row whereas the deleted table will be empty. Now check the data by issuing a select query against the Employee Table or the vwEmployeeDetails view.

**SELECT \* FROM vwEmployeeDetails** will give us the below result.

ID	Name	Gender	Salary	Department
1	Pranaya	Male	25000.00	IT
2	Priyanka	Female	30000.00	HR
3	Anurag	Male	40000.00	HR
4	Preety	Female	35000.00	Sales
5	Sambit	Male	27000.00	IT
6	Hina	Female	33000.00	HR
7	Saroj	Male	50000.00	IT

As you can see from the above image, the record is inserted as expected into the Employee table.

#### Instead Of Update Trigger in SQL Server:

The INSTEAD OF UPDATE Trigger in SQL server gets fired instead of the UPDATE event on a table or a view. For example, let's say we have an INSTEAD OF UPDATE trigger on a view or on a table, and when we try to update a record (or records) from that view or table, instead of the actual UPDATE event, the trigger gets fired automatically. The **Instead Of Update Trigger in SQL Server** is usually used to correctly update the records from a view that is based on multiple tables.

Please update the Department and Employee table as shown below to understand this concept.

#### Department

ID	Name
1	IT
2	HR
3	Sales

#### Employee

ID	Name	Gender	DOB	Salary	DeptID
1	Pranaya	Male	1996-02-29 10:53:27.060	25000.00	1
2	Priyanka	Female	1995-05-25 10:53:27.060	30000.00	2
3	Anurag	Male	1995-04-19 10:53:27.060	40000.00	2
4	Preety	Female	1996-03-17 10:53:27.060	35000.00	3
5	Sambit	Male	1997-01-15 10:53:27.060	27000.00	1
6	Hina	Female	1995-07-12 10:53:27.060	33000.00	2

We want to retrieve the following data from the Employee and Department table.

ID	Name	Gender	Salary	Department
1	Pranaya	Male	25000.00	IT
2	Priyanka	Female	30000.00	HR
3	Anurag	Male	40000.00	HR
4	Preety	Female	35000.00	Sales
5	Sambit	Male	27000.00	IT
6	Hina	Female	33000.00	HR

**So, let's create a view that will return the above results.**

```
CREATE VIEW vwEmployeeDetails  
AS  
SELECT emp.ID, emp.Name, Gender, Salary, dept.Name AS Department  
FROM Employee emp  
INNER JOIN Department dept  
ON emp.DeptID = dept.ID
```

Now let's try to update the view vwEmployeeDetails in such a way that it affects both the underlying tables such as Employee and Department table and see if we get any error.

The following UPDATE statement changes the Name and Salary column from the Employee table and the Department Name column from the Department table.

```
UPDATE vwEmployeeDetails  
  
SET Name = 'Kumar',  
Salary = 45000,  
Department = 'HR'  
  
WHERE Id = 1
```

When we execute the above update query, we get the error as **“View or function ‘vwEmployeeDetails’ is not updatable because the modification affects multiple base tables.”**

**Note:** When the view is based on multiple tables and if the update statement affects more than one table then the update failed.

Now let's try to change only the department of Pranaya from IT to HR. The following UPDATE query affects only one base table that is the Department table. So, the update query should succeed. But before executing the query please note that employee Sambit is also in the IT department.

```
UPDATE vwEmployeeDetails  
  
SET Department = 'HR'  
  
WHERE Id = 1
```

Once we execute the above query, then select the data from the view and notice that **Sambit's Department** is also changed to **HR**.

```
SELECT * FROM vwEmployeeDetails
```

ID	Name	Gender	Salary	Department
1	Pranaya	Male	25000.00	HR
2	Priyanka	Female	30000.00	HR
3	Anurag	Male	40000.00	HR
4	Preety	Female	35000.00	Sales
5	Sambit	Male	27000.00	HR
6	Hina	Female	33000.00	HR

We intended to just change Pranaya's Department Name but it also changes the Department Name of Sambit. So the UPDATE didn't work as expected. This is because the UPDATE query updated the Department Name from IT to HR in the Department table.

**SELECT \* FROM Department**

ID	Name
1	HR
2	HR
3	Sales

As you can see the Record with Id = 1, has the Department Name changed from IT to HR. So, the conclusion is that, if a view is based on multiple tables, and if we want to update the view, the UPDATE may not always work as expected. To correctly update the underlying base tables, through a view, the Instead Of Update Trigger in SQL Server can be used.

Before we create the trigger, let's update the Department Name to IT for record with Id = 1.

**UPDATE Department SET Name = 'IT' WHERE ID = 1**

**Please use the below SQL Script to create the Instead Of Update Trigger in SQL Server**

```

CREATE TRIGGER tr_vwEmployeeDetails_InsteadOfUpdate
ON vwEmployeeDetails
INSTEAD OF UPDATE
AS
BEGIN
-- if EmployeeId is updated
IF(UPDATE(ID))
BEGIN
RAISERROR('Id cannot be changed', 16, 1)
RETURN
END
-- If Department Name is updated
IF(UPDATE(Department))
BEGIN

```

```
DECLARE @DepartmentID INT

SELECT @DepartmentID = dept.ID

FROM Department dept

INNER JOIN INSERTED inst

ON dept.Name = inst.Department

IF(@DepartmentID is NULL )

BEGIN

RAISERROR('Invalid Department Name', 16, 1)

RETURN

END

UPDATE Employee set DeptID = @DepartmentID

FROM INSERTED

INNER JOIN Employee

on Employee.ID = inserted.ID

End

-- If gender is updated

IF(UPDATE(Gender))

BEGIN

UPDATE Employee SET Gender = inserted.Gender

FROM INSERTED

INNER JOIN Employee

ON Employee.ID = INSERTED.ID

END

-- If Salary is updated

IF(UPDATE(Salary))

BEGIN

UPDATE Employee SET Salary = inserted.Salary

FROM INSERTED

INNER JOIN Employee

ON Employee.ID = INSERTED.ID

END

-- If Name is updated

IF(UPDATE(Name))

BEGIN

UPDATE Employee SET Name = inserted.Name
```

```
FROM INSERTED
INNER JOIN Employee
ON Employee.ID = INSERTED.ID
END
END
```

**Note:** The Update() function used in the trigger returns true, even if we update with the same value. For this reason, I recommended comparing values between inserted and deleted tables, rather than relying on the Update() function. The Update() function does not operate on a per-row basis but across all rows.

**Now, let's try to update Pranaya's Department to HR.**

**UPDATE vwEmployeeDetails SET Department = 'HR' WHERE Id = 1**

The UPDATE query works as expected. The INSTEAD OF UPDATE trigger, correctly updates, Pranaya's DeptId to 2 in Employee table.

Now, let's try to update Name, Gender, Salary and Department Name. The UPDATE query, works as expected, without raising the error – **'View or function vwEmployeeDetails is not updatable because the modification affects multiple base tables.'**

```
UPDATE vwEmployeeDetails
SET Name = 'Preety',
    Gender = 'Female',
    Salary = 44000,
    Department = 'IT'
WHERE Id = 1
```

### **Instead Of Delete Trigger in SQL Server:**

The INSTEAD OF DELETE Trigger in SQL server gets fired instead of the DELETE event on a table or a view. For example, let's say we have an INSTEAD OF DELETE trigger on a view or on a table, and when we try to delete a row from that view or table, instead of the actual DELETE event, the trigger gets fired automatically. INSTEAD OF DELETE TRIGGERS are usually used to delete the records from a view that is based on multiple tables.

Please update the Department and Employee table as shown below to understand this concept.

## Department Employee

ID	Name	ID	Name	Gender	DOB	Salary	DeptID
1	IT	1	Pranaya	Male	1996-02-29 10:53:27.060	25000.00	1
2	HR	2	Priyanka	Female	1995-05-25 10:53:27.060	30000.00	2
3	Sales	3	Anurag	Male	1995-04-19 10:53:27.060	40000.00	2
		4	Preety	Female	1996-03-17 10:53:27.060	35000.00	3
		5	Sambit	Male	1997-01-15 10:53:27.060	27000.00	1
		6	Hina	Female	1995-07-12 10:53:27.060	33000.00	2

We want to retrieve the following data from the Employee and Department table.

ID	Name	Gender	Salary	Department
1	Pranaya	Male	25000.00	IT
2	Priyanka	Female	30000.00	HR
3	Anurag	Male	40000.00	HR
4	Preety	Female	35000.00	Sales
5	Sambit	Male	27000.00	IT
6	Hina	Female	33000.00	HR

So, let's create a view that will return the above results.

```
CREATE VIEW vwEmployeeDetails
AS
SELECT emp.ID, emp.Name, Gender, Salary, dept.Name AS Department
FROM Employee emp
INNER JOIN Department dept
ON emp.DeptID = dept.ID
```

Now let's try to delete a record from the view vwEmployeeDetails by executing the following query.

```
DELETE FROM vwEmployeeDetails WHERE ID = 1
```

When we execute the above query it gives us the error as **'View or function vwEmployeeDetails is not updatable because the modification affects multiple base tables.'**

Here we can see that deleting a record from a view that is based on multiple tables gives us an error. Now let's understand how the INSTEAD OF TRIGGERS can help us in a situation like this. As we are getting an error when we are trying to Delete a record from the view, let's create an INSTEAD OF DELETE trigger on the view vwEmployeeDetails to correctly delete the data.

```
CREATE TRIGGER tr_vwEmployeeDetails_InsteadOfDelete
ON vwEmployeeDetails
INSTEAD OF DELETE
AS
```

**BEGIN**

-- Using Inner Join

**DELETE FROM** Employee

**FROM** Employee emp

**INNER JOIN DELETED** del

**ON** emp.ID = del.ID

-- Using the Subquery

-- DELETE FROM Employee

-- WHERE ID IN (SELECT ID FROM DELETED)

**END**

Now, let's execute the below delete statement.

**DELETE FROM vwEmployeeDetails WHERE ID = 1**

The Instead Of Trigger deletes the row correctly from the Employee table as expected. Since we are deleting a row, the deleted magic table will contain all the rows that we want to delete whereas the inserted table will be empty. Now check the data by issuing a select query against the Employee Table or the vwEmployeeDetails view.

**SELECT \* FROM vwEmployeeDetails** will give us the below result.

ID	Name	Gender	Salary	Department
2	Priyanka	Female	30000.00	HR
3	Anurag	Male	40000.00	HR
4	Preety	Female	35000.00	Sales
5	Sambit	Male	27000.00	IT
6	Hina	Female	33000.00	HR

As you can see from the above image, the record is deleted as expected from the Employee table.

---

#### Topic5: DDL Triggers in SQL Server with Examples

1. **What are DDL TRIGGERS in SQL Server?**
2. **Types of DDL triggers?**
3. **Database Scoped DDL Triggers in SQL Server**
4. **How to Create a Database-Scoped DDL Trigger?**
5. **Where can I find the Database-scoped DDL triggers?**
6. **How to drop, enable and disable a Database Scoped DDL trigger?**
7. **Server-scoped DDL Triggers in SQL Server**



8. **How to Create a Server-Scoped DDL Trigger?**
9. **Where can I find the Server-scoped DDL triggers?**
10. **How to drop, enable and disable a Server Scoped DDL trigger?**
11. **A real-time example of DDL Trigger**

## **What are DDL TRIGGERS in SQL Server?**

The DDL triggers in SQL Server are fired in response to a variety of data definition language (DDL) events such as Create, Alter, Drop, Grant, Denay, and Revoke (Table, Function, Index, Stored Procedure, etc...). That means DDL triggers in SQL Server are working on a database.

DDL triggers are introduced from SQL Server 2005 version which will be used to restrict the DDL operations such as CREATE, ALTER and DROP commands.

We can think of a DDL trigger as a special kind of stored procedure that executes in response to a server scoped or database scoped events. We will discuss the examples of both server scoped and database scoped.

The point to remember is that the DDL triggers fire only after the DDL statements execute so we cannot use the “Instead Of Triggers” here and moreover the DDL triggers will not fire in response to events that affect the local temporary tables.

### **Syntax:**

```
CREATE TRIGGER <TRIGGER NAME>
ON ALL SERVER/ DATABASE
[with trigger attributes]
FOR / ALTER <Event_Type>
AS
BEGIN
    <TRIGGER BODY/ STATEMENTS>
END
```

Here **Event\_Type** refers to the event that will fire the trigger which can be anything like **Create\_Table**, **Drop\_Table**, **Alter\_Table**, etc.

## **Types of DDL triggers in SQL Server?**

There are two types of DDLs triggers available in SQL Server. They are as follows:

1. **Database Scoped DDL Trigger**
2. **Server Scoped DDL Trigger**

The DDL triggers can be created in a specific database or at the server level. If we set the scope to server-level then it is applied to all the databases of that server. Here, in this article, first, we will discuss the database scoped triggers, and then, we will discuss server scoped triggers.

### **Database Scoped DDL Triggers in SQL Server**

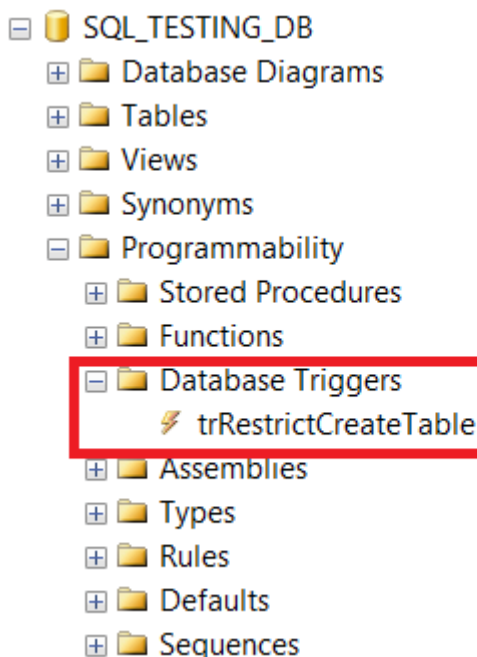
First Create a database with the name SQL\_TESTING\_DB

**Example1: Create a trigger that will restrict creating a new table on a specific database.**

```
USE SQL_TESTING_DB  
  
GO  
  
CREATE TRIGGER trRestrictCreateTable  
  
ON DATABASE  
  
FOR CREATE_TABLE  
  
AS  
  
BEGIN  
  
PRINT 'YOU CANNOT CREATE A TABLE IN THIS DATABASE'  
  
ROLLBACK TRANSACTION  
  
END
```

**Where can I find the Database-Scoped DDL triggers?**

In the Object Explorer window expand the SQL\_TESTING\_DB database by clicking on the plus symbol. Expand the **Programmability** folder. Then Expand the **Database Triggers** folder as shown below



**Note:** If you can't find the trigger that you just created in the SQL\_TESTING\_DB database, make sure to refresh the Database Triggers folder. When you execute the following code to create a table, the trigger will automatically fire and will print the message – **YOU CANNOT CREATE A TABLE IN THIS DATABASE**

```
CREATE TABLE tblTest (ID INT)
```

**Example2: Create a trigger that will restrict ALTER operations on a specific database table.**

```
CREATE TRIGGER trRestrictAlterTable
```

```
ON DATABASE
FOR ALTER_TABLE
AS
BEGIN
PRINT 'YOU CANNOT ALTER TABLES'
ROLLBACK TRANSACTION
END
```

**Example3: Create a trigger that will restrict dropping the tables from a specific database.**

```
CREATE TRIGGER trRestrictDropTable
ON DATABASE
FOR DROP_TABLE
AS
BEGIN
PRINT 'YOU CANNOT DROP TABLES'
ROLLBACK TRANSACTION
END
```

**Note:** We cannot implement business logic in DDL Trigger. To be able to create, alter or drop a table we either have to disable or delete the trigger.

### **How to drop a Database Scoped DDL trigger in SQL Server?**

Right-click on the trigger in object explorer and select “Delete” from the context menu. We can also drop the trigger using the following T-SQL command

```
DROP TRIGGER trRestrictCreateTable ON DATABASE
```

```
DROP TRIGGER trRestrictAlterTable ON DATABASE
```

```
DROP TRIGGER trRestrictDropTable ON DATABASE
```

Let us see an example of how to prevent users from creating, altering, or dropping tables from a specific database using a single trigger.

```
CREATE TRIGGER trRestrictDDLEvents
ON DATABASE
FOR CREATE_TABLE, ALTER_TABLE, DROP_TABLE
AS
BEGIN
```

```
PRINT 'You cannot create, alter or drop a table'
```

```
ROLLBACK TRANSACTION
```

```
END
```

### How to disable a Database Scoped DDL trigger in SQL Server?

Right-click on the trigger in object explorer and select “Disable” from the context menu. We can also disable the trigger using the following T-SQL command

```
DISABLE TRIGGER trRestrictDDLEvents ON DATABASE
```

### How to enable a Database Scoped DDL trigger in SQL Server?

Right-click on the trigger in object explorer and select “Enable” from the context menu. We can also enable the trigger using the following T-SQL command

```
ENABLE TRIGGER trRestrictDDLEvents ON DATABASE
```

Certain system stored procedures that perform DDL-like operations can also fire DDL triggers. The following trigger will be fired whenever we rename a database object using the **sp\_rename** system stored procedure.

```
CREATE TRIGGER trRenameTable
ON DATABASE
FOR RENAME
AS
BEGIN
PRINT 'You just renamed something'
END
```

Let’s create a table and test this.

First, disable the trRestrictDDLEvents trigger: **DISABLE TRIGGER trRestrictDDLEvents ON DATABASE**

Then create a table using the command: **CREATE TABLE tblTest (ID INT)**

The following code changes the name of the table tblTest to tblTestChanged. When this code is executed, it will fire the trigger trRenameTable automatically.

```
sp_rename 'tblTest', 'tblTestChanged'
```

When we execute the above code, it will display the below output.

```
Caution: Changing any part of an object name could break scripts and stored procedures.
You just renamed something
```

**Server-scoped DDL Triggers in SQL Server:**

Let's understand the need for a Server-Scoped DDL Trigger with an example. We already created the following trigger.

```
CREATE TRIGGER trRestrictDDLEvents  
ON DATABASE  
FOR CREATE_TABLE, ALTER_TABLE, DROP_TABLE  
AS  
BEGIN  
PRINT 'You cannot create, alter or drop a table'  
ROLLBACK TRANSACTION  
END
```

The above trigger is an example of a **Database Scoped DDL Trigger**. This trigger will prevent the users from creating, altering, or dropping tables only from the database on which it is created.

But, if we have another database on the same server, then the users will be able to create, alter or drop tables in that database. So, if we want to prevent the users from creating, altering, or dropping tables from that database then we need to create the trigger again in that particular database.

Think of a situation, where we have 50 databases on a particular server and we want to prevent the users from creating, altering, or dropping tables from all those 50 databases. Creating the same trigger again and again for all those 50 databases is bad for the following two reasons.

1. It is a tedious process as well as error-prone
2. Maintainability is a nightmare. If for some reason we have to change the trigger, then we will have to do it in all the 50 databases, which again is a tedious process as well as error-prone.

This is the ideal situation where the Server-Scoped DDL triggers come into the picture. When we create a server-scoped DDL trigger, then it will be fired in response to the DDL events happening in all of the databases on that particular server.

### **How to Create a Server-Scoped DDL Trigger in SQL Server?**

Creating a server scoped DDL trigger in SQL Server is very much similar to creating a database scoped DDL trigger, except that we will have to change the scope to ALL Server as shown in the below example.

```
CREATE TRIGGER trServerScopedDDLTrigger  
ON ALL SERVER  
FOR CREATE_TABLE, ALTER_TABLE, DROP_TABLE  
AS  
BEGIN  
PRINT 'You cannot create, alter or drop a table in any database of this server'
```

## ROLLBACK TRANSACTION

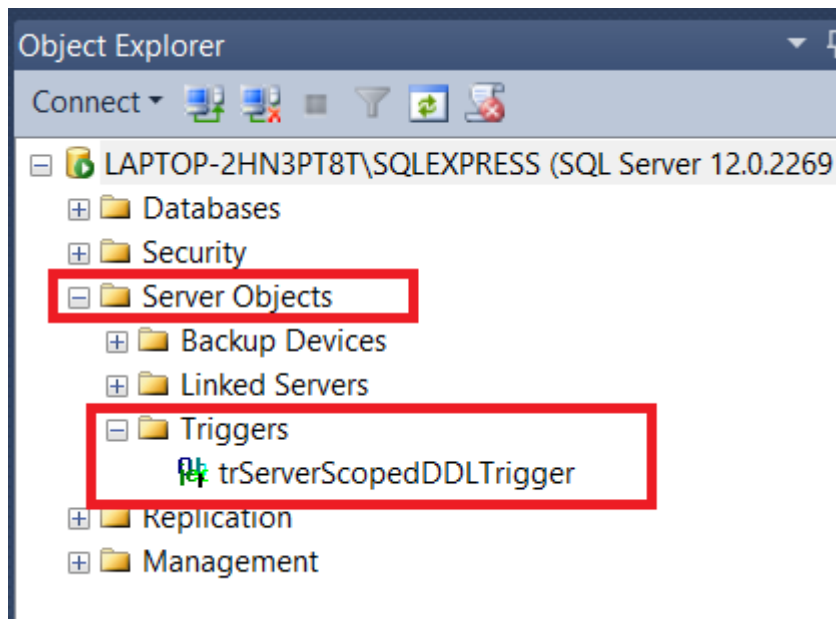
END

Now if you will try to create, alter or drop a table in any of the databases on that particular server, then the above Server-Scoped DDL trigger will be fired.

### Where can I find the Server-scoped DDL triggers?

To find the Server-Scoped DDL Triggers in SQL Server Follow the below steps

1. In the Object Explorer window, expand the “Server Objects” folder
2. Then Expand the Triggers folder as shown in the below image



### How to disable Server-Scoped DDL trigger in SQL Server?

Right-click on the trigger in object explorer and select “**Disable**” from the context menu. We can also disable the trigger using the SQL Command

as **DISABLE TRIGGER trServerScopedDDLTrigger ON ALL SERVER**

### How to enable Server-Scoped DDL trigger in SQL Server?

Right-click on the trigger in object explorer and select “**Enable**” from the context menu. We can also enable the trigger using the T-SQL command

as **ENABLE TRIGGER trServerScopedDDLTrigger ON ALL SERVER**

### How to drop Server-scoped DDL trigger in SQL Server?

Right-click on the trigger in object explorer and select “**Delete**” from the context menu. We can also drop the trigger using the SQL command

as **DROP TRIGGER trServerScopedDDLTrigger ON ALL SERVER**

### DDL Trigger Real-Time Example in SQL Server:

The DDL triggers in SQL Server will be very much handy to audit and control the DDL changes in a database. Below are such real-time scenarios:

1. To Track the DLL changes

2. Track the DDL statement which is fired
3. Who has fired the DDL statements? For example, we may be interested in identifying who has dropped the table or who has modified the table.
4. When the DDL statement is fired.
5. Block the user from doing some DDL changes like DROP TABLE, DROP PROCEDURE, etc.
6. Allow the DDL changes only during a specified window (i.e. only during particular hours of the day)

**So let us discuss how to audit table changes in SQL Server using a DDL trigger.**

-- Create the TableAudit table

```
CREATE TABLE TableAudit
(
    DatabaseName nvarchar(250),
    TableName nvarchar(250),
    EventType nvarchar(250),
    LoginName nvarchar(250),
    SQLCommand nvarchar(2500),
    AuditDateTime datetime
)
Go
```

-- The following trigger audits all table changes in all databases on a particular Server

```
CREATE TRIGGER tr_AuditTableChangesInAllDatabases
ON ALL SERVER
FOR CREATE_TABLE, ALTER_TABLE, DROP_TABLE
AS
BEGIN
DECLARE @EventData XML
SELECT @EventData = EVENTDATA()
INSERT INTO SQL_TESTING_DB.dbo.TableAudit
(DatabaseName, TableName, EventType, LoginName,
SQLCommand, AuditDateTime)
VALUES
(
    @EventData.value('/EVENT_INSTANCE/DatabaseName)[1]', 'varchar(250)'),
    @EventData.value('/EVENT_INSTANCE/ObjectName)[1]', 'varchar(250)'),
```

```

@EventData.value('/EVENT_INSTANCE/EventType)[1]', 'nvarchar(250)'),
@EventData.value('/EVENT_INSTANCE/LoginName)[1]', 'varchar(250)'),
@EventData.value('/EVENT_INSTANCE/TSQLCommand)[1]', 'nvarchar(2500)'),
GetDate()
)
END

```

In the above example, we are using the EventData() function which will returns event the data in XML format. The following XML is returned by the EventData() function when I created a table with name = MyTestTable in SQL\_TESTING\_DB database.

```
CREATE TABLE MyTestTable
```

```

(
Id INT,
Name VARCHAR(50),
Gender VARCHAR(50),
Salary INT
)

```

Once you create the above table then select the TableAudit as shown below

**SELECT \* FROM TableAudit** will give us the below output.

DatabaseName	TableName	EventType	LoginName	SQLCommand	AuditDateTime
SQL_TESTING_DB	MyTestTable	CREATE_TABLE	LAPTOP-2HN3PT8T...	CREATE TABLE MyTestTabl...	2018-10-10 22:05:37.453

The EVENTDATA() function give us the data in below XML Format

```

<EVENT_INSTANCE>
<EventType>CREATE_TABLE</EventType>
<PostTime> 2018-10-10 22:05:37.453 </PostTime>
<SPID>58</SPID>
<ServerName> LAPTOP-2HN3PT8T </ServerName>
<LoginName>LAPTOP-2HN3PT8T\Pranaya</LoginName>
<UserName>dbo</UserName>
<DatabaseName>SQL_TESTING_DB</DatabaseName>
<SchemaName>dbo</SchemaName>
<ObjectName>MyTestTable</ObjectName>
<ObjectType>TABLE</ObjectType>
<TSQLCommand>
<SetOptions ANSI_NULLS="ON" ANSI_NULL_DEFAULT="ON"
ANSI_PADDING="ON" QUOTED_IDENTIFIER="ON"

```



```
ENCRYPTED="FALSE" />
```

```
<CommandText>
```

```
CREATE TABLE MyTestTable
```

```
(
```

```
Id INT,
```

```
Name VARCHAR(50),
```

```
Gender VARCHAR(50),
```

```
Salary INT
```

```
)
```

```
</CommandText>
```

```
</TSQLCommand>
```

```
</EVENT_INSTANCE>
```

---

## Topic6: Triggers Execution Order in SQL Server with Examples

In this article, I am going to discuss the Triggers Execution Order in SQL Server with Examples. Please read our previous article where we discussed [DDL Triggers in SQL Server](#) with examples. At the end of this article, you will understand how to set the execution order of triggers using the **sp\_settriggerorder** system stored procedure.

**Note:** The Server-Scoped Triggers in SQL Server are always fired before any of the databases scoped triggers and we cannot change this execution order.

### **Example to understand Triggers Execution Order in SQL Server:**

Let's create two DDL Triggers with one database Scoped and one Server-Scoped. We need to create the Database Scoped Trigger in a specific database.

```
CREATE TRIGGER tr_DatabaseScopeDDLTrigger
```

```
ON DATABASE
```

```
FOR CREATE_TABLE
```

```
AS
```

```
BEGIN
```

```
Print 'Database Scope DDL Trigger'
```

```
END
```

```
GO
```

```
CREATE TRIGGER tr_ServerScopeDDLTrigger
```

```
ON ALL SERVER
FOR CREATE_TABLE
AS
BEGIN
Print 'Server Scope DDL Trigger'
END
GO
```

Here we have created one database-scoped (tr\_DatabaseScopeDDLTrigger) and one server-scoped (tr\_ServerScopeDDLTrigger) DDL trigger and both the triggers handling the same DDL event i.e. CREATE\_TABLE.

**Let's create a table and see what happens**

**CREATE TABLE Employee(ID INT, Name VARCHAR(100))**

**When we execute the above create statement it will give us the below output.**

```
Server Scope DDL Trigger
Database Scope DDL Trigger
```

So, from the above output, it proves that the Server-Scoped DDL Trigger is always fired before the Database-Scoped DDL Trigger is fired in SQL Server.

We can use the **sp\_settriggerorder** system stored procedure to set the execution order of Server-Scoped or Database-Scoped DDL triggers in SQL Server.

The **sp\_settriggerorder** system stored procedure has 4 parameters

1. **@triggername:** The Name of the Trigger
2. **@order:** The order value can be First, Last, or None. When we set to None, the trigger is fired in random order
3. **@stmttype:** The SQL statement that fires the trigger and the value can be INSERT, UPDATE, DELETE, or any DDL event
4. **@namespace:** The namespace determines the Scope of the trigger and the value can be DATABASE, SERVER, or NULL

**To understand the execution order, Let's create another database scoped DDL Trigger as shown below**

```
CREATE TRIGGER tr_DatabaseScopeDDLTrigger
ON DATABASE
FOR CREATE_TABLE
AS
BEGIN
Print 'Database Scope DDL Trigger'
```

```
END
```

```
GO
```

**Now set the Execution order using the sp\_settriggerorder system stored procedure as shown below.**

```
EXEC sp_settriggerorder
```

```
@triggername = 'tr_DatabaseScopeDDLTrigger1',
```

```
@order = 'FIRST',
```

```
@stmttype = 'CREATE_TABLE',
```

```
@namespace = 'DATABASE'
```

```
GO
```

**Now create a table and see the order**

```
CREATE TABLE Employee1(ID INT, Name VARCHAR(100))
```

**When we execute the above create table statement, it will give us the below output.**

```
Server Scope DDL Trigger
Database Scope DDL Trigger1
Database Scope DDL Trigger
```

If we have both a database-scoped and a server-scoped DDL trigger handling the same event and if we have to set the execution order at both levels. Here is the execution order of the triggers.

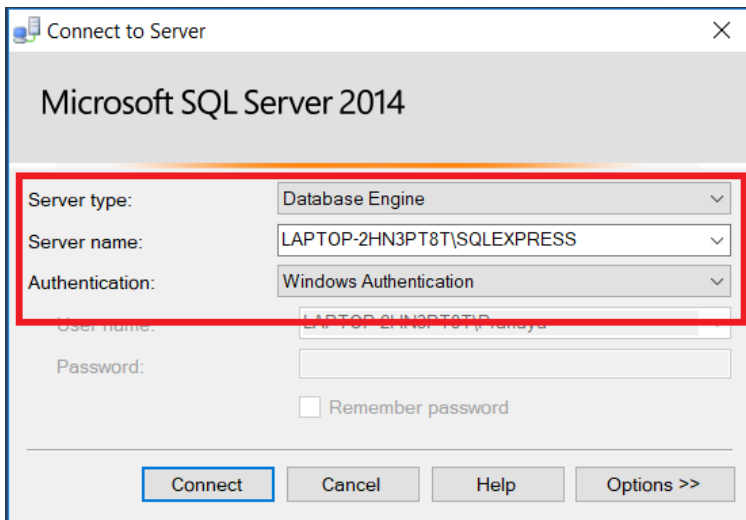
1. The server-scope trigger set as First
2. Then other server-scope triggers
3. Next, the server-scope trigger set as Last
4. The database-scope trigger set as First
5. Then other database-scope triggers
6. The database-scope trigger set as Last

---

## **Topic7: Creating and Managing Users in SQL Serve**

In this article, I am going to discuss **Creating and Managing Users in SQL Server** step by step. In this article, we are just going to see how to create a new user and how to reset the password of an existing user. In our upcoming articles, we will discuss these things in detail.

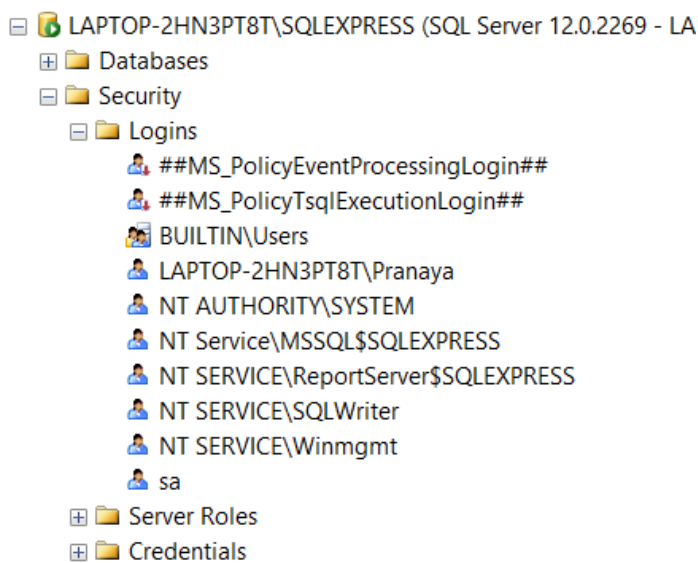
Let's first log in to the SQL Server using the Run as administrator mode using SSMS (SQL Server Management Studio) as shown in the below image.



Select the Server Type as Database Engine, provide the Server name and select the Authentication as Windows Authentication and then click on the connect button which will connect to the SQL Server Database.

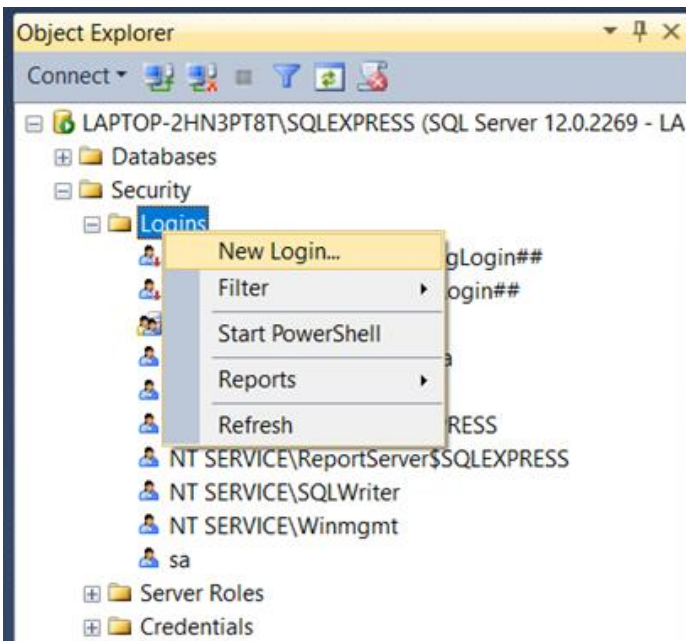
### How do we see all the users of the server?

To see all the users of the server navigate to the **Security->Logins** folder on the left side of your window as shown below.



### How to create a new user in SQL server?

To create a new user, Right-click on the Logins folder and click on the New Login option as shown in the below image

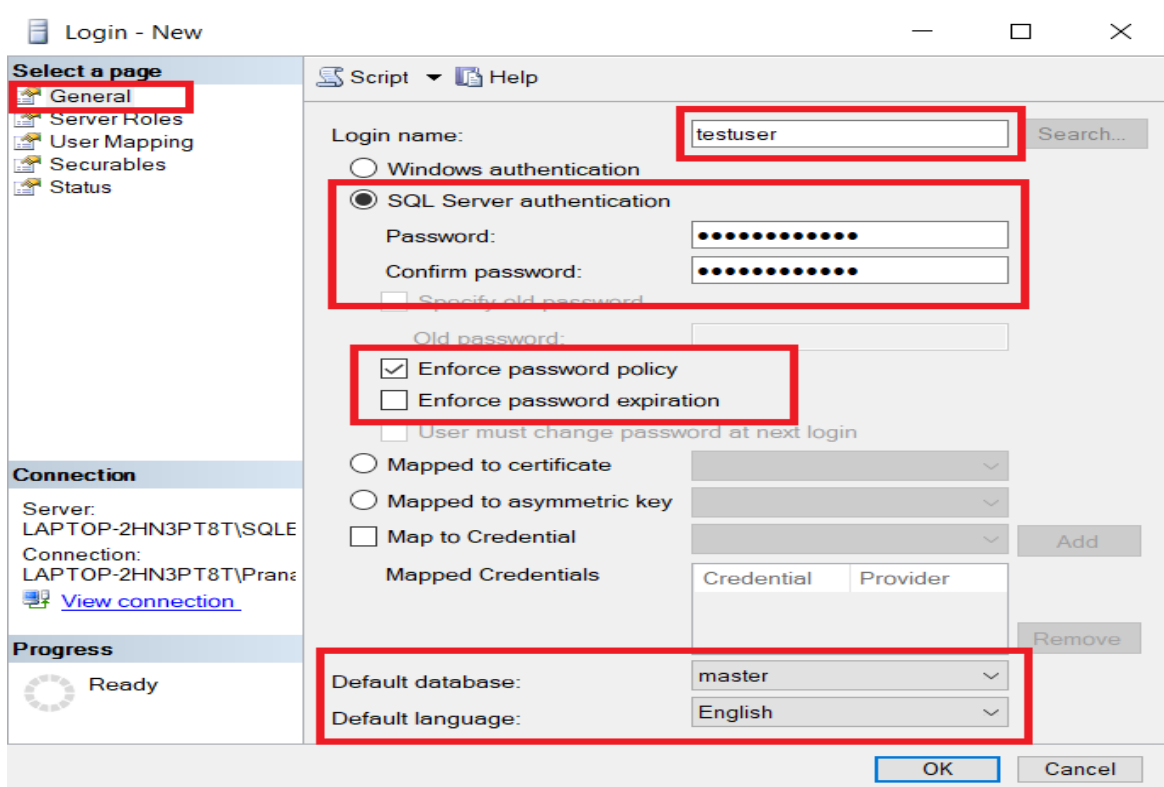


Once you click on the New Login option it will open a new popup as shown below.

### General Tab:

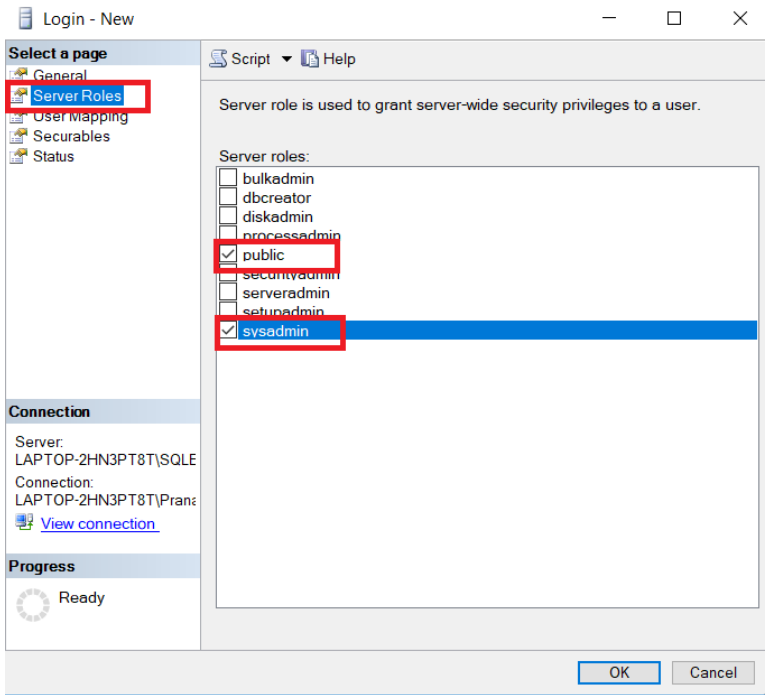
Select the General Tab and provide the below details

1. Login Name: testuser
2. Select the SQL Server Authentication and provide the Password and Confirm Password
3. Check and Uncheck the Enforce Password Policy and Enforce Password Expiration.
4. Then select the default Database and default Language as shown in the below image,



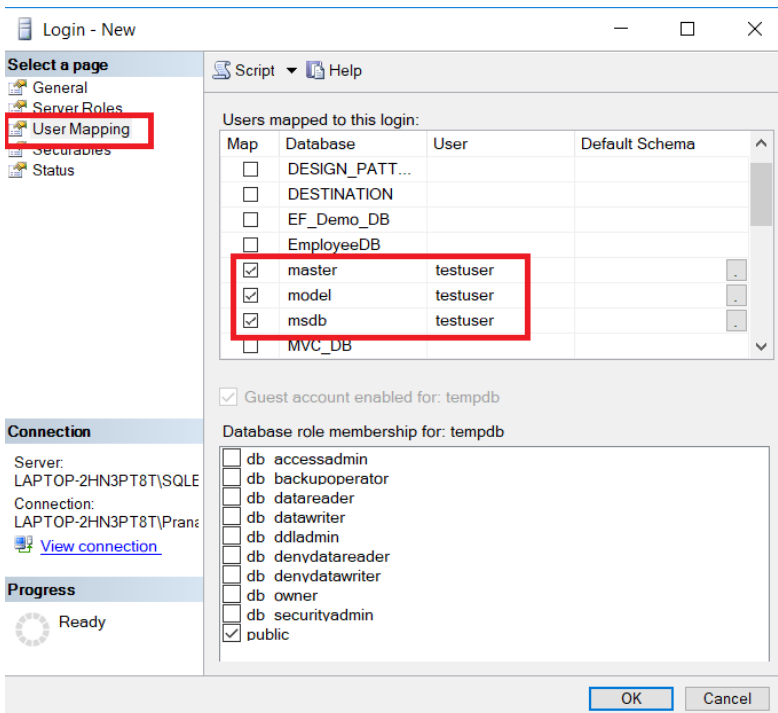
### Server Roles:

Select the Server Roles Tab and then check the public and sysadmin check box as shown in the below image.



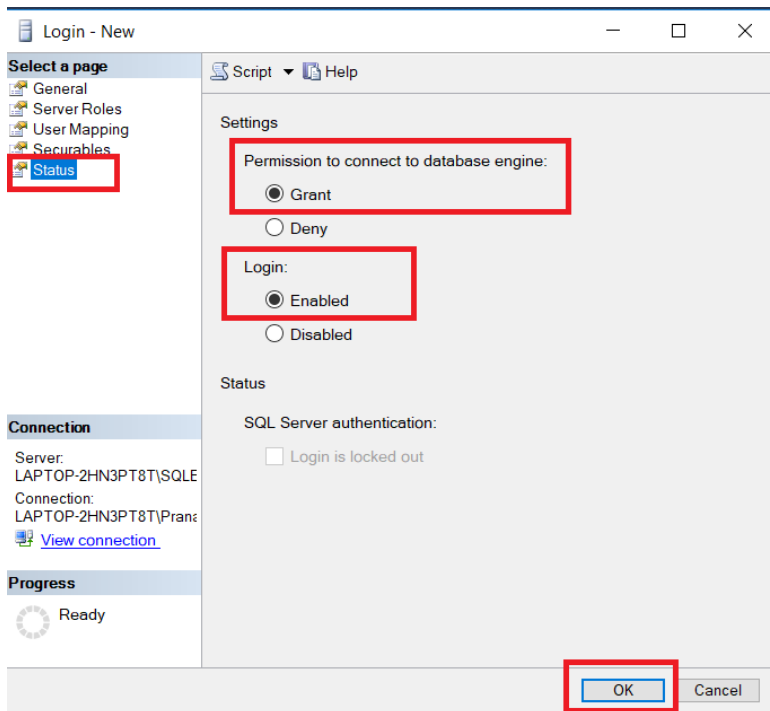
## User Mapping:

Then select the user mapping tab and check the databases to which the above user can access as shown below along with all the system databases such as tempdb, master, model, and msdb.

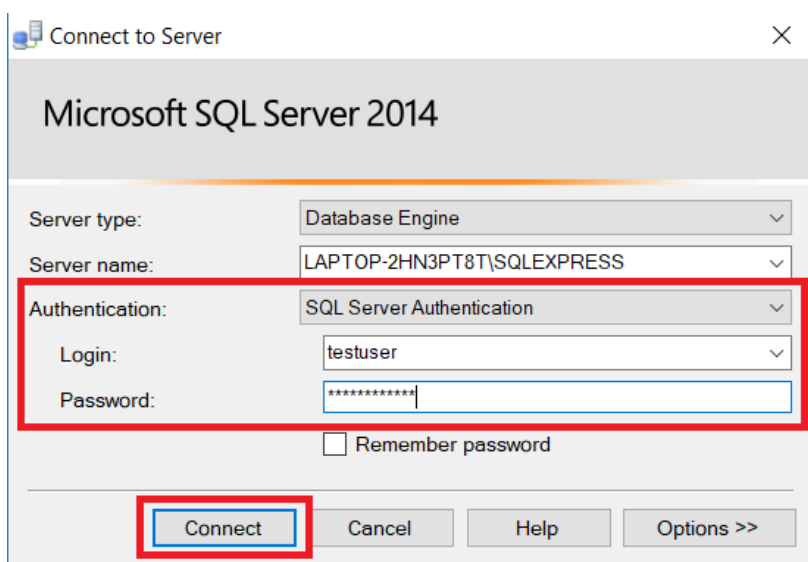


## Status Tab:

From the status tab, select the Grant and Enabled option and then click on the OK button as shown below.



That's it. We have created the testuser and you can see the testuser under the **Security => Logins** folder. Let's log in to the server using the testuser as shown in the below image.

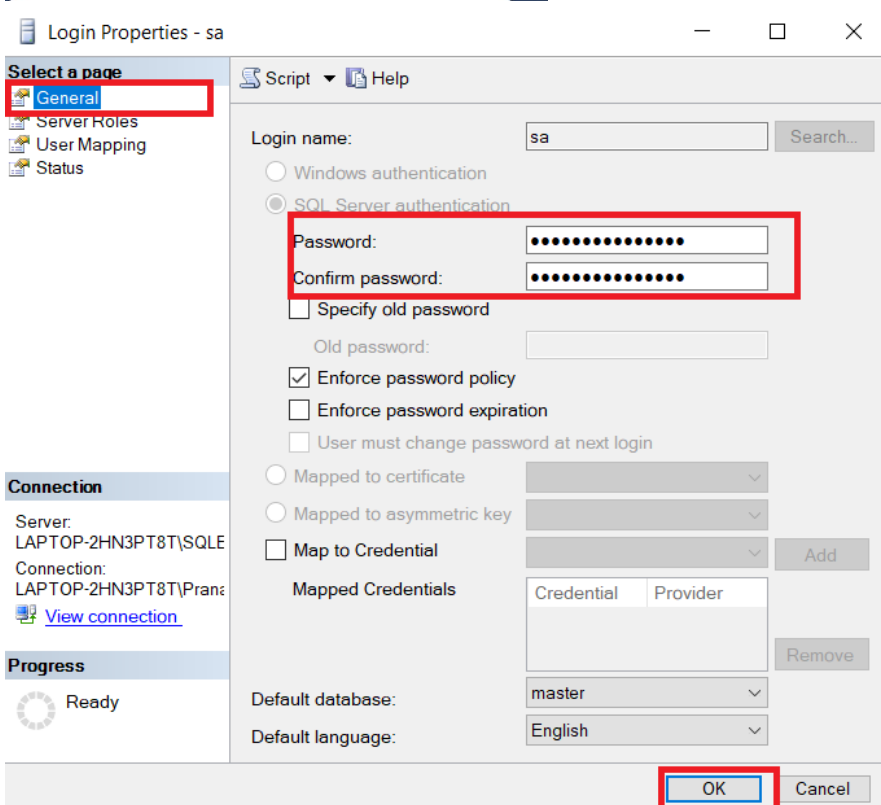
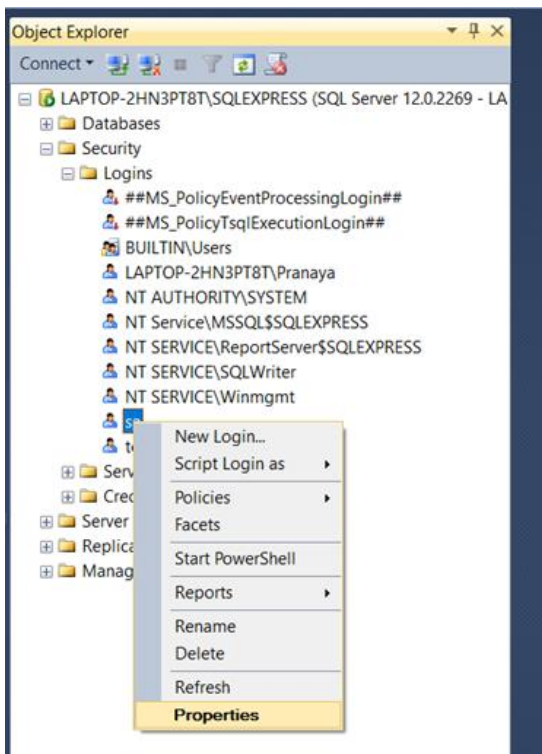


Here we need to select the Authentication type as SQL Server Authentication, provide the Login Name as testuser and the password, and then click on the connect button which will connect to the database successfully.

### How to Reset the Password of an existing user?

As we know by default the sa user is created when we installed the database. If you forgot the password for sa user or any other user then how you can reset the password.

To reset the password of a particular user, Right-click on that user and click on the properties tab as shown in the below image which will open the properties for that user where you can change the password.



Here you need to reset the password and once you reset the password then click on the OK button. So in this article, we discussed how to create a new user and how to change the password of an existing user in the SQL server. Here we create a new user testuser and we change the password of sa user.

## **Topic8: Logon Triggers in SQL Server with Examples**

1. **What are Logon Triggers in SQL Server?**
2. **Why we need Logon Triggers?**



3. **How we can create a Logon Trigger?**
4. **Real-time Examples of Logon Trigger**

**Note:** The Logon Triggers are DDL Triggers and they are created at the Server Level. They are introduced in SQL Server 2005 SP2.

### What is Logon Trigger in SQL Server?

The Logon Triggers in SQL Server are the special kind of stored procedure or we can also say a special type of operation which fire or executed automatically in response to a LOGON event and moreover, we can define more than one Logon trigger on the server.

The Logon triggers are fired only after the successful authentication but before the user session is actually established. If the authentication is failed then the logon triggers will not be fired.

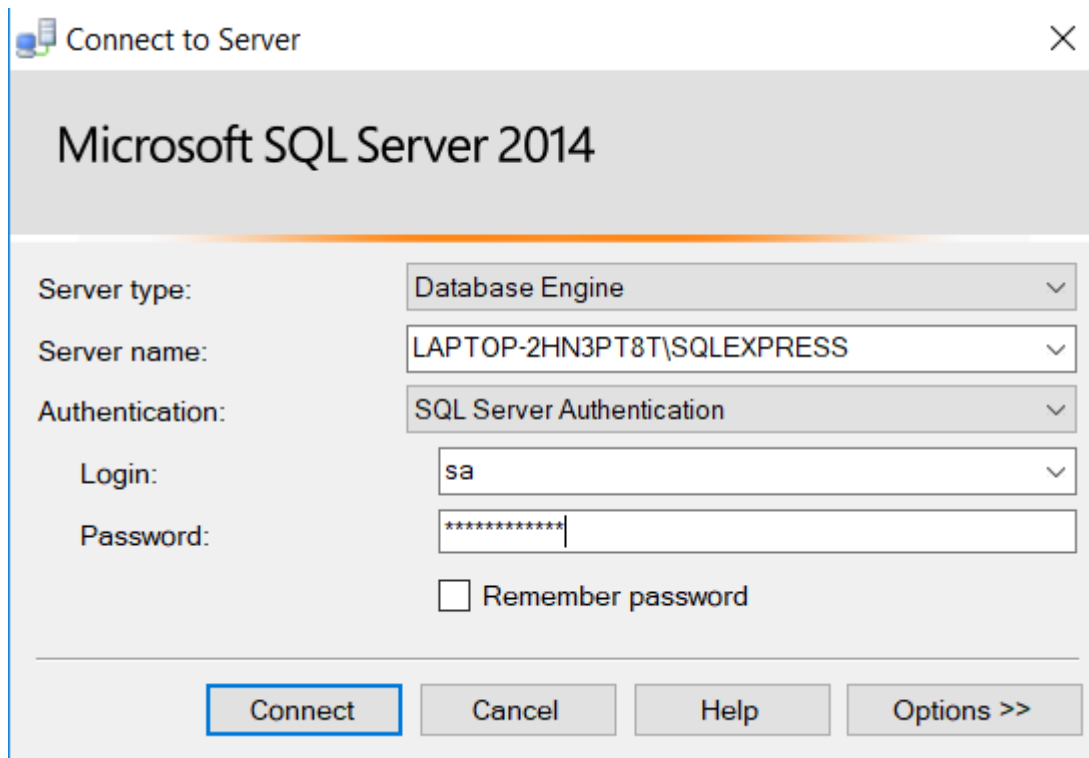
### Why we need Logon Trigger in SQL Server?

The Logon Triggers in SQL Server are commonly used to audit and control the server sessions such as

1. Tracking the Login Activity
2. Limiting the number of concurrent sessions for a single user
3. Restricting logins to SQL Server based on time of day, hostnames, application names

### Understanding the Logon Triggers with an example.

First, log in to the SQL Server in administrator mode using admin user as shown below.



Connect to Server

Microsoft SQL Server 2014

Server type: Database Engine

Server name: LAPTOP-2HN3PT8T\SQLEXPRESS

Authentication: SQL Server Authentication

Login: sa

Password: \*\*\*\*\*

☐ Remember password

Connect Cancel Help Options >>

### Creating Logon Trigger in SQL Server:

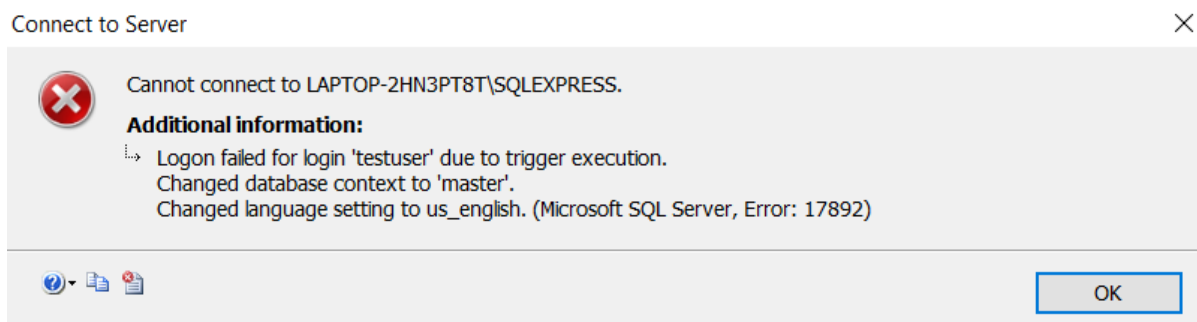
The following logon trigger will limit the maximum number of open connections for a user to 2 except the sa user. That means the following logon trigger will limit the open connections for all Logins except the 'sa' user once the open connection limit is reached to 2, then the user will not be able to create a new connection. Instead, the user will get an error message.

```

CREATE TRIGGER tr_Conn_Limit_LogonTriggers
ON ALL SERVER WITH EXECUTE AS 'sa'
FOR LOGON
AS
BEGIN
DECLARE @LoginName NVARCHAR(100)
SET @LoginName = ORIGINAL_LOGIN()
IF @LoginName <> 'sa'
AND
( SELECT COUNT(*)
FROM sys.dm_exec_sessions
WHERE Is_User_Process = 1 AND
Original_Login_Name = @LoginName
) > 2
BEGIN
PRINT 'Third session for the user ' + @LoginName + ' is blocked'
ROLLBACK
END
END

```

So let's login with the testuser user and try to open more than 3 connections and while you are creating the 3<sup>rd</sup> connection we get the below error message.



But you can open as many connections as you want using the sa user as there is no restriction for the sa user with the logon trigger. You can find the trigger error messages which will be written to the error log by executing the following command.

### Execute sp\_readerrorlog

It will give the following output.

LogDate	Processl	Text
2018-10-14 12:41:27.460	spid56	Third Connection for the user testuser is blocked
2018-10-14 12:41:27.460	spid56	Error: 3609, Severity: 16, State: 2.
2018-10-14 12:41:27.460	spid56	The transaction ended in the trigger. The batch has be...

To delete the Logon Trigger use the following command

```
DROP TRIGGER tr_Conn_Limit_LogonTriggers ON ALL SERVER
```

### Example2:

The following logon trigger will block all the users except the sa user from connecting to SQL Server after office hours.

```
CREATE TRIGGER tr_Limit_Connection_After_Office_Hours
```

```
ON ALL SERVER WITH EXECUTE AS 'sa'
```

```
FOR LOGON
```

```
AS
```

```
BEGIN
```

```
DECLARE @LoginName NVARCHAR(100)
```

```
SET @LoginName = ORIGINAL_LOGIN()
```

```
IF @LoginName <> 'sa' AND
```

```
(DATEPART(HOUR, GETDATE()) < 9 OR
```

```
DATEPART (HOUR, GETDATE()) > 18)
```

```
BEGIN
```

```
PRINT 'You are not authorized to login after office hours'
```

```
ROLLBACK
```

```
END
```

```
END
```

### Example3:

Create a logon trigger that only allows only the whitelisted hostnames to connect to the server.

```
CREATE TRIGGER tr_Restrictied_Host_Only
```

```
ON ALL SERVER
```

```
FOR LOGON
```

```
AS
```

```
BEGIN
```

```
IF
```

```
(
```

```
-- White list of allowed hostnames are defined here.
```

```
HOST_NAME() NOT IN ('DevHost','QAHost','UATHost','ProdHost')
)

BEGIN

PRINT 'You are not allowed to login from this hostname.'

ROLLBACK;

END

END
```

### Logon Trigger Real-Time Example:

Please use below SQL Script to create the Database, the Database Table and the Logon Trigger to audit the Logon Data.

-- Creates LogonAuditDB database for storing the audit data

```
CREATE DATABASE LogonAuditDB

USE LogonAuditDB

GO
```

-- Creates TableAudit table for logons inside LogonAuditDB

```
CREATE TABLE TableLogonAudit
(
SessionId int,
LogonTime datetime,
HostName varchar(50),
ProgramName varchar(500),
LoginName varchar(50),
ClientHost varchar(50)
)
```

**GO**

-- Create Logon trigger for storing the User login data

```
CREATE TRIGGER LogonAuditTrigger
ON ALL SERVER
FOR LOGON
AS
BEGIN
DECLARE @LogonTriggerData xml,
@EventTime datetime,
@LoginName varchar(50),
```

```

@ClientHost varchar(50),

@LoginType varchar(50),

@HostName varchar(50),

@AppName varchar(500)

SET @LogonTriggerData = eventdata()

SET @EventTime = @LogonTriggerData.value('/EVENT_INSTANCE/PostTime')[1], 'datetime')

SET @LoginName = @LogonTriggerData.value('/EVENT_INSTANCE/LoginName')[1], 'varchar(50)')

SET @ClientHost = @LogonTriggerData.value('/EVENT_INSTANCE/ClientHost')[1], 'varchar(50)')

SET @HostName = HOST_NAME()

SET @AppName = APP_NAME()

INSERT INTO LogonAuditDB.dbo.TableLogonAudit

(

SessionId,

LogonTime,

HostName,

ProgramName,

LoginName,

ClientHost

)

VALUES

(

@@spid,

@EventTime,

@HostName,

@AppName,

@LoginName,

@ClientHost

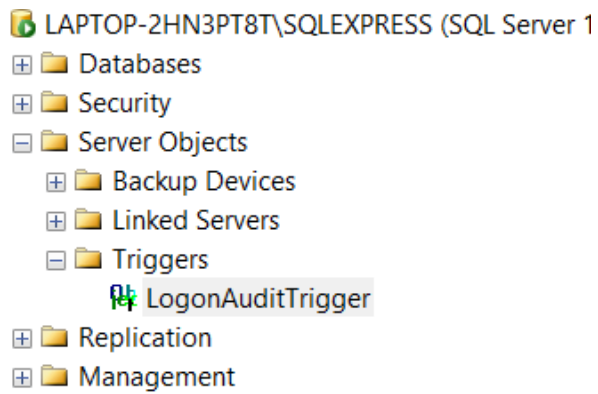
)

END

GO

```

**You can find the Trigger inside the Triggers folder which is inside the Server Object Folder as shown in the below image.**



The **EVENTDATA()** in SQL Server is an XML document that can be only available within the context of a DDL Trigger and It has the following schema

```
<EVENT_INSTANCE>
<EventType>event_type</EventType>
<PostTime>post_time</PostTime>
<SPID>spid</SPID>
<ServerName>server_name</ServerName>
<LoginName>login_name</LoginName>
<LoginType>login_type</LoginType>
<SID>sid</SID>
<ClientHost>client_host</ClientHost>
<IsPooled>is_pooled</IsPooled>
</EVENT_INSTANCE>
```

You can view the logon audit data by using the below SQL Query.

```
SELECT * FROM LogonAuditDB.dbo.TableLogonAudit
```