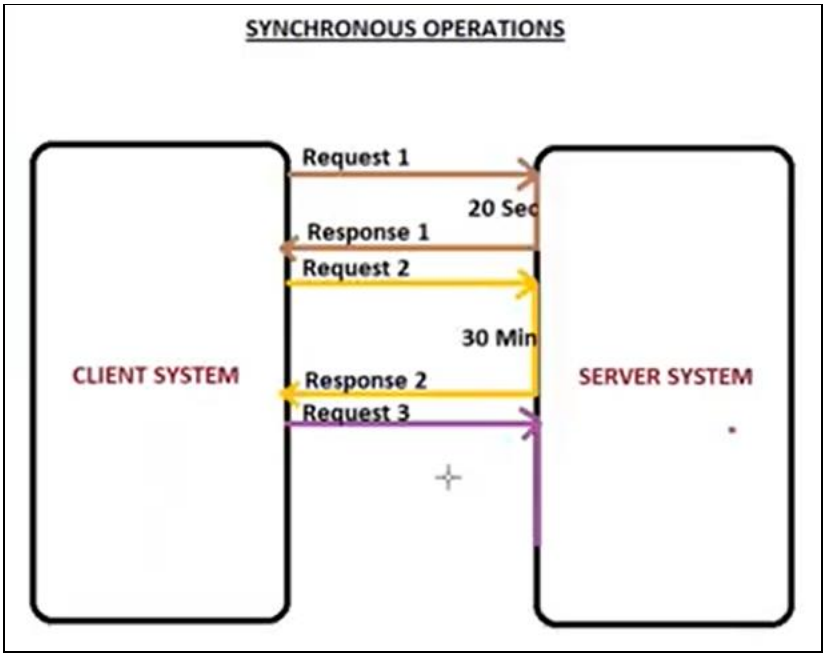
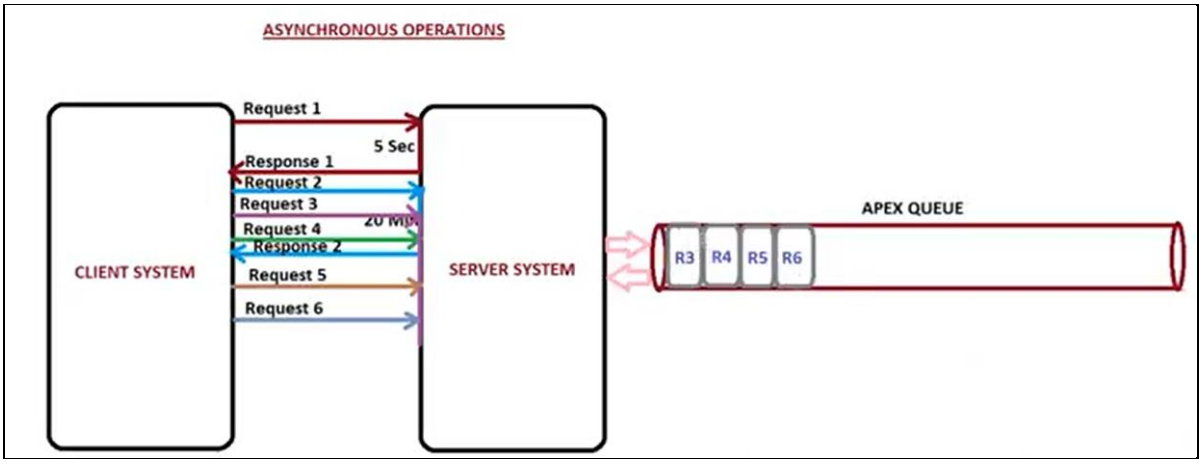
ASYNCHRONOUS PROGRAMMING

An asynchronous process is a process or function that executes a task "in the background" without the user having to wait for the task to finish.

Apex offers multiple ways for running your Apex code asynchronously. We can choose the asynchronous apex feature that best suits your needs.

Asynchronous Process will be used while performing long running operations which involves the processing millions of records and complex transactions.





Asynchronous operations in Salesforce can be implemented by using

1. Batch Programming / Batch Apex

2. Schedule Programming / Schedule Apex

3. Future Methods

4. Queueable Apex / Queueable Interface

5. Flex Queues

Batch Programming / Batch Apex: Batch Programming is used to perform the operations on the bulk records by dividing them to the various smaller chunks called as "Batches".

Batch Jobs will always be run outside the organization, by placing inside the "Apex Queue".

By using Batch jobs, we can process maximum of 50 million records. It will divide the records to the various batches of size 1 - 2000.

i.e., Minimum Batch Size: 1

Maximum Batch Size: 2000

Default Batch Size: 200

Each batch will be running under a discrete transaction. Hence one batch results will not affect the results of the other batch.

Salesforce maintains the Partial Processing between the batches.

All the batch job results will get reside inside the "AsynApexJob" object.

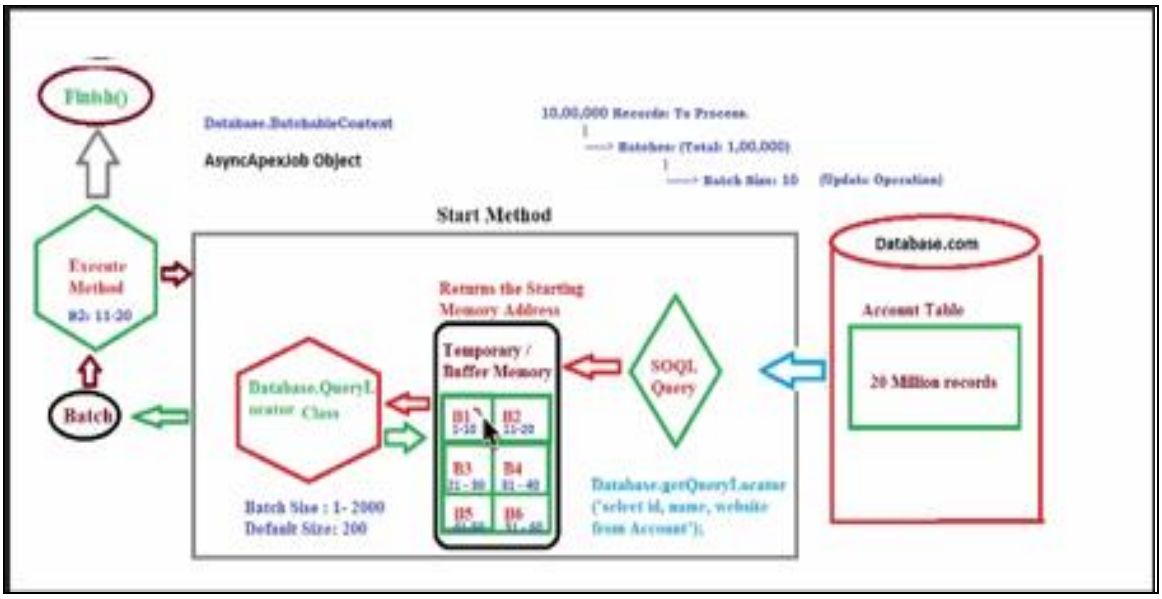
We can process maximum of 2,50,000 batches in 24-hour time frame.

To implement the batch Programming, we use an Interface called as "Database.Batchable" interface which contains the below 3 methods.

1. Start()

2. Execute()

3. Finish()



Implementation Steps

Step 1: Define a Global class, which implements the Interface

"Database.Batchable<SObject>"

```
Global Class <BatchClassName> implements Database.Batchable<SObject>
        {
        // Write the Business Logic
        }
```

Step 2: Provide the implementation for the Interface Methods

```
Global Class <BatchClassName> implements Database.Batchable<SObject>
        {
Global Database.QueryLocator Start (Database.BatchableContext <refName>)
        {
                // Write the Business Logic
        }
Global void Execute (Database.BatchableContext <refName>, List<SObject> records)
        {
                // Write the Business Logic
        }
```

```
Global void Finish (Database.BatchableContext <refName>)
        {
                // Write the Business Logic
        }
        }
```

Step 3: Invoke the Batch Class by using Database.ExecuteBatch() method

i. Create the Object of the Batch Class

<BatchClassName> <objectName> = new <BatchClassName>();

ii. Invoke the Batch Class

ID jobId = Database.ExecuteBatch(<objectName>); // Default Size:200

(OR)

ID jobId = Database.ExecuteBatch(<objectName>, Integer batchSize);

Step 4: Track the Status of the Batch Job

i. Wizard Format:

Click on "Setup" menu.

1. Search for the option "Apex Jobs".
2. Verify the Batch Job Status Details.

ii. Programmatically by using "AsynApexJob" object.

AsynApexJob jobInfo = [Select id,Status, totalJobItems, jobItemsProcessed, numberOfErrors, CreatedBy.Email from AsynApexJob Where Id =: <jobId>];

Use Case: Create a Batch Job to update All the Account Records by assigning the below Values for the fields by dividing the records to various batches of Size "5".

1. Account : Active == 'Yes'
2. Account : Type == "Installation Partner"
3. Account : Ownership = 'Public'
4. Account : AnnualRevenue == 4500000

Class Code

```
Global class UpdateAccountRecordsBatch implements Database.Batchable<SObject>
{
        Global Database.QueryLocator Start(Database.BatchableContext bContext)
    {
```

```
    String accountsQuery = 'Select id, name, rating, industry, annualrevenue, type, ownership,
active__c '+ ' from Account';
    return Database.getQueryLocator(accountsQuery);
  }
  Global void Execute (Database.BatchableContext bContext, List<SObject> lstRecords)
      {
          if(! lstRecords.isEmpty())
              {
          List<Account> accountsToUpdate = new List<Account>();
         for (SObject obj: lstRecords)
              {
        // Convert the SObject Type into Account Type
        Account accRecord = (Account) obj;
        // Assign the New Values for the Record
            accRecord.Active__c = 'Yes';
            accRecord.Ownership = 'Public';
            accRecord.Type = 'Installation Partner';
            accRecord.AnnualRevenue = 4500000;
            // Add the Record to Collection
            accountsToUpdate.Add(accRecord);
         }
      // Update the Records to Database
      if (! accountsToUpdate.isEmpty())
        Update accountsToUpdate;
    }
  }
  Global void Finish (Database.BatchableContext bContext)
  {
    System.debug('Batch Job Id is: '+ bContext.getJobId());
   AsyncApexJob jobDetails = [Select id, Status, totalJobItems, jobItemsProcessed,
                                          numberOfErrors, CreatedBy.Email
                              from AsyncApexJob Where id =: bContext.getJobId()];
```

```
        // Send the Results to the Users through Email Alert
        Messaging.SingleEmailMessage sEmail = new Messaging.SingleEmailMessage();
       String[] toAddress = new String[]{jobDetails.CreatedBy.Email, 'srinivas@gmail.com'};
            sEmail.setToAddresses(toAddress);
            sEmail.setSenderDisplayName('DELL Customer Support Center.');
            sEmail.setReplyTo('support@dell.com');
            String emailSubject = 'Customer Details Update Batch Job Status Notification - UpdateAccountRecordsBatch ( '+ bContext.getJobId()+ ' )';
            sEmail.setSubject(emailSubject);
            String emailContent = 'Dear Customer Support Team, <br/><br/> '+
                                  'We are pleased to Inform you, that Weekly Customer Details Update Batch Job has been Processed Successfully. <br/><br/>'+
                                  'Here are the Batch Job Results...: <br/><br/>'+
                                  'Batch Job Id is.....: '+ bContext.getJobId()+
                                  '<br/>Batch Job Name is.....: UpdateAccountRecordsBatch'+
                                  '<br/>Batch Job Status is......: '+ jobDetails.Status+
                                  '<br/>Total Number Of Batches Available...: '+ jobDetails.TotalJobItems+
                                  '<br/>Number of Batches Processed....: '+ jobDetails.JobItemsProcessed+
                                  '<br/>Number of Batches Failed......: '+ jobDetails.NumberOfErrors+
                                  '<br/>Batch Job Owner Email Address.....: '+ jobDetails.CreatedBy.Email+
                                  '<br/><br/> Please Contact on the below address, if any queries. <br/><br/>'+
                                  '***<i>This is a System-Generated Email. Please Do Not Reply.</i><br/><br/>'+
                                  'Thanks & Regards, <br/>Customer Support Center, <br/>DELL Inc.';
            sEmail.setHtmlBody(emailContent);
            Messaging.sendEmail(new Messaging.SingleEmailMessage[]{sEmail});
    }
}
```

Execution

```
// Create the Object of the Batch Class
UpdateAccountRecordsBatch accountsBatch = new UpdateAccountRecordsBatch();
// Invoke the Batch Class
```

Id jobId = Database.executeBatch(accountsBatch, 5);

State Management

Batch programming runs stateless by default.

Each batch runs under a discrete transaction. One batch result will not be carried forward to further batches.

To make the program run in stateful mechanism, use the interface Database. Stateful.

Use Case: Configure a Batch Job to calculate the sum of all the Account Record Annual Revenue values by dividing the records to various batches of size "4".

Class Code

```
Global class CalculateTotalRevenueBatch implements Database.Batchable<SObject>, Database.Stateful
{
   Global Decimal totalAnnualRevenue = 0.0;
Global Database.QueryLocator Start (Database.BatchableContext bContext)
   {
      String accountsQuery = 'Select id, name, rating, industry, annualrevenue from Account Where annualRevenue != Null';
      return Database.getQueryLocator(accountsQuery);
   }
   Global void Execute (Database.BatchableContext bContext, List<SObject> accountRecords)
   {
      If (! accountRecords.isEmpty())
      {
         For (SObject obj : accountRecords)
         {
            Account accRecord = (Account) obj;
            totalAnnualRevenue += accRecord.AnnualRevenue;
         }
      }
   }
   Global void Finish (Database.BatchableContext bContext)
   {
      System.debug('Batch Job Id is: '+ bContext.getJobId());
```

```
        AsyncApexJob jobDetails = [ Select id, status, totalJobItems, jobItemsProcessed,
                                    numberOFErrors, CreatedBy.Email from AsyncApexJob
                                                Where id =: bContext.getJobId()];
        MessagingHelperUtility.SendBatchJobStatusNotificationEmail(jobDetails,
'CalculateTotalRevenueBatch', totalAnnualRevenue);
    }
}
```

MessagingHelperUtility Class:

```
public class MessagingHelperUtility
{
        Public static void SendBatchJobStatusNotificationEmail(AsyncApexJob jobDetails,
String jobName, Decimal totalRevenue)
  {
      if(jobDetails.Id != Null)
      {
          Messaging.SingleEmailMessage sEmail = new Messaging.SingleEmailMessage();
          String[] toAddress = new String[]{jobDetails.CreatedBy.Email, 'gopal @gmail.com'};
          sEmail.setToAddresses(toAddress);
         sEmail.setSenderDisplayName('DELL Customer Support Center.');
         sEmail.setReplyTo('support@dell.com');
         String emailSubject = 'Weekly Customers Total Annual Revenue Job Status : '+jobName+
' ('+ jobDetails.Id+ ').';
       sEmail.setSubject(emailSubject);
       String emailContent = 'Dear Customer Support Team, <br/><br/> '+
            'We are pleased to Inform you, that Weekly Customers Total Annual Revenue Batch
Job has been Processed Successfully. <br/><br/>'+
            'Here are the Batch Job Results...: <br/><br/>'+
            'Batch Job Id is.....: '+ jobDetails.Id+
            '<br/>Batch Job Name is.....: '+jobName +
            '<br/>Batch Job Status is......: '+ jobDetails.Status+
            '<br/>Total Number Of Batches Available...: '+ jobDetails.TotalJobItems+
            '<br/>Number of Batches Processed......: '+ jobDetails.JobItemsProcessed+
```

```
            '<br/>Number of Batches Failed.....: '+ jobDetails.NumberOfErrors+
            '<br/>Batch Job Owner Email Address.....: '+ jobDetails.CreatedBy.Email+
            '<br/>Total Annual Revenue Value is.....: '+ totalRevenue +
                '<br/><br/> Please Contact on the below address, if any queries. <br/><br/>'+
                '***<i>This is a System-Generated Email. Please Do Not Reply.</i><br/><br/>'+
                'Thanks & Regards, <br/>Customer Support Center, <br/>DELL Inc.';
                sEmail.setHtmlBody(emailContent);
                Messaging.SendEmailResult[] results = Messaging.sendEmail(new
Messaging.SingleEmailMessage[] {sEmail});
      }
   }
}
```

Execution

```
CalculateTotalRevenueBatch revenueBatch = new CalculateTotalRevenueBatch();
Database.executeBatch(revenueBatch, 4);
```

Schedule Programming: By using this feature, we can Schedule an Apex Class to run at periodical intervals.

Salesforce provides "System.Schedulable" interface to schedule based on the required frequency.

Implementation Steps:

Step 1: Create a Global Class, which implements with the interface "System.Schedulable".

```
Global Class <ScheduleClassName> implements System.Schedulable
    {
        // Write the Business Logic
    }
```

Step 2: Provide the Implementation for the Interface Method "Execute ()".

Ex:

```
Global Class <ScheduleClassName> implements System.Schedulable
    {
Global void Execute (System.SchedulableContext <refName>)
    {
        // Write the Business Logic to invoke the Apex Class
```

```
    }
    }
```

Step 3: Schedule the Global Class, to run at Periodical Intervals.

Click on "Setup" menu.

1. Search for the option "Apex Classes" from Quick Find Box.
2. Click on "Schedule Apex" button.
3. Enter the Schedule Job Name in the Textbox.
4. Select the "Apex Class", to be get Scheduled by using "Lookup icon". (i.e., Global Class)
5. Select the Frequency of Interval.
6. Select the Start Date and End Date.
7. Select the Preferred Start Time from the Picklist.
8. Click on "Save" button.

Step 4: Monitor Schedule Job Status.

We can monitor the Schedule Job Status as below.

Click on "Setup" menu.

1. Search for the option "Scheduled Jobs" in Quick Find box.
2. Monitor the status of the job in the table.

Once an Apex Class has been Scheduled, then we cannot edit the Class Code.

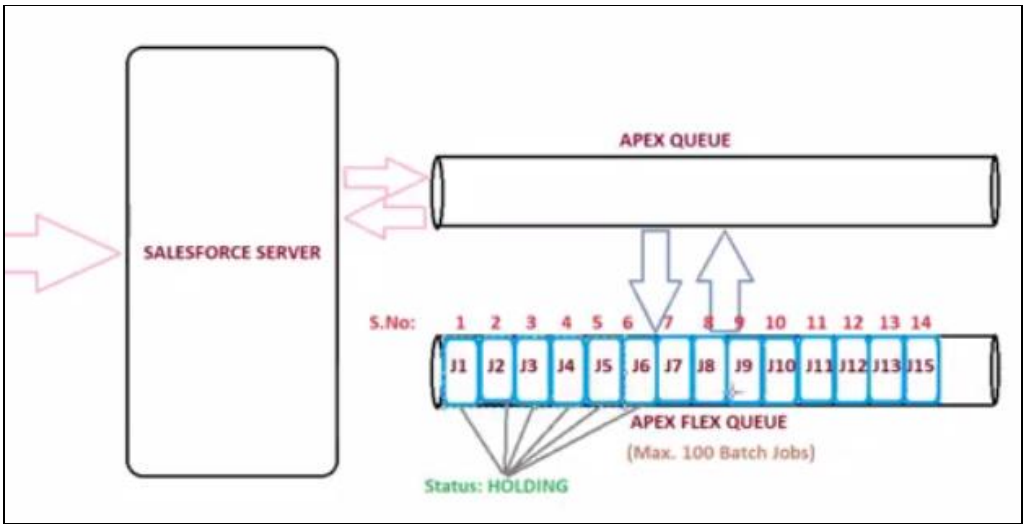
Example:

```
Global class CalculateTotalRevenueScheduleJob implements System.Schedulable
{
    Global void Execute (System.SchedulableContext sContext)
  {
      // Write the Code to Invoke Batch Job
      CalculateTotalRevenueBatch revenueBatch = new CalculateTotalRevenueBatch();
      Database.executeBatch(revenueBatch, 4);
   }
}
```

FLEX QUEUE

From an organization, we can push maximum of only five batch jobs in apex queue.

A flex queue is a queue which can hold up to 100 batch jobs.



For each job in flex queue, sales force allocates a serial number starting from one and the status as 'Holding'.

Apex queue keeps on watching the flex queue. If any jobs are waiting in the flex queue, the apex queue will pick up the first five jobs and the status of the jobs will be changed to 'Queued'.

Once the server becomes free, it takes first job from the apex queue to process the job and changes the status of the job to 'Processing'.

After the first job is processed by the server, the job status changes to 'Completed' and picks up the second job from the apex queue.

Once all the five jobs in the apex queue are processed, it will pick up another five jobs waiting in the flex queue to get processed and the process repeats.

Flex queues allows to execute preferred jobs first by reordering the jobs. But once the jobs are collected by the apex queue, they cannot be reordered.

Future Methods: Salesforce provides a set of Readymade objects by default as part of Salesforce CRM and create our own Custom objects based on the Business need.

All the Salesforce Objects has been categorized into 2 types.

1. Setup Objects: Setup Objects can be interacted with setup area/builder area of Salesforce declaratively i.e., we can interact with the metadata through mouse clicks.

Ex: User, Profile, Group, Workflow, WorkflowEmailAlert, ValidationRule, BusinessProcess, PageLayout, CustomObject, CustomField etc.

2. Non-Setup Objects: The remaining standard and custom objects are referred as "Non-Setup" objects.

Ex: Account, Contact, Opportunity, Lead, Case, Position__C, Candidate__C ...etc.

While performing the DML Operations on both Setup and Non-Setup Objects within a single transaction, then Salesforce will raise an exception "MIXED_DML_OPERATION" exception.

This exception can be avoided using "Future Methods".

Future Methods can be used to

i. Implement long running operations.

ii. Implement complex business logic and complex transactional flows.

iii. Integrate the salesforce application with external systems.

Future Methods will be always executed by placing inside the "Apex Queue" i.e., outside of the organization based on the availability of the salesforce resources.

Note

i. Future Methods should be always defined with the annotation "@future()".

ii. Future Methods should be always defined as "Static".

iii. Future Methods won't return any value back to the calling environment i.e., return type should be always "void".

iv. We can invoke the Integrations through Future Methods as below.

Ex: @future (callout = true)

v. Future Methods will execute asynchronously by placing inside the "Apex Queue".

Use Case: Write an apex program, to perform DML Operations on both "User Object" and "Account Object" within a transaction.

Class Code:

```
public class CommonHelperUtility
{
        Public static void DoDMLOperations()
    {
        // Update User Record.... (Setup Object)
        User userToDeActivate = [Select id, firstname, lastname, username, isActive
                                 from User  Where userName = 'testinguser@dl.com' and isActive = true];
        If (userToDeActivate.id != Null)
        {
            userToDeActivate.Username = 'trainingbatch1@dl.com';
            userToDeActivate.IsActive = false;
            Update userToDeActivate;
            // Invoke the Method to Create Account Record
```

```
            InsertAccountRecord();
        }
    }
    @future()
    Public static void InsertAccountRecord()
    {
        // Insert an Account Record (Non-Setup Object)
        Account acc = new Account();
            acc.Name = 'Reliance Inc.';
            acc.Rating = 'Hot';
            acc.Industry = 'Manufacturing';
            acc.AnnualRevenue = 25000000;
            acc.Phone = '9900333333';
            acc.Fax = '8899888888';
            acc.Ownership = 'Private';
            acc.Type = 'Installation Partner';
            acc.Website = 'www.reliance.com';
            acc.CustomerPriority__C = 'High';
            acc.Active__C = 'Yes';
            acc.BillingCity = 'Mumbai';
            acc.BillingState = 'Maharashtra';
           Insert acc;
    }
}
```

Execution: CommonHelperUtility.DoDMLOperations();

Limitations

i. Future Method doesn't support SObject type of parameters. It supports Primitive Type, Arrays of Primitive, and Collection of Primitive types only.

ii. Future Methods won't support Chaining Mechanism i.e., One Future Method cannot invoke another Future Method.

3. Future Methods won't provide the Tracking Mechanism i.e., while placing the Future Methods inside the Apex Queue, the Queue will allocate an ID for the Future Method, but it won't return the ID back to the calling environment.

The above problems can be overcome by using "Queueable Apex/Queueable Interface".

Implementation Steps:

Step 1: Create a Class, which implements the Interface "System.Queueable".

Ex: Public Class <ClassName> implements System.Queueable

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

Step 2: Provide the Implementation for Interface Method "Execute ()".

Ex: Public Class <ClassName> implements System.Queueable

```
{
    Public void Execute (System.QueueableContext <refName>)
        {
            // Write the Business Logic
        }
        }
```

Step 3: Invoke / Push the Queueable Class into the Apex Queue.

Ex: ID jobId = System.EnqueueJob(new <QueueableClassName>());

Step 4: Track the Status through programming from "AsyncApexJob" object.

Ex: AsyncApexJob jobDetails = [Select id, status, totalJobItems,
jobItemsProcessed, numberOFErrors, CreatedBy.Email
from AsyncApexJob Where id =: jobId];

Use Case: Create two Queueable Classes, to create "Hiring Manager" Record and an associated "Position Record".

1. Implement the Chaining Mechanism (i.e., Invoke One Queueable class from another).

2. Supply the SObject Parameter to the Queueable Class.

3. Track the Status of the Queueable Class, through Programming.

Hiring Manager Queueable:

```
public class HiringManagerQueueableHelper implements System.Queueable
{
    Public void Execute (System.QueueableContext qContext)
```

```
    {
        Hiring_Manager__C hrRecord = new Hiring_Manager__c();
            hrRecord.Name = 'Sudeep Kumar';
            hrRecord.Location__c = 'Hyderabad';
            hrRecord.Contact_Number__c = '9900445778';
            hrRecord.Email_Address__c = 'sudeep.kumar@gmail.com';
            hrRecord.Designation__c = 'Recruitment Specialist';
            hrRecord.Current_CTC__c = 1500000;
        Insert hrRecord;
        If (hrRecord.Id != Null)
        {
            System.debug('Hiring Manager Record id is....: '+ hrRecord.Id);
             // Invoke the Position Queueable Class
            Id posJobId = System.enqueueJob(new PositionQueueableHelper(hrRecord));
             // Track the Status
             AsyncApexJob jobDetails = [Select id, status, totalJobItems, jobItemsProcessed,
                                        numberOfErrors, CreatedBy.Email
                                                from AsyncApexJob Where id=: posJobId];
        }
    }
}
Position Queueable:
public class PositionQueueableHelper implements System.Queueable
{
    Hiring_Manager__C hr;
    Public PositionQueueableHelper(Hiring_Manager__C hrRecord)
    {
        hr = hrRecord;
    }
    Public void Execute (System.QueueableContext qContext)
    {
```

```
        if(hr.Id != Null)
        {
            Position__C pos = new Position__c();
                    pos.Name = 'Technical Architect';
                    pos.Location__c = hr.Location__c;
                    pos.HR_Contact_Number__c = hr.Contact_Number__c;
                    pos.HR_Email_ID__c = hr.Email_Address__c;
                    pos.Position_Status__c = 'Open Approved';
                    pos.Number_of_Vacancies__c = 1;
                    pos.Open_Date__c = System.today();
                    pos.Milestone_Date__c = System.today().AddMonths(1);
                    pos.Minimum_Budget__c = 1500000;
                    pos.Maximum_Budget__c = 2000000;
                    pos.Travel_Required__c = false;
                    pos.Passport_Required__c = true;
                    pos.Position_Description__c = 'Required 12+ years of experience in Technical Architect.';
                    pos.Skills_Required__c = 'Required 12+ years of experience in Technical Architect.';
                    pos.Hiring_Manager__c = hr.Id;

             Insert pos;
            if (pos.Id != Null)
             System.debug('Position Record Id is....: '+ pos.Id);
        }
    }
}
```

Execution: System.enqueueJob(new HiringManagerQueueableHelper());