TRIGGERS IN SALESFORCE

Apex triggers enable you to perform custom actions before or after changes to Salesforce records, such as insertions, updates, or deletions.

TYPES OF TRIGGERS

BEFORE TRIGGERS

USED TO UPDATE OR VALIDATE RECORDS BEFORE THEY ARE SAVED INTO THE DATABASE.

AFTER TRIGGERS

AFTER TRIGGERS ARE USED TO ACCESS FIELD VALUES THAT ARE SET IN THE SYSTEM.

TO AFFECT CHANGES IN OTHER RECORDS.

When to use before triggers and after triggers?

BEFORE TRIGGER

Before Triggers are executed in Salesforce prior to the records being inserted/updated getting committed to the Database.This means that you can make changes to the record being updated without issuing an additional DML statement(insert, update etc)

USE BEFORE TRIGGERS, IF YOU WANT TO MAKE CHANGES IN THE RECORD BEING UPDATED.

LIMITATIONS:

- You can't access the Id field of new records in Before Triggers
- Formula fields are not calculated in Before Triggers
- Roll-up summaries are not available

AFTER TRIGGERS

After Triggers are executed after records are committed to the Database. This means that you can't make more changes to the records that were updated without issuing an additional DML statement.

USE AFTER TRIGGER TO MAKE CHANGES TO A RELATED OBJECT

LIMITATIONS::

- You cannot make additional field updates to a record in an After Trigger without issuing an additional DML statement

TRIGGER EVENTS

Before

INSERT
UPDATE
DELETE

AFTER

INSERT

UPDATE

DELETE

UNDELETE

Trigger syntax

```
trigger TriggerName on ObjectName (trigger_events) {
    code_block;
}
```

ORDER OF EXECUTION OF TRIGGERS

1. System Validation Rule
2. Apex Before Triggers
3. Custom Validation rules
4. Duplicate rules (After the 4th step the RECORD is temporarily saved (not committed)into the database, so further at any point roll back can be done.At this point a RECORD ID is ALSO GENERATED)
5. Apex After triggers
6. Assignment Rules
7. Auto-Response Rules
8. Workflow Rules
9. Process Builder and Flows
10. Escalation Rules
11. Roll-up summary fields(COMMIT TRANSACTION TAKES PLACE AFTER 11TH Step ,roll back cannot happen after this step)

Trigger context variables ::All triggers define implicit variables that allow developers to access run-time context. These variables are contained in the **System.Trigger** class.

Trigger.isBefore - Boolean
Trigger.isAfter - Boolean
Trigger.isInsert - Boolean
Trigger.isUpdate - Boolean
Trigger.isDelete - Boolean
Trigger.isUndelete - Boolean
Trigger.size -Integer
Trigger.isExecuting - Boolean

Trigger.new -List
Trigger.old - List
Trigger.newMap - Map
Trigger.oldMap-Map
Trigger.operationType-Enum

EVENT	BEFORE	AFTER
INSERT	Trigger.new	Trigger.new Trigger.newMap
UPDATE	Trigger.new Trigger.newMap Trigger.old Trigger.oldMap	Trigger.new Trigger.newMap Trigger.old Trigger.oldMap
DELETE	Trigger.old Trigger.oldMap	Trigger.old Trigger.oldMap
UNDELETE		Trigger.new Trigger.newMap

Trigger.new :: Used to get a list of all the records after trigger activation.
Trigger.new cannot be changed in after dml operations since it has already been stored; however, you can change it in a before dml operation.
If you try to change trigger.new in an after dml operation, you'll get an error like this: execution of afterupdate caused by: system.finalexception: record is read-only ()

Trigger.old :: Prior to the update process, this method returns a list of the old records. Only update and delete triggers have this feature. There are no old values in insert triggers.

Trigger.oldMap :: Prior to the update operation, it returns a Map from the record Id to the related record, e.g. Map<Id, Account>. Only update and remove triggers have this option. There are no old values in insert triggers.

Trigger.newMAP:: Returns a map from a record Id to the record it belongs to. For instance, Map<Id, Account>. This map will only be viewable once the record Ids have been filled in. As a result, the Map in the Before Insert Trigger cannot be accessed. After insert, before the update, and after undelete triggers all support the newMap Context variable.

Trigger.operationType::

Returns an enum of type System.TriggerOperation corresponding to the current operation. Possible values of the System.TriggerOperation enum are: BEFORE_INSERT, BEFORE_UPDATE, BEFORE_DELETE,AFTER_INSERT, AFTER_UPDATE, AFTER_DELETE, and AFTER_UNDELETE. If you vary your programming logic based on different trigger types, consider using the switch statement with different permutations of unique trigger execution enum states.

Considerations while using trigger context variables

- trigger.new and trigger.old cannot be used in Apex DML operations.
- You can use an object to change its own field values using trigger.new, but only in before triggers. In all after triggers, trigger.new is not saved, so a runtime exception is thrown.
- trigger.old is always read-only.
- You cannot delete trigger.new.

SOME TRIGGER RELATED SCENARIOS

- 1. **Write a trigger, when a new Account is created then create a contact related to that account.**

```
trigger NewAcc on Account (after insert) {
        if(Trigger.isAfter && Trigger.isInsert){
        List<Contact> acc= new List<Contact>()
        for(Account a: trigger.new)
        {
        Contact c=new Contact();
        c.id=a.Id;
     c.LastName='Test contact';
        acc.add(c);
        }
        insert acc;
        }
}
```

IN THIS SCENARIO AFTER TRIGGER IS USED BECAUSE WE HAVE TO MAKE CHANCES TO A OBJECT RELATED TO ACCOUNT OBJECT.TRIGGER.NEW VARIABLE IS USED TO FETCH THE NEWLY INSERTED RECORDS IDS AND DML OPERATION IS USED TO INSERT THE NEW RECORDS OF THE RELATED OBJECT.

· 2 Create a trigger to prevent users from creating Duplicate Account Records with the same name.

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

for(Account a: Trigger.new){
   LIST<Account> acclist=[Select Name FROM Account where Name=:a.Name];
   if(acclist.size()>0){
   a.AddError('Duplicate Account Name exists');
                }
           }

           }
```

3.Create field 'Count of Contacts" on ACCOUNT OBJECT

When we add the contacts for that account,then count should populate on the Account details page.

When we delete the Contacts for that Account,then count will update automatically.

```
public class ContactHandlerr {

    public static void noOfContacts(){

        set<Id> Accountswithcontacts = new set<ID>();

        for(Contact c:(List<Contact>) Trigger.new){

            if(c.AccountID!=null)
            {
                Accountswithcontacts.add(c.AccountID);
            }
        }
        list<Account> acctobeupdated=new list<Account>();
        list<Account>  acc=[SELECT ID,NoOfCONTACTS__c,(SELECT ID FROM Contacts) FROM
ACCOUNT WHERE ID IN:Accountswithcontacts ];
        for(account a : acc){
            a.noOfContacts__c=a.contacts.size();
            acctobeupdated.add(a);
        }
        update acctobeupdated;
    }
```

```
//Trigger
trigger contactTrigger on Contact (after insert,after update) {
    switch on Trigger.OperationType{
        when after_INSERT{
            ContactHandlerr.noOfContacts();
        }when after_UPDATE{
            ContactHandlerr.noOfContacts();
        }
          }
}
```

4. When a user creates an account record without rating then prepopulates the rating as cold.

```
trigger AccountTrigger on Account (before insert) {
    list<Account> accRating=new list<Account>();
    for(Account a : Trigger.new){
        if(a.Rating==Null){
            a.Rating='cold';
            accRating.add(a);
        }
    }try{
        insert accRating;
    }
    catch(Exception e){
        System.debug(e.getMessage());
    }}
```

Write a trigger, to achieve the following:Create a new Opportunity whenever an account is created/updated for Industry – Agriculture.Opportunity should be set as below:Stage = 'Prospecting', Amount = $0, CloseDate = '90 days from today'.

```
trigger NewOpp on Account (after insert,after update) {
    List<Account> acclist=new List<Account>();
    acclist=[Select Id,Name,Industry FROM Account where Industry='Agriculture'];
    List<Opportunity> opplist=new List<Opportunity>();
    if(Trigger.isInsert && Trigger.isAfter){
        for (Account a:accList){
                Opportunity newOppo=new Opportunity();
                newOppo.Name= a.Name +'OPPORTUNITY ';
                newOppo.AccountId=a.id;
                newOppo.StageName='Prospecting';
                newOppo.Amount=0;
            newOppo.CloseDate=System.today()+90;
                opplist.add(newOppo);
```

```
if(oppList.isEmpty() == false) {
            insert oppList;}}
    if(Trigger.isUpdate && Trigger.isAfter)
    {     List<Opportunity> listOfOpportunities = new List<Opportunity>();
       for (Account a:accList){
            if(a.Industry!=Trigger.oldMap.get(a.Id).Industry && a.Industry=='Agriculture' )
            {
            Opportunity newOppo=new Opportunity();
            newOppo.Name= a.Name +'OPPORTUNITY ';
            newOppo.AccountId=a.id;
            newOppo.StageName='Prospecting';
            newOppo.Amount=0;
           newOppo.CloseDate=System.today()+90;
            opplist.add(newOppo);
            }}
```

TRIGGER BEST PRACTICES

1) One Trigger Per Object

A single Apex Trigger is all you need for one particular object. If you develop multiple Triggers for a single object, you have no way of controlling the order of execution if those Triggers can run in the same contexts

2) Logic-less Triggers

If you write methods in your Triggers, those can't be exposed for test purposes. You also can't expose logic to be re-used anywhere else in your org.

3) Context-Specific Handler Methods

Create context-specific handler methods in Trigger handlers

4) Bulkify your Code

Bulkifying Apex code refers to the concept of making sure the code properly handles more than one record at a time.

5) Avoid SOQL Queries or DML statements inside FOR Loops

An individual Apex request gets a maximum of 100 SOQL queries before exceeding that governor limit. So if this trigger is invoked by a batch of more than 100 Account records, the governor limit will throw a runtime exception

6) Using Collections, Streamlining Queries, and Efficient For Loops

It is important to use Apex Collections to efficiently query data and store the data in memory. A combination of using collections and streamlining SOQL queries can substantially help writing efficient Apex code and avoid governor limits

7) Querying Large Data Sets

The total number of records that can be returned by SOQL queries in a request is 50,000. If returning a large set of queries causes you to exceed your heap limit, then a SOQL query for loop must be used instead. It can process multiple batches of records through the use of internal calls to query and queryMore

8) Use @future Appropriately

It is critical to write your Apex code to efficiently handle bulk or many records at a time. This is also true for asynchronous Apex methods (those annotated with the @future keyword). The differences between synchronous and asynchronous Apex can be found

9) Avoid Hardcoding IDs

When deploying Apex code between sandbox and production environments, or installing Force.com AppExchange packages, it is essential to avoid hardcoding IDs in the Apex code. By doing so, if the record IDs change between environments, the logic can dynamically identify the proper data to operate against and not fail.