TRIGGERS

Triggers are the custom apex code which will fire automatically based on the specified DML events.

By using triggers, we can implement custom validations, complex validations, de-duplication processes, complex business logic and complex transactional flows also.

Each Trigger should be associated with an object. But we can create one/more triggers on an object.

Syntax: Trigger <TriggerName> ON <ObjectName>(<Events List>)

```
{
        // Write the Business Logic
}
```

Ex: Trigger AccountsTrigger ON Account(<Events List>)

```
{
        // Write the Business Logic
}
```

```
Trigger OpportunityTrigger ON Opportunity (<Events List>)
{
        // Write the Business Logic
}
```

```
Trigger PositionsTrigger ON Position__C(<Events List>)
{
        // Write the Business Logic
}
```

Each trigger code will reside inside the file with the extension .apxt. All the triggers information will be stored inside "Apex Trigger" object.

Trigger Events: Event is nothing but an occasion/situation which describes when the trigger should fire.

Salesforce provides the below 7 Events for the trigger.

1. **Before Insert**: This event will fire the trigger before inserting the record inside the object.

Ex: Trigger AccountsTrigger ON Account (Before Insert)

```
{
        // Write the Business Logic
```

}

2. **Before Update**: This event will fire the trigger before updating the record inside the object.

Ex: Trigger CaseTrigger ON Case (Before Update)

{

// Write the Business Logic

}

Trigger LeadTrigger ON Lead (Before Insert, Before Update)

{

// Write the Business Logic

}

3. **Before Delete**: This event will fire the trigger before deleting the record from the object.

Ex: Trigger AccountsTrigger ON Account (Before Delete)

{

// Write the Business Logic

}

4. **After Insert**: This Event will fire the trigger once a new record has been inserted inside the object.

Ex: Trigger ContactsTrigger ON Contact (After Insert)

{

// Write the Business Logic

}

5. **After Update**: This event will fire the trigger, when the record has been updated successfully inside the object.

Ex: Trigger HiringManagerTrigger ON Hiring_Manager__C(After Update)

{

// Write the Business Logic

}

6. **After Delete**: This Event will fire the trigger after removing a record from the object.

Ex: Trigger PositionsTrigger ON Position__C(After Delete)

{

// Write the Business Logic

}

7. **After Undelete**: This event will fire the trigger on after restoring the record back to the actual object.

Ex: Trigger LeadTrigger ON Lead(After UnDelete)

```
{
        // Write the Business Logic
}
```

Ways to Create the Trigger

Apex provides the below 2 ways to Create the Triggers.

1. By using "Standard Navigation".

    Click on "Setup" menu.

    1. Search for the option "Apex Triggers".
    2. Click on "New" button
    3. Create a Trigger.

2. By using "Developer Console".

    Click on "Setup ---> Developer Console".

    1. Go to the "Developer Console" Editor.
    2. Click on "File ---> New ---> Apex Triggers".
    3. Enter the Trigger Name in the Textbox.
    4. Select the "Object Name" from the Picklist.
    5. Click on "Submit" button.

    Observation: It will Create a Trigger on the Object.

    6. Write the Business Logic inside the Trigger.
    7. Save the Trigger by using "CTRL+S".

Trigger Context Variables

Trigger Context Variables are used to check the status of the trigger based on which we can execute the associated block of statements inside the trigger i.e., by using Context Variables, we can divide the trigger code into the various smaller pieces based on the event.

Apex provides the below 12 Trigger Context Variables.

1. Boolean Trigger.IsInsert: It returns TRUE, if the trigger has been fired because of an Insert operation else it returns FALSE.

2. Boolean Trigger.IsUpdate: It returns TRUE, if the trigger has been fired because of an Update operation else it returns FALSE.

3. Boolean Trigger.IsDelete: It returns TRUE, if the trigger has been fired because of Delete operation else it returns FALSE.

4. Boolean Trigger.IsUnDelete: It returns TRUE, if the trigger has been fired because of UnDelete operation else it returns FALSE.

5. Boolean Trigger.IsBefore: It returns TRUE, if the trigger is about to perform the operations, else it returns FALSE.

6. Boolean Trigger.IsAfter: It returns TRUE, if the trigger has been done with the operation, else it returns FALSE.

Ex: Trigger LeadTrigger ON Lead (Before Insert, After Insert)

```
{
        if (Trigger.isInsert && Trigger.isBefore)
        {
                // Write the Code for the Validation.
        }

        if(Trigger.isInsert && Trigger.isAfter)
        {
                // Write the Code, to Create a Related Prospect Record.

                // Write the Code to Send the Email Alert
        }
}
```

7. Boolean Trigger.IsExecuting: It returns TRUE if the trigger is currently executing the statements else it returns FALSE.

8. TriggerOperation Trigger.OperationType: It is an enumeration which returns the event name, on which the trigger has been fired.

It returns the event name in the form of Constants as below.

1. BEFORE_INSERT
2. BEFORE_UPDATE
3. BEFORE_DELETE
4. AFTER_INSERT
5. AFTER_UPDATE
6. AFTER_DELETE
7. AFTER_UNDELETE

Ex: Trigger LeadTrigger ON Lead (Before Insert, After Insert)

```
{
    Switch ON Trigger.OperationType
    {
        When BEFORE_INSERT
        {
            // Write the Code for the Validation.
        }
        When AFTER_INSERT
        {
            // Write the Code, to Create a Related Prospect Record.

            // Write the Code, to Send the Email Alert
        }
    }
}
```

9. Trigger.New: Trigger. New is a context variable which holds the current context records in the form of "List Collection".

Syntax: List<SObjectDataType> Trigger.New;

Trigger.New context variable will be available in both "Insert and Update Operations". It will not be available in "Delete Operation".

10. Trigger.NewMap: Trigger.NewMap is a Context variable which holds the current context records in the form of "Map Collection"

Record Id ----> Key.

Complete Record ----> Value.

Syntax: Map<Id, SObjectDataType> Trigger.NewMap;

Trigger.NewMap context variable will be available in both "After Insert, and Update Operations". It will not be available in "Delete Operation".

11. Trigger.Old: It is a Context variable, which holds the previous context (old) records in the form of "List Collection".

Syntax: List<SObjectDataType> Trigger.Old;

Trigger.Old context variable will be available in both "Update and Delete" operations. It will not be available in "Insert" operation.

12. Trigger.OldMap: It is a context variable, which holds the previous context records in the form of "Map Collection".

Record Id ----> Key.

Complete Record ---> Value.

Syntax: Map<ID, SObjectDataType> Trigger.OldMap;

Trigger.OldMap context variable will be available in both "Update and Delete" operations. It will not be available in "Insert" operation.

Trigger Bulkification / Bulkify Trigger

Upon processing the records through trigger, don't fire the trigger for each record separately.

Make sure that the trigger should fire only once for multiple records, so that the application performance can be improved. It can be implemented with the help of "Trigger Context Variables, and Iterative Statements".

Ex: Trigger AccountsTrigger ON Account (Before Insert)

```
{
    If (Trigger.isInsert && Trigger.isBefore)
    {
        For (Account acc : Trigger.New)
        {
            if(acc.Rating == Null || acc.Rating == '')
            {
                // Show the Error Message
            }
            else if(acc.Industry == Null || acc.Industry == '')
            {
                // Show the Error Message
        }
        }
        }
        }
```

Deactivating Triggers: We can deactivate the Triggers with the below Navigation.

Click on "Setup" menu.

1. Search for the option "Apex Triggers" in Quick Find Box.
2. Click on the Required Trigger Name.
3. Click on "Edit" button.
4. Unselect the Checkbox "Active".
5. Click on "Save" button.

Use Case: Create a Trigger on the Lead Object to make sure Lead Industry, Phone, Fax, Email fields are required.

Object Name: Lead Object.

Event Name : Before Insert

```
trigger LeadTrigger on Lead (before insert)
    {
    if(Trigger.isInsert && Trigger.isBefore)
    {
        for(Lead ldRecord : Trigger.New)
        {
        if(ldrecord.Industry == Null || ldRecord.Industry == '')
            ldrecord.Industry.AddError('Please Select the Industry Name.');
        else if(ldrecord.Phone == Null || ldrecord.Phone == '')
            ldRecord.Phone.AddError('Please Enter the Contact Number.');
        else if(ldRecord.Fax == Null || ldrecord.Fax == '')
            ldrecord.Fax.AddError('Please Enter the Fax Number.');
        else if(ldRecord.Email == Null || ldrecord.Email == '')
            ldrecord.Email.AddError('Please Enter Email Address.');
        }
        }
    }
```

Use Case: Create a Trigger on Account Object, to make sure Account Rating, Phone, and Annual Revenue values should be required upon creating/updating of record.

Object Name : Account Object

Event Name : Before Insert, Before Update

```
trigger AccountRecordsTrigger on Account (before insert, before update)
{
```

```
        if( Trigger.isBefore && ( Trigger.isInsert || Trigger.isUpdate) )
        {
        for(Account accRecord : Trigger.New)
        {
        if(accRecord.Rating == Null || accRecord.Rating == '')
                accRecord.Rating.AddError('Please Select the Rating Value.');
        else if(accRecord.Phone == Null || accRecord.Phone == '')
                accRecord.Phone.AddError('Please Enter Customer Contact Number.');
        else if(accRecord.AnnualRevenue == Null)
                accRecord.AnnualRevenue.AddError('Please Enter the Annual Revenue.');
                }
        }
}
```

Use Case: Create a Trigger on Account Object, to Auto Populate Annual Revenue for the Account based on Industry Name as below.

Industry Name	Annual Revenue
Banking	3500000
Finance	7200000
Insurance	5300000
Manufacturing	4600000
Education	3600000
Technology	6700000

Object Name : Account

Event Name : Before Insert, Before Update

```
trigger AutoPopulateRevenueTrigger on Account (before insert, before update)
{
        if(Trigger.isBefore && (Trigger.isInsert || Trigger.isUpdate))
        {
                for(Account accRecord : Trigger.New)
                {
                        Switch ON accrecord.Industry
```

```
                {
                    When 'Banking'
                    {
                        accRecord.AnnualRevenue = 3500000;
                    }
                    When 'Finance'
                    {
                        accRecord.AnnualRevenue = 7200000;
                    }
                    When 'Insurance'
                    {
                        accRecord.AnnualRevenue = 5300000;
                    }
                    When 'Manufacturing'
                    {
                        accRecord.AnnualRevenue = 4600000;
                    }
                    When 'Technology'
                    {
                        accRecord.AnnualRevenue = 6700000;
                    }
                    When 'Education'
                    {
                        accRecord.AnnualRevenue = 3600000;
                    }
                }
            }
        }
```

Use Case: Create a trigger on Account Object, to prevent the deletion of Active Account Records.

Object Name : Account

Event Name : Before Delete

```
trigger PreventAccountDeletionTrigger on Account (before delete)
{
    if(Trigger.isBefore && Trigger.isDelete)
    {
        for(Account accRecord : Trigger.Old)
        {
            if(accRecord.Active__c == 'Yes')
                accRecord.AddError('You Are Not Authorized to Delete an Active Account.');
        }
    }
}
```

Use Case: Create a Trigger on the Case Object, to make sure each Case Record should be associated with an Account and Contact.

Object Name : Case Object

Event Name : Before Insert, Before Update

```
trigger CaseRecordTrigger on Case (before insert, before update)
{
    if(Trigger.isBefore && (Trigger.IsInsert || Trigger.isUpdate))
    {
        for(Case csRecord : Trigger.New)
        {
        if(csRecord.AccountId == Null)
            csRecord.AccountId.AddError('Please Select the Account Record');
        else if(csRecord.ContactId == Null)
            csRecord.ContactId.AddError('Please Select the Contact Record');
        }
    }
}
```

Use Case: Create a Trigger on "Hiring Manager Object" to prevent the duplicate Records based on the combination of "Hiring Manager Name, Contact Number and Email Address".

Object Name : Hiring Manager Object

Event Name : Before Insert

```
trigger HiringManagerTrigger on Hiring_Manager__c (before insert)
{
    if(Trigger.isBefore && Trigger.isInsert)
    {
        for(Hiring_Manager__C hrRecord : Trigger.New)
        {
            Integer recordsCount = [Select count() from Hiring_Manager__C
                Where name =: hrRecord.Name and
                Contact_Number__c =: hrRecord.Contact_Number__c and
Email_Address__c =: hrRecord.Email_Address__c ];
    if(recordsCount > 0)
        hrRecord.AddError('Duplicate Record found. Hence Record cannot be Inserted');
        }
    }
}
```

Use Case: Create a trigger on the Account Object to prevent the deletion of related contact records from the object upon removing the Account Record.

Object Name : Account Object

Event Name : Before Delete

```
trigger PreventContactDeletionTrigger on Account (before delete)
{
    if(Trigger.isBefore && Trigger.isDelete)
    {
        List<Contact> lstContacts = [Select id, firstname, lastname, accountid
            from Contact  Where AccountID IN : Trigger.OldMap.KeySet() ];
    if(! lstContacts.isEmpty())
        {
```

```
                    For (Contact conRecord : lstContacts)
                    {
                        conRecord.AccountId = null;
                    }
                    Update lstContacts;
                }
            }
        }
```

Lead Conversion: Lead is a Person/Organization/Business who just showed interest in our Organization Products/ Services.

Once the Lead Person is ready to buy the products, then we can convert the lead as a customer.

While converting the lead as the customer, Salesforce generates the below 3 Records.

1. Account Record

2. Contact Record

3. Opportunity Record

In Salesforce we have to convert each lead record as the customer manually by using "Convert" button. Bulk lead conversion process is not available in Salesforce.

To convert the bulk lead records as customer we have to use "Database.LeadConvert" class.

Database.LeadConvert, is a readymade class, which can hold a lead record to be get Converted.

To store the multiple lead records we have to create a list Collection as below.

Ex: List<Database.LeadConvert>

Database.LeadConvert class provides the below instance methods.

Ex: Database.LeadConvert lConvert = new Database.LeadConvert();

Methods

1. SetLeadId(<LeadRecordID>): It is used to store the lead Record Id which needs to be converted.

Ex: lConvert.SetLeadId('00Q5j000006r9X3EAI');

2. SetSendNotificationEmail(Boolean): Used to specify whether an email notification regarding the lead Conversion should be sent to the lead owner or not.

TRUE ---> Send email alert to lead owner.

FALSE ---> Won't send an email alert.

Ex: lConvert.SetSendNotificationEmail(true);

3. SetDoNotCreateOpportunity(Boolean): It is used to decide whether an opportunity record should be generated or not during lead conversion.

TRUE ---> Won't Generate Opportunity

FALSE ---> Will generate Opportunity

Ex: lConvert.SetDoNotCreateOpportunity(false)

4. SetConvertedStatus(String): It is used to specify the status to be assigned for the lead record upon conversion.

All the Lead Status options will get stored inside the "Lead Status" object.

Ex: LeadStatus lStatus = [Select id, masterlabel, isConverted from LeadStatus

Where isConverted = true];

lConvert.SetConvertedStatus(lStatus.masterlabel);

Converting Lead: To convert the lead as the customer, use "Database.ConvertLead()" method.

Ex: Database.LeadConvertResult[] results =Database.ConvertLead(lConvert);

Use Case: Create a trigger on the Lead Object to auto convert the lead as the customer upon updating the Lead Status as "Closed - Converted".

Object Name : Lead Object

Event Name : After Update

```
trigger AutoLeadConvertTrigger on Lead (After Update)
    {
        if(Trigger.isAfter && Trigger.isUpdate)
        {
        LeadStatus lStatus = [Select id, MasterLabel, isConverted  from LeadStatus
                                                    Where isConverted = true];

        List<Database.LeadConvert>leadRecordsToConvert=new
List<Database.LeadConvert>();

        for(Lead ldRecord : Trigger.New)
        {
        if(ldrecord.IsConverted == False && ldRecord.Status == 'Closed - Converted')
            {
            // Write the Code to Convert the Lead..
            Database.LeadConvert lConvert = new Database.LeadConvert();
```

```
                        lConvert.setLeadId(ldrecord.Id);
                        lConvert.setDoNotCreateOpportunity(true);
                        lConvert.setSendNotificationEmail(true);
                        lConvert.setConvertedStatus(lStatus.MasterLabel);
                        // Add the record to Collection..
                        leadRecordsToConvert.Add(lConvert);
                                }
                        }
                        // Convert the Lead Records as Customers.
Database.LeadConvertResult[] results =      Database.convertLead(leadRecordsToConvert,
false);
                }
        }
```

Trigger Handler Factory / Logic Less Trigger

Example:

Trigger Code:

```
trigger AutoLeadConvertTrigger on Lead (After Update)
{
        if(Trigger.isAfter && Trigger.isUpdate)
   {
      AutoLeadConvertTriggerHandler.AfterUpdate(Trigger.New);
   }
}
```

Handler Class:

```
public class AutoLeadConvertTriggerHandler
{
         Public static void AfterUpdate(List<Lead> lstLeads)
   {
       LeadStatus lStatus = [Select id, MasterLabel, isConverted from LeadStatus
                                                Where isConverted = true];
      List<Database.LeadConvert> leadRecordsToConvert = new List<Database.LeadConvert>();
      for(Lead ldRecord : lstLeads)
```

```
    {
        if(ldrecord.IsConverted == False && ldRecord.Status == 'Closed - Converted')
        {
            // Write the Code to Convert the Lead
            Database.LeadConvert lConvert = new Database.LeadConvert();


                lConvert.setLeadId(ldrecord.Id);
                lConvert.setDoNotCreateOpportunity(false);
                lConvert.setSendNotificationEmail(true);
                lConvert.setConvertedStatus(lStatus.MasterLabel);
            // Add the record to Collection
            leadRecordsToConvert.Add(lConvert);
        }
    }
    // Convert the Lead Records as Customers
    Database.LeadConvertResult[]results =Database.convertLead(leadRecordsToConvert,
false);
  }
}
```

Trigger Best Practices

1. Trigger should support bulk records as well. Always use "For Loop" to process records inside the trigger.

2. Always check for null pointer exceptions.

3. Always use try and catch block in the trigger to handle exceptions.

4. Differentiate events with separate blocks to avoid usage of trigger.new and trigger.old.

5. Do not use DML operations inside the for loop. Add them to the list and update/insert/delete the list outside the for loop.

6. Do not use SOQL and SOSL statements inside for loop.

7. Write the Proper test Classes for the trigger and maintain minimum 1% of code coverage for each trigger. Overall organization wide maintain 75% of code coverage while moving the code from sandbox to production environment.

8. Avoid recursiveness in triggers by putting the proper conditions.

Note: Recursive Triggers can be avoided by using the "Static Boolean" variables.

Do not execute the trigger on all the update. Only execute the trigger on specific update.

9. As a Best Practice, it is Recommended to maintain only One Trigger per an Object. Because, We don't have the option in salesforce to control the execution order of triggers.

10. It is recommended to maintain Trigger Handler Factory (i.e., instead of writing the Trigger code inside the Trigger, write the code in a Class and call the class from the trigger).

Order of Execution in Triggers in Salesforce

1. The original record is loaded from the database.

2. The new record field values are loaded from the request and overwrite the old values.

3. All before triggers execute.

4. System validation occurs such as verifying that all required fields have a non-null value and running any user-defined validation rules.

5. The record is saved to the database, but not yet committed.

6. All after triggers execute.

7. Assignment rules execute.

8. Auto-response rules execute.

9. Workflow rules execute.

10. If there are workflow field updates, the record is updated again.

11. If the record was updated with workflow field updates, before and after triggers fire one more time (and only one more time) along with Standard Validation Rules. (Note: In this case, Custom validation rules will not fire).

12. Escalation rules execute.

13. All DML operations are committed to the database.

14. Post-commit logic executes such as sending email.