Salesforce Batch Apex | Part 2

Apex Development

Execute the Batch Apex Job Programmatically

In the last **episode**, we discussed batch apex three methods Start(), Execute() and Finish() as well as it's implementation.

In this episode, we will discuss invoking and executing a batch job programmatically. So let's get started...

In apex, we can use the **Database.executeBatch()** method to programmatically begin the batch job in salesforce.

Database.**executeBatch** method takes two parameters: An instance of a class that implements the **Database**. Batchable interface. An optional parameter scope. This parameter specifies the number of records to pass into the execute method.
Let's have a look at the syntax of it then we will discuss how to use it.

Syntax :

```
==============================================
public static ID executeBatch (sObject className)

public static ID executeBatch (sObject className, Integer Scope)
==============================================
```

In the above two methods are static methods of Database class. We can use any one of the methods to execute the batch job.

NOTE : The class name what we are passing to the **Database.executeBatch()**method should be an object of the class which has implemented**Database.executeBatch()** method should be an object of the class which has implemented **Database.Batchable** interface.

Order Of Execution of Batch Apex Job :

Below is the order of execution of batch apex job when we invoke through**Database.executeBatch()**.

STEP 1 :
Create an object for the class which has implemented **Database.Batchable**interface.

STEP 2 :
Pass this object which you have created in the first step as a parameter to the**Database.Batchable** interface.

STEP 3 :
When **database.executeBatch()** method is called it will add the batch job to the queue.

STEP 4 :
Once the resource is available in the queue automatically **Start()** method will be invoked and it will collect all the records on which we need t perform the operation.

STEP 5 :
The records that are fetched from the **Start()** method are divided into small group/batch (chunks) of size given in the **Database.execute()** method.

STEP 6 :
On every batch of records execute method will be invoked.

STEP 7 :
Once all the batches are executed, then the finish method will be called.

NOTE :
1) **Database.execute** method is going to return Id of the batch job. Using which we can monitor or abort the operation.

2) All the asynchronous jobs are stored to '**AsyncApexJob**' object from this we can monitor, no. of jobs processed for our asynchronous job and status of the job.

EXAMPLE :

```
==========================================
ID batchprocessid = Database.executeBatch(reassign);

AsyncApexJob aa = [SELECT Id, Status, JobItemsProccessed]
                          FROM AsyncApexJob WHERE Id =: batchprocessid];
```

==

Let's understand more clearly with the help of a real-time scenario

Scenario :

1. Create a vidualforce page with Input text button.
2. When we click on the Replace button in the custom visualforce page.
3. Fetch All the records in the customer object with the customer name matching with a name given in the visulforce page.
4. Update their Account Type as 'Saving' account.

SOLUTION :

STEP 1 :
Create a batchapex class to perform an update of the Account Type.
==

```
global class CustomerBatch implements Database.Batchable <sObject>
{
 public string myname;

 global customerBatch(String myname)
 {
  this.myname = myname;
 }
global Database.QueryLocator Start(Database.BatchableContext BC)
 {
  return Database.getQueryLocator('SELECT id, Account_Type__c FROM
Account WHERE Name='+ myname);
 }
 global void execute(Database.BatchableContext BC, List<Customer__c>
Scope)
```

```
 {
  List<Customer__c> cust = new List<Customer__c>();
  for(Customer__c c : cust)
 {
  c.Account_Type__c = 'Saving';
  cust.add(c);
 }
update cust;
 }
global void finish (Database.BatchableContext BC)
 {
//Method to send and email
  Messaging.singleEmailMessage myemail = new
Messaging.singleEmailMessage();

String[] toadd = new String[] {'abc@gmail.com'};
myemail.setToAddress(toadd);
myemail.setSubject('Batchprocessed');
myemail.setplainTextBody('Batch completed successfully');
Messaging.sendEmail(new Messaging.Email(){myemail});
 }
}
```

==

STEP 2 :

Create an Apex Class to invoke the batch apex job.

==

```
public class TestMyBatch
{
 public string cname{get; set;}
 public pageReference show()
 {
  customerBatch mybatch = new customerBatch (cname);

Id id = Database.executeBatch (mybatch, 400);
system.debug ('My Job id' + id);
 }
}
==============================================
```

STEP 3 :
Create a visualforce page to call the TestMyBatch class

```
==============================================
<apex:page controller="TestMyBatch">
 <apex:form>
  <apex:outputLabel> Enter Name </apex:outputLable>
  <apex:inputText value="{!myname}"/>
  <apex:commandButton value="click" action="{!show}"/>
 </apex:form>
</apex:page>
==============================================
```

Cool !!! In this way, you can achieve the solution for the above scenario.
now let's understand the next important concept.

Database.Stateful :

When using **Database**.**Stateful**, only instance member variables retain their values between transactions. Static member variables don't retain their values and are reset between transactions.

Each execution of batch apex job is considered as a discrete transaction. Which means when you have a batch job with 1000 records executed with an optional scope of 200 records then

5 batch
 Batch -> 1-200
 Batch -> 201-400
 Batch -> 401-600

When we call execute on batch, to summarize the value

```
Integer sum = 0;
public void execute(Database.BatchableContext bc, List<Account> scope)
{
 for(Account a : scope)
 {
  for(Account a : scope)
 sum = sum+a.AnnualRevenue;
 }
}
```

- The first batch of 1-200 records call the execute method: with 200 records
 Before calling **execute() : sum = 0**
 After calling **execute() : sum = 30000 (assume it)**

- When the execute() is called on batch 2 of records 201-400 then Initial value of sum again set to zero.

Before calling **execute() on Batch1 : 1-200 records : sum=0;**
After calling **execute() on Batch1 : 1-200 records : sum=3000;**
Before calling **execute() on Batch2 : 201-400 records : sum=3000;**

- When we call execute() on **Batch2 : 201-400** records.
    First Sum is set=0 again: sum=0.

Then again fresh summary value is calculated again.

- Which means the state of the batch is forwarded from one **execute()** to another**execute().**

- If you specify Database.stateful in the class definition, you can maintain State across these transactions. This is useful for counting or summarizing records as they are processed.

For Example :
Suppose your job processed opportunity records. You could define a method in execute to aggregate totals of the opportunity amounts as they were processed.
==

```
global class SummarizeAccountTotal implements
Database.Batchable<sObject>, Database.Stateful
{
 global final string query;
 global integer summary;

 global summarizeAccountTotal(string q)
 {
  Query = q ;
  summary = 0 ;
 }

global Database.QueryLocator Start(Database.BatchableContext BC)
 {
  return Database.getQueryLocator(query);
 }

global void execute (Database.BatchableContext BC, List<sObject> scope)
 {
```

```
 for(sObject s : scope)
 {
 summary = Integer.valueOf(s.get('total__c'))+ Summary;
 }
 }

global void finish(Database.BatchableContext BC)
 {
 }
}
```
==

Governor Limits :

1) Only one batch Apex job's start method can run at a time in an organization.

2) Up to 5 queued or active batch jobs are allowed for the apex.

3) The maximum number of batch apex method executions per 24-hours period is 2,50,000.

4) The batch ApexStart method can have up to 15 query cursors open at a time per users.

5) A maximum of 50 million records can be returned in the **Database.QueryLoactor**object.

6) The Start(), Execute() and Finish() methods can implement up to 10 callouts each.

NOTE :
If we have 1000 records with a scope of 200 records then they are divided into 5 batches.

Hence execute() method is called 5 times. Which means in every execute() we call 10 callouts. So in this scenario, we called

Start() -> 10 callouts
Executes() -> 10*5=50 callouts
Finish() -> 10 callouts

Total -> 70 callouts in entire operations

Limitations :

- Methods declared as the future aren't allowed in the classes that implement Database.Batchable interface.

- Methods declared as future can't be called from Batch Apex class.

- For Sharing recalculation, we recommend that the execute method to delete and then re-create all Apex managed sharing for the records in the batch.

- For every 10,000 AsyncApexJob records, Apex creates one additional AsyncApexJob record of type BatchApexWorker for internal use.