A Peek at Apex Triggers

March 3, 2014 Celin Joseph 1

A trigger is simply an Apex code that is executed before or after any specific DML events. They are usually stored as metadata in the application under the associated object.

Triggers are usually used when we want to reflect the actions performed on a particular object on itself or any other object. They can also be used to perform the action of rollup summary, which is generally restricted to 10 per object. Using triggers, we can even call apex class to perform actions.

Defining Triggers:

To define a trigger on:

- Standard object – under Setup, click Customize, select the name of the object and finally click Triggers.
- Custom object – under Setup, click Create | Objects and select the name of the object on which the trigger is to be written.

Trigger Syntax:

trigger triggerName on ObjectName (trigger_events) {

code_block}

where the trigger_events could be

- before insert
- after insert
- before update
- after update
- before delete
- after delete or
- after undelete

Trigger events can also be used in combination by separating each one of them by a comma.

Classification:

Apex triggers can be classified into two types:

1**. Before triggers** – which are used to validate or update record values before saving it to the database.

2. **After triggers** – which are specifically used to access field values that are already set by the database (eg: LastModifiedDate field) and to modify other records accordingly, such as by firing asynchronous events using a queue

Example for Before Insert:

trigger NameTrigger on Contact (before insert) {

for(Contact c : Trigger.New)

c.Name__c = ‘Jill’;

}

If you just change the event to after insert in the example given above, you will end up having an error as the record has been locked. To resolve this error and avoid the record locking issue, you will have to allocate memory and then update the same record explicitly.

Example for After Insert:

```
trigger NameTrigger on Contact (after insert) {

List<Contact> contactList = new List<Contact>();

for(Contact c : Trigger.New){

Contact con = new Contact(id = c.id, Name__c = 'Jill');

contactList.add(con);

}

if(contactList.size()>0)

update contactList;

}
```

To update the same record we should use before trigger but to work with system generated fields we need the aid of after trigger.

By default all triggers are **bulk triggers;** therefore it can process multiple records at a time. We should always be prepared to process more than one record at a time. It can handle single record updates as well as bulk operations like Mass actions or Data import

Note: Any recurring event object cannot be processed in bulk for insert, update or delete triggers.

<u>Implementation Consideration:</u>

- Upsert triggers fire
    - both before and after insert in case a new record is to be inserted or
    - both before and after update triggers in case an existing record is updated according to the DML operation.
- Merge triggers fire
    - both before and after delete triggers for the records that are being deleted as a result of merge statement and
    - before update triggers only for those records that are retained after the merge statement.

Consider: if two accounts are merged, only the delete and update account triggers fire. No triggers for records related to the accounts such as opportunities or contacts will be fired.

- Field history cannot be recorded until the end of a trigger.
- Triggers that are executed after a record is undeleted, will only work with specific objects like account, lead, contact, etc.

<u>Trigger Context Variables:</u>

To access runtime context all triggers define implicit variables that belongs to the class System.Trigger.

Some of the trigger variables are:

- isBefore
- isAfter
- isInsert

- isDelete
- isUpdate
- isUndelete
- isExecuting
- new
- old
- newMap
- oldMap
- size

<u>Context Variable Considerations:</u>

There are certain considerations while writing a trigger.

- trigger.new and trigger.old cannot be used in Apex DML operations.
- trigger.new is used in before insert and before update trigger events for modifying the field values in a record.. In all after triggers, trigger.new is not saved and so it throws a runtime exception.
- trigger.old is always read-only.
- trigger.new cannot be deleted

The following table gives a list of considerations about certain actions in different trigger events:

Trigger Event	**Can change fields using trigger.new**	**Can update original object using an update DML operation**	**Can delete original object using a delete DML operation**
before insert	Allowed	Not applicable.	Not applicable.
before update	Allowed	Not allowed. A runtime error is thrown.	Not allowed. A runtime error is thrown.
before delete	Not allowed. A runtime error is thrown.	Allowed.	Not allowed. A runtime error is thrown.
after insert	Not allowed.	Allowed	Allowed, but unnecessary.
after update	Not allowed.	Allowed.	Allowed.
after delete	Not allowed.	Not applicable. The object has already been deleted.	Not applicable. The object has already been deleted.
after undelete	Not allowed.	Allowed	Allowed, but unnecessary.

- The after undelete trigger event only works with recovered records or also called undeleted records. Also, this event only fires for certain objects.
- Triggers usually follow an order of execution but may change under certain circumstances like when there are multiple triggers being executed on the same object.
- Certain bulk operations do not invoke triggers like mass address updates, mass email actions.
- There are certain entity and field considerations to be observed before writing triggers. Certain fields or entities may not be updateable in before or after triggers. Example:
  - Fields like Task.isClosed cannot be updated in before triggers and Task.WhoId cannot be updated in after triggers.
  - Entities like FeedItem cannot be supported in update and after undelete triggers.
- DML operations can be prevented from occurring by triggers by calling the addError() method on a record or field. When used on Trigger.new records in case of insert and update triggers and on Trigger.old records in case of delete triggers, an error message is displayed in the application interface and logged.

Conclusion

Therefore, keeping these considerations in mind we can create tailored apex triggers to handle logic that cannot be achieved even by using workflow rules. Thereby we can make our application more customized and more responsive according to our requirement.