# Triggers, Transactions and Stored Procedures

Jyotishman Sarkar

## Agenda

- Introduction & Motivation
- Section 1: Triggers
- Section 2: Transactions
- Section 3: Stored Procedures
- Best Practices & Pitfalls
- Q&A

## Why Automate SQL Logic?

- Manual SQL code scattered across applications can be error-prone and lead to inconsistent behavior .
- Embedding business rules at the database layer ensures they’re enforced uniformly, regardless of client or interface .
- Centralizing logic in triggers, transactions, and stored procedures reduces code duplication and simplifies maintenance .
- Executing logic close to the data minimizes network round-trips, improving throughput and overall performance
- Leveraging built-in SQL Server constructs enhances security by controlling who can execute operations instead of granting direct table access

## Section 1: Triggers

What is a Trigger?

- A trigger is a special kind of stored procedure that fires automatically.
- It responds to Data Manipulation Language (DML) or Data Definition Language (DDL) events.
- Common uses: auditing, cascading changes, complex validations.

## Types of Triggers in SQL Server

| Trigger Type | Fire On | Purpose |
|---|---|---|
| **DML Triggers** | INSERT, UPDATE, DELETE on tables/views | Track or validate data changes and make edits |
| **DDL Triggers** | CREATE, ALTER, DROP statements | Monitor schema changes |
| **Logon Triggers** | When a user logs into SQL Server | Control or audit login activity |

# DML Triggers

- DML (Data Manipulation Language) triggers are special stored procedures that automatically execute in response to INSERT, UPDATE, or DELETE operations on a table or view.
- They help enforce business rules, audit changes, or maintain data integrity.

| Trigger Type | Description |
|---|---|
| AFTER Trigger | Executes **after** the DML operation completes. |
| INSTEAD OF Trigger | Executes **in place of** the DML operation. Useful for views or custom logic. |

## Examples

### AFTER Trigger

```
CREATE OR ALTER TRIGGER trg_Emps_SalaryAudit
ON Emps
AFTER UPDATE
AS
BEGIN
    INSERT INTO EmployeeAudit (EmployeeID, OldSalary, NewSalary)
    SELECT
        d.Empno,
        d.Salary,
        i.Salary
    FROM
        deleted AS d
    INNER JOIN
        inserted AS i
      ON d.Empno = i.Empno
    WHERE
        ISNULL(d.Salary, 0) <> ISNULL(i.Salary, 0);
END;
```

### INSTEADOF Trigger

```
CREATE TRIGGER trg_InsteadOfDelete_Customers
ON Customers
INSTEAD OF DELETE
AS
BEGIN
  UPDATE Customers
  SET IsActive = 0
  WHERE CustomerID IN (SELECT CustomerID FROM deleted);
END;
```

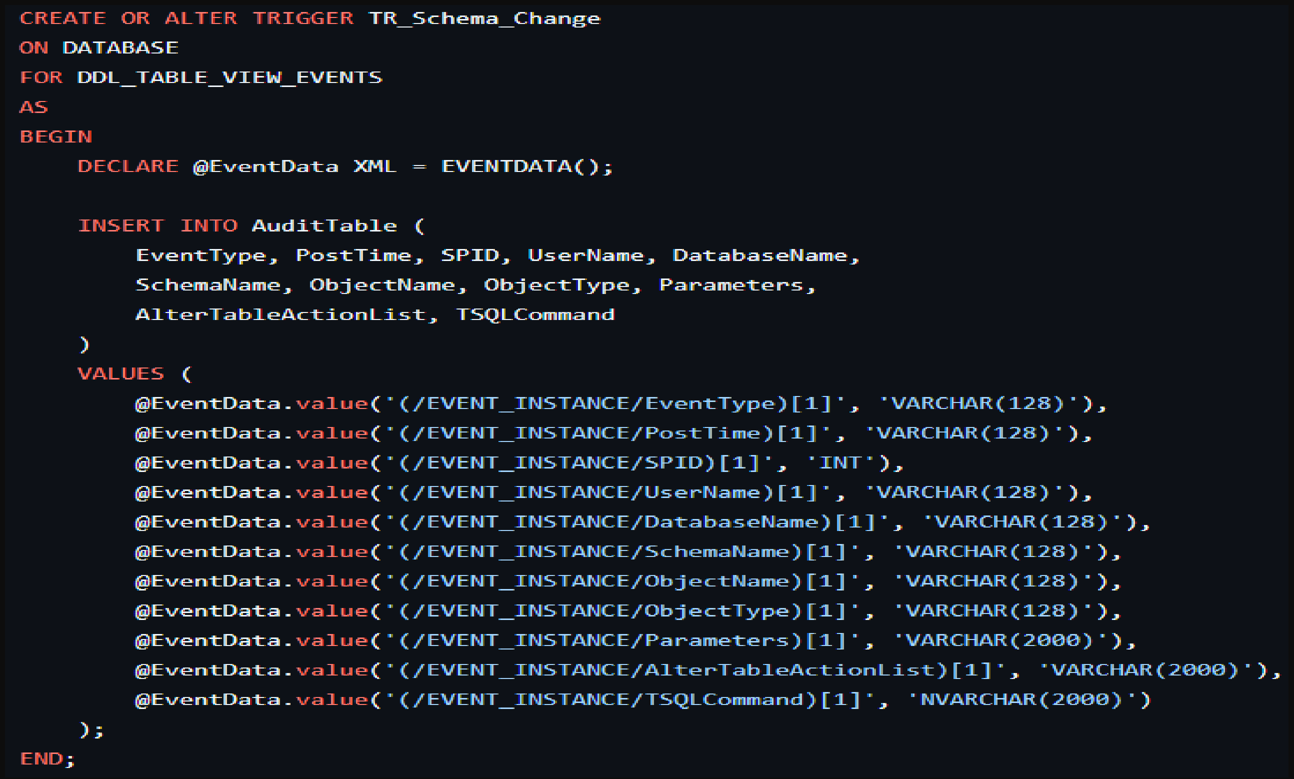
## DDL Triggers

- These respond to schema changes such as:
  - CREATE, ALTER, DROP (tables, views, procedures, etc.)
  - GRANT, REVOKE, DENY

- Scope:
  - Database Scoped: Applies to changes within a specific database.
  - Server Scoped: Applies to changes at the server level.

Example use: Prevent unauthorized schema changes, log structural modifications.

## Examples



```sql
CREATE OR ALTER TRIGGER TR_Schema_Change
ON DATABASE
FOR DDL_TABLE_VIEW_EVENTS
AS
BEGIN
    DECLARE @EventData XML = EVENTDATA();

    INSERT INTO AuditTable (
        EventType, PostTime, SPID, UserName, DatabaseName,
        SchemaName, ObjectName, ObjectType, Parameters,
        AlterTableActionList, TSQLCommand
    )
    VALUES (
        @EventData.value('(/EVENT_INSTANCE/EventType)[1]', 'VARCHAR(128)'),
        @EventData.value('(/EVENT_INSTANCE/PostTime)[1]', 'VARCHAR(128)'),
        @EventData.value('(/EVENT_INSTANCE/SPID)[1]', 'INT'),
        @EventData.value('(/EVENT_INSTANCE/UserName)[1]', 'VARCHAR(128)'),
        @EventData.value('(/EVENT_INSTANCE/DatabaseName)[1]', 'VARCHAR(128)'),
        @EventData.value('(/EVENT_INSTANCE/SchemaName)[1]', 'VARCHAR(128)'),
        @EventData.value('(/EVENT_INSTANCE/ObjectName)[1]', 'VARCHAR(128)'),
        @EventData.value('(/EVENT_INSTANCE/ObjectType)[1]', 'VARCHAR(128)'),
        @EventData.value('(/EVENT_INSTANCE/Parameters)[1]', 'VARCHAR(2000)'),
        @EventData.value('(/EVENT_INSTANCE/AlterTableActionList)[1]', 'VARCHAR(2000)'),
        @EventData.value('(/EVENT_INSTANCE/TSQLCommand)[1]', 'NVARCHAR(2000)')
    );
END;
```

## LogOn Triggers

- These fire when a user logs into SQL Server.
  - Used to control or monitor login activity.
  - Can restrict access based on time, IP, or user role.

Example use: Enforce login policies, block logins during maintenance.

## Examples

```sql
CREATE TRIGGER Logon_By_IP
ON ALL SERVER
FOR LOGON
AS
BEGIN
    DECLARE @LoginName NVARCHAR(100) = ORIGINAL_LOGIN();
    DECLARE @ClientHost NVARCHAR(128) = EVENTDATA().value('(/EVENT_INSTANCE/ClientHost)[1]', 'NVARCHAR(128)');

    -- Block login if IP is not in the allowed list
    IF @LoginName = 'testuser' AND @ClientHost NOT IN ('192.168.1.100', '10.0.0.5')
    BEGIN
        ROLLBACK;
    END
END;
```

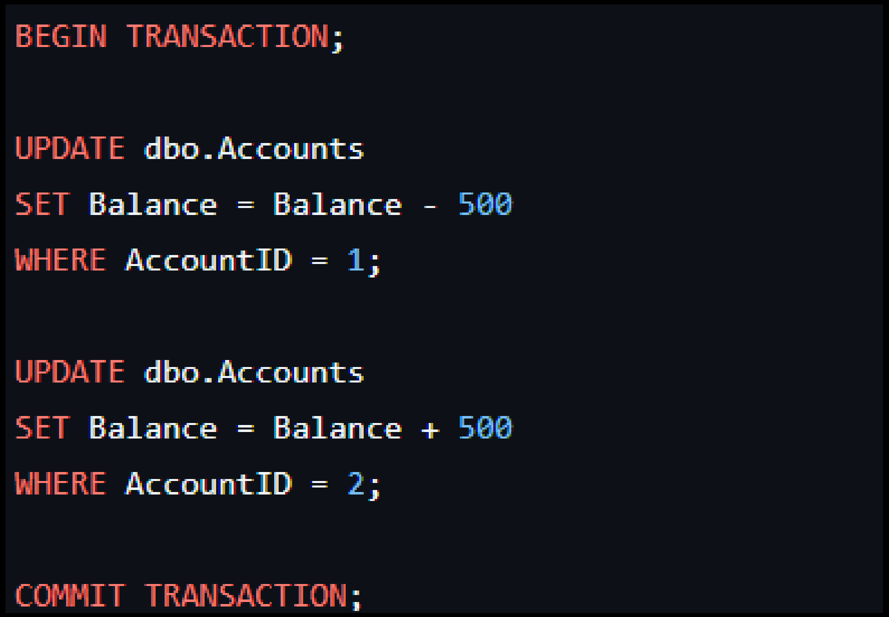
## Trigger Best Practices

- Keep logic lightweight; avoid long-running operations.
- Use SET NOCOUNT ON to reduce network chatter.
- Be cautious with nested and recursive triggers.
- Document side effects clearly.

## Section 2: Transactions

- A transaction groups operations into a single, atomic unit.

- Ensures ACID properties:
    1. Atomicity
    2. Consistency
    3. Isolation
    4. Durability

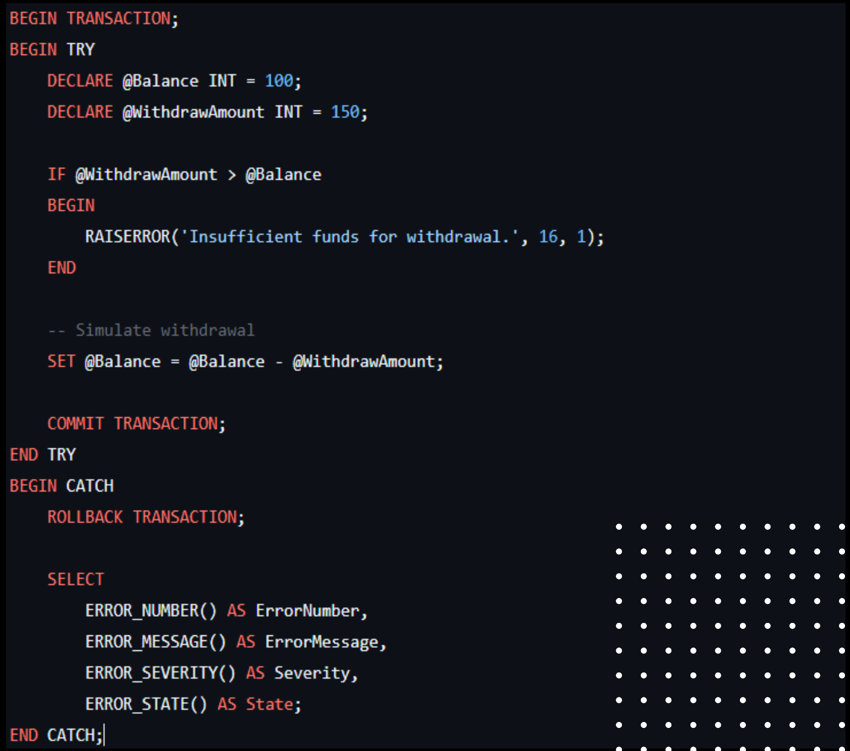


```sql
BEGIN TRANSACTION;

UPDATE dbo.Accounts
SET Balance = Balance - 500
WHERE AccountID = 1;

UPDATE dbo.Accounts
SET Balance = Balance + 500
WHERE AccountID = 2;

COMMIT TRANSACTION;
```

## Basic Transaction Commands

# Error Handling & Rollback

- **BEGIN TRY** encloses safe operations.
- **BEGIN CATCH** handles exceptions.
- **ROLLBACK** undoes all changes if an error occurs.
- **ERROR_MESSAGE()** returns the error text.
- **RAISERROR** statement in SQL Server is a powerful tool for generating custom error messages
- **SAVE can define save-points to roll back only part of the transaction without aborting the whole unit.**



```sql
BEGIN TRANSACTION;
BEGIN TRY
    DECLARE @Balance INT = 100;
    DECLARE @WithdrawAmount INT = 150;

    IF @WithdrawAmount > @Balance
    BEGIN
        RAISERROR('Insufficient funds for withdrawal.', 16, 1);
    END

    -- Simulate withdrawal
    SET @Balance = @Balance - @WithdrawAmount;

    COMMIT TRANSACTION;
END TRY
BEGIN CATCH
    ROLLBACK TRANSACTION;

    SELECT
        ERROR_NUMBER() AS ErrorNumber,
        ERROR_MESSAGE() AS ErrorMessage,
        ERROR_SEVERITY() AS Severity,
        ERROR_STATE() AS State;
END CATCH;
```

## Use Cases of Transactions

| Use Cases | Descriptions |
|---|---|
| **Financial Transfers** | Ensures that debit and credit operations both succeed or both fail. |
| **Batch Billing & Invoicing** | Processes multiple charges atomically to prevent partial billing. |
| **Data Migrations** | Moves or transforms data across tables safely, avoiding corruption. |
| **Referential Integrity Enforcement** | Inserts/updates across related tables (e.g., Orders and OrderDetails). |
| **Inventory Adjustments** | Updates stock levels and logs changes in audit tables simultaneously. |

## Section 3: Stored Procedures

- **What Is a Stored Procedure?**

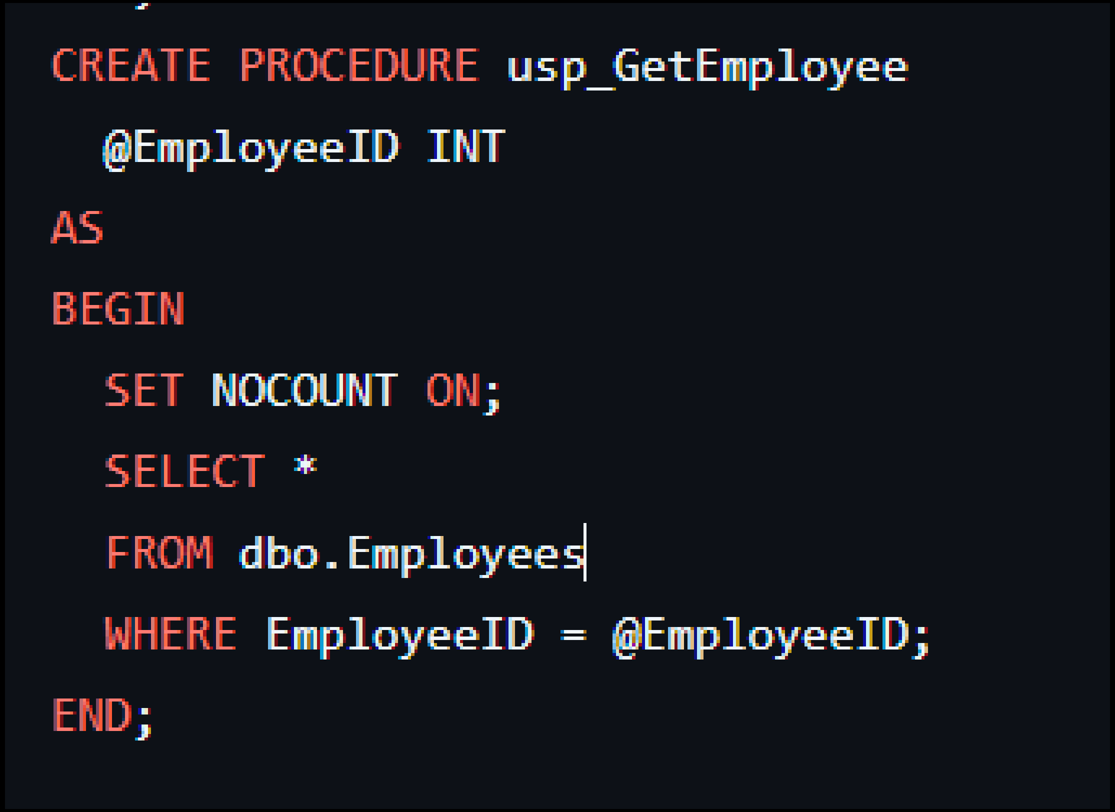
A stored procedure is a precompiled collection of SQL statements stored in the database. It can accept parameters, perform operations, and return results.

- **Key Features**

| **Features** | **Description** |
|---|---|
| **Encapsulation** | Groups logic into reusable blocks |
| **Performance** | Precompiled for faster execution |
| **Security** | Can restrict direct table access |
| **Maintainability** | Easier to update logic in one place |
| **Testability** | Can be tested independently from application code |
| **Reduce Network Traffic** | Only single call to server |

## Creating a Stored Procedure

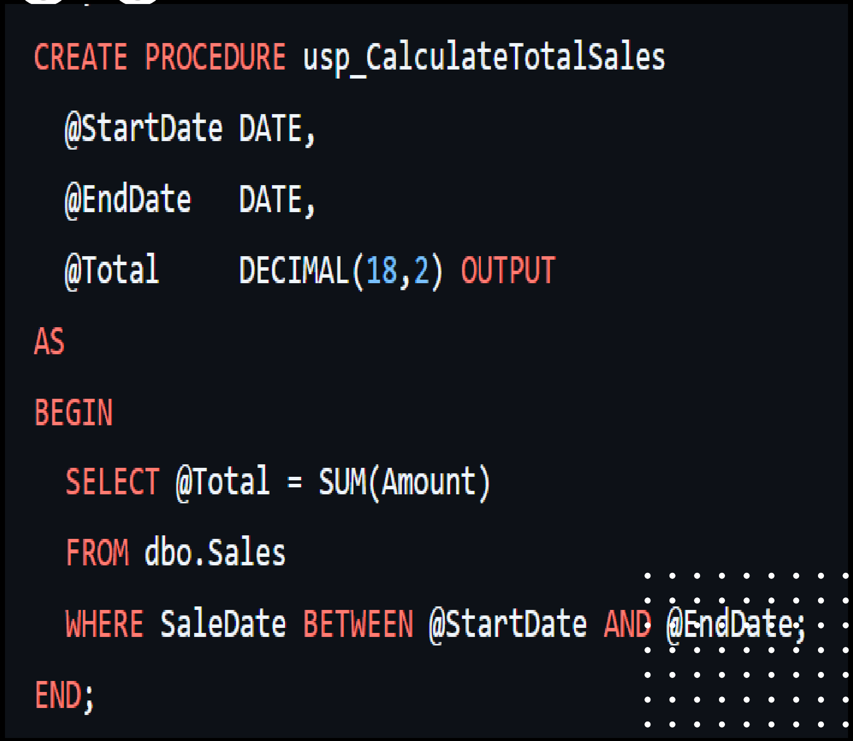
- EXEC or EXECUTE command used to execute this stored procedure.
- EXEC usp_GetEmployee @EmployeeID = 101;



```sql
CREATE PROCEDURE usp_GetEmployee
  @EmployeeID INT
AS
BEGIN
  SET NOCOUNT ON;
  SELECT *
  FROM dbo.Employees
  WHERE EmployeeID = @EmployeeID;
END;
```

## Input & Output Parameters

- Defined in procedure header.
- INPUT Parameters any data type (including UDTTs).
- OUTPUT Parameters must be defined with OUTPUT keyword.
- Calling Code must also declare variable OUTPUT.
- OUTPUT can return scalar (count, status, messages).



```sql
CREATE PROCEDURE usp_CalculateTotalSales
  @StartDate DATE,
  @EndDate   DATE,
  @Total     DECIMAL(18,2) OUTPUT
AS
BEGIN
  SELECT @Total = SUM(Amount)
  FROM dbo.Sales
  WHERE SaleDate BETWEEN @StartDate AND @EndDate;
END;
```

## Executing , Permissions & System’s SPs

- EXEC or EXECUTE share same functionalities
- GRANT used to grant specific user access to the stored program
  - Like, GRANT EXECUTE ON usp_Procedure TO [UserOrRoleName].
- Some Simple system stored procedures

| Procedure | Purpose |
|---|---|
| **sp_help** | Shows details about a database object (table, view, etc.) |
| **sp_helptext** | Displays the source code of a stored procedure, view, or function |
| **sp_rename** | Renames a database object |
| **sp_tables** | Lists all tables in the current database |
| **sp_columns** | Lists all columns of a specified table |

## Case Study: SPs vs UDFs

| Feature | Stored Procedure (usp_) | User-Defined Function (UDF) |
|---|---|---|
| Purpose | Perform actions (DML, logic, control flow) | Return a value or table; used in expressions |
| Return Type | Can return multiple result sets or none | Must return a scalar value or a table |
| Execution | Invoked using EXEC or EXECUTE | Used in SELECT, WHERE, JOIN, etc. |
| Output Parameters | Supports OUTPUT parameters | No OUTPUT parameters; returns via RETURN |
| Side Effects (INSERT/UPDATE) | Allowed | Not allowed (must be deterministic) |
| Use in Queries | Cannot be used directly in SELECT statements | Can be used inline in queries |
| Error Handling | Supports TRY...CATCH | Limited error handling |
| Transaction Control | Can manage transactions (BEGIN, COMMIT, etc.) | Cannot manage transactions |
| Performance | Slightly more flexible, less restricted | Optimized for reusable logic in queries |
| Use Case | Complex logic, batch operations, procedural tasks | Calculations, reusable expressions, table filters |

## Stored Procedure Best Practices

- Use meaningful names prefixed with usp_.
- Avoid excessive dynamic SQL; prefer parameterization.
- Handle exceptions with TRY…CATCH and THROW.
- Comment purpose, parameters and side effects.

## Summary and Q&A

- Triggers automate responses to data events.
- Transactions enforce consistency with ACID.
- Stored procedures encapsulate and secure logic.

Thank You