Asynchronous vs Synchronous HandBook

WiseQuarter Education

WiseQuarter

Asynchronous vs Synchronous in Apex

Synchronous in Apex:

- Synchronous term means existing or occurring at the same time.
- Synchronous Apex means entire Apex code is executed in one single go.
- Executes at one point of time as a single transaction when we call.
- In a Synchronous call, the thread will wait until it completes its tasks before proceeding to next. In a Synchronous call, the code runs in single thread.

Synchronous:

- Quick and Immediate actions
- Transactions are immediate and serial
- Normal Governor Limits

Asynchronous in Apex:

- 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.
- Asynchronous Apex is used to run processes in a separate thread.
- One of the main benefits of running asynchronous Apex is higher governor and execution limits.
- In a Asynchronous call, the thread will not wait until it completes its tasks before proceeding to next. Instead it proceeds to next leaving it run in separate thread. In a Asynchronous call, the code runs in multiple threads which helps to do many tasks as background jobs.
- 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.
- You’ll typically use Asynchronous Apex for callouts to external systems, operations that require higher limits, and code that needs to run at a certain time.
- Asynchronous operation means that a process operating independently of other processes
- If we have to run some job that will take some time then in the case of synchronous apex we will get the limit error, heap size error, or timeout error. To avoid such issues, we can do those long-running or time-consuming operations using **Asynchronous** apex which will run in a separate thread and will not impact the main thread. So Asynchronous Apex is **used to run the process in a separate thread at a later tim**e.

Asynchronous:

- Actions that will not block the transaction or Process
- Duration is not priority
- Higher Governor limits

Asynchronous Apex can be used in long-running or time-consuming operations like:

- Sending Email to the user.
- Creating complex reports using Apex code
- Calling an external system for multiple records
- Schedule a job for a specific time
- Chaining apex code execution which might use API call
- Sharing Re-calculation

Salesforce supports four types of Asynchronous Apex:

- Future Methods (making methods Asynchronous)
- Batch Apex
- Queueable Apex
- Scheduled Apex

Future Method:

- it is a basic asynchronous feature, when we make a **web call out** or when we want to **prevent the mixed DML error**.
- A future method runs in the background, **asynchronously**.
- You can call a future method for executing **long-running** operations, such as callouts to external Web services or any operation you'd like to run in its own thread, on its own time.
- You can also use future methods to isolate **DML** operations on different sObject types to prevent the **mixed DML error**.
- Each future method is queued and executes when system resources become available. That way, the execution of your code doesn't have to wait for the completion of a long-running operation.
- Future methods are not guaranteed to execute in the same order as they are called.
- A benefit of using future methods is that some governor limits are higher, such as SOQL query limits and heap size limits.

Future method syntax:

```
public class FutureClass{

    @future
    public static void myFutureMethod(){
        // Perform some operations
    }
}
```

- Methods with the future annotation must be static methods, and can only return a void type.
- The specified parameters must be **primitive data types**, arrays of primitive data types, or collections of primitive data types.

```
@future
public static void myMethodName(List<Id> accIdList){
  List<Account> accList = [SELECT id, Name FROM Account
WHERE id : accIdList];
    }
```

- **Primitive data types: Boolean, Date, datetime, time, ID, String, all numbers…**
- Methods with the future annotation **can't take sObjects or objects as arguments**.
- *The reason why sObjects can't be passed as arguments to future methods is because the sObject can change between the time you call the method and the time it executes. In this case, the future method gets the old sObject values and can overwrite them. To work with sObjects that already exist in the database, pass the sObject ID instead (or collection of IDs) and use the ID to perform a query for the most up-to-date record.
- An Apex **callout** enables you to tightly integrate your Apex with an external service by making a call to an external Web service or sending a HTTP request from Apex code and then receiving the response. Apex provides integration with Web services that utilize SOAP and WSDL, or HTTP services (RESTful services).
- **@Future(callout=true) syntax:**

```
public class FutureClass{

    @future(callout=true)
    public static void myFutureMethod(){
        // Perform some operations
    }
}
```

Example:

Utility Class

```
public class Utility {
  @Future
  public static void seeMixedDMLException (String accName){

        Account acc = new Account();
        acc.name = accName;

        try{
            insert acc;
            }
        catch(Exception e){
         system.debug('exception Account ' + e);
        }

UserCreation.createUser('username@sss.com', 'AAAA', 'UserAp', 'xxx@mail.com');
  }

}
```

```
createUser(String userNameEmail, String UserLastName, String userAlias, String userEmail)
```

UserCreation Class

```
public class UserCreation {

  public static void createUser(String userNameEmail, String UserLastName,
String userAlias, String userEmail) {

        User brandNewUsr= new User();

        brandNewUsr.Username = userNameEmail;
        brandNewUsr.LastName = UserLastName;
        brandNewUsr.Alias = userAlias;
        brandNewUsr.Email = userEmail;

Profile prof= [SELECT id FROM Profile WHERE NAME='Standard Platform' LIMIT 1];
        brandNewUsr.ProfileId = prof.id;
        brandNewUsr.LocaleSidKey = 'en_US';
        brandNewUsr.LanguageLocaleKey = 'en_US';
        brandNewUsr.EmailEncodingKey = 'UTF-8';
        brandNewUsr.TimeZoneSidKey = 'America/Los_Angeles';

        try{
            insert brandNewUsr;
        }
        catch(Exception e){
            system.debug('exception ' + e);
        }
   }
}
```

MixedDMLError Class

```
public class MixedDMLError {
    public static void insertUserAndAccount(){

     // to create an account record

 Utility.seeMixedDMLException('AnyAccountName');
    }
 }
```

Anonymous Windows

```
MixedDMLError.insertUserAndAccount();
```

Commentary of Exam with Future method:

To avoide MixedDMLException error we need to use @Future (Best Practice)

Note: A Mixed DML operation error occurs when you try to persist in the same transaction, changes to a Setup Object and a non-Setup Object. For example, if you try to update an Opportunity record and a User record at the same time.

- **SetUp objects**: are objects that are used to interact with the metadata.(Usr,Profile, Layout)
- **Non**-SetUp Object: every other objects like those which are native(standard Objects) and Custom Objects fall into the category of Non-SetUp Objects (Account, Contact, Lead, Opportunity…).

Future Method Performance Best Practices:

Salesforce uses a **queue-based framework** to handle asynchronous processes from such sources as future methods and batch Apex. This queue is used to balance request workload across organizations.

Use the following best practices to ensure your organization is efficiently using the queue for your asynchronous processes.

- **Avoid adding large numbers of future methods to the asynchronous queue**, if possible. If more than 2,000 unprocessed requests from a single organization are in the queue, any additional requests from the same organization will be delayed while the queue handles requests from other organizations.
- Ensure that future methods execute as fast as possible. To ensure fast execution of batch jobs, minimize Web service callout times and tune queries used in your future methods. **The longer the future method executes, the more likely other queued requests are delayed when there are a large number of requests in the queue.**
- Test your future methods at scale. To help determine if delays can occur, test using an environment that generates the maximum number of future methods you'd expect to handle.
- Consider using batch Apex instead of future methods to process large numbers of records.

Example:

Create a field Account “Number of Contacts” with data type Number.

Update the field using @future annotation when the contact is created or Deleted.

ContactTrigger Class

```
trigger ContactTrigger on Contact (after insert, after delete) {
 // After events
      if(Trigger.isafter) {

            if(Trigger.isInsert) {
            ContactTriggerHandler.onAfterSave(Trigger.New);
            }

            if(Trigger.isDelete) {
            ContactTriggerHandler.onAfterSave(Trigger.Old);
            }
      }
}
```

ContactTriggerHandler Class

```
public class ContactTriggerHandler {

                              //== Method to execute on after Insert of contact

 public static void onAfterSave (List<Contact> contacts) {

      Set<Id> acctIds = new Set<Id>();     // Set to hold the account Id's from Contact

                                          // Loop to iterate over the list of contacts
      for(Contact con : contacts) {
            if(con.AccountId != null) {
                  acctIds.add(con.AccountId);
            }
      }
      if(acctIds.size() > 0 && acctIds != null) {
            updateAccount(acctIds);
      }
  }
                  //== Method to update the account based on Number of contacts

  @future
  public static void updateAccount(Set<Id> accIds) {

            // Qeury to fetch the account records based on AccountId from contact

List<Account> accnts = [SELECT ID, Name, Number_of_Contacts__c, (SELECT Id FROM
                                          Contacts) FROM Account WHERE ID = :accIds];
        System.debug('query results : ' +accnts.size());

            // Loop to iterate over the list of account records from Query results
      for(Account acc : accnts) {
      List<Contact> cons = acc.Contacts; //List to hold the contacts that related to account

         acc.Number_of_Contacts__c = cons.size();
System.debug('Account:'+ acc.Name + ' has ' + acc.Number_of_Contacts__c + ' Contact childs');
      }
      try {
            update accnts;
       } catch(DMLException e) {
            throw new stringException('Faile to update the accounts : '+e.getMessage());
      }
  }
}
```

Things to Remember

Things to Remember when using **@future** annotation

1. Future method should **return void** and it should be declared as **static.**
2. Can we pass objects as a parameter to the future method? No, we cannot pass the objects (custom, standard) as parameter. We can only pass the **Id's as a parameter**, **primitive data type or collection of primitive data types as a parameter.**
3. Why we cannot pass the object (custom, standard) as a parameter? Remember we are working with asynchronous apex they will execute only **when the resources are available** so if we use the object as a parameter then the system will not hold the current data.
4. Did future method execute in the same order they called? The answer is no, as they are not guaranteed to execute in the same order. So, can we call it a limitation? Yes, we can overcome this by using the queueable apex.
5. Will future methods run concurrently which means updating the same record by 2 future methods at a same time from different sources or the same source? Yes, and which could result in locking the record. so, need to incredibly careful while dealing with future methods.
6. Do I have to take any special care while writing the test class for future methods? Yes, you need to enclose the code between start and stop test methods.
7. Can I use future method for bulk transactions? The answer will not be likeable, but it is good to avoid future and good to use batch processing.
8. What is the limit on future calls per apex invocation? **50**, additionally there are limits base on the 24-hour time frame.
9. Can we user future method in visual force controller constructors? The answer is no.
10. Can I call a future method from another future method? No, **you cannot call one future method from another future method.**

Batch Apex:

- Batch Apex is used to run large jobs (think thousands or millions of records!) that would exceed normal processing limits.
- Using Batch Apex, you can process records asynchronously in batches (hence the name, “Batch Apex”) to stay within platform limits.
- If you have a lot of records to process, for example, data cleansing or archiving, Batch Apex is probably your best solution.
- To use batch Apex, write an Apex class that implements the Salesforce-provided interface Database.Batchable and then invoke the class programmatically.
- Every transaction starts with a new set of governor limits, making it easier to ensure that your code stays within the governor execution limits.
- If one batch fails to process successfully, all other successful batch transactions aren’t rolled back
- **Batch Apex syntax:**

```
public class BatchApexExample implements Database.Batchable<sObject>{

public (Database.QueryLocator | Iterable<sObject>) start(Database.BatchableContext BC){

       // collect the batches of records or objects to be passed to execute
}
    public void execute(Database.BatchableContext BC, List<Account> accList){

       // process each batch of records - default size is 200.
    }
    public void finish(Database.BatchableContext BC) {

       // execute any post-processing operations like sending email
    }

}
```

Start Method:

```
public(Database.QueryLocator | Iterable<sObject>) start(Database.BatchableContext bc){}
```

- To collect the records or objects to pass to the interface method **execute**, call the start method at the beginning of a batch Apex job. This method returns either a **Database.QueryLocator** object or an iterable that contains the records or objects passed to the job.
- When you’re using a simple query (**SELECT**) to generate the scope of objects in the batch job, use the **Database.QueryLocator** object. If you use a QueryLocator object, the governor limit for the total number of records retrieved by SOQL queries is bypassed.
- **QueryLocator** object, the governor limit for the total number of records retrieved by SOQL queries is bypassed and you can query up to 50 million records. However, with an Iterable, the governor limit for the total number of records retrieved by SOQL queries is still enforced.
- Use the iterable to create a complex scope for the batch job. You can also use the iterable to create your own custom process for iterating through the list.

Execute Method:

```
public void execute(Database.BatchableContext BC, list<P>){}
```

- Performs the actual processing for each chunk or “batch” of data passed to the method. The default batch size is 200 records (max 2000 records each). Batches of records are not guaranteed to execute in the order they are received from the start method.
- This method takes the following:
    - A reference to the Database.BatchableContext object.
    - A list of sObjects, such as List<sObject>, or a list of parameterized types. If you are using a Database.QueryLocator, use the returned list.

Finish Method:

```
public void finish(Database.BatchableContext BC){}
```

- Used to execute post-processing operations (for example, sending an email) and is called once after all batches are processed.

State in Batch Apex:

- Batch Apex is typically **stateless**. That means for each execution of your execute method, you receive **a fresh copy** of your object. All fields of the class are initialized, static and instance.
- Each execution of a batch Apex job is considered a discrete transaction. For example, a batch Apex job that contains 1,000 records and is executed without the optional *scope* parameter is considered five transactions of 200 records each.
- If you specify **`Database.Stateful`** in the class definition, you can maintain state across these transactions.
- 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.
- Maintaining state is useful for counting or summarizing records as they're 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.
- If you don't specify **`Database.Stateful`**, all static and instance member variables are set back to their original values.

Invoking A Batch Class:

- To invoke a batch class, simply instantiate it and then call Database.executeBatch with the instance:

```
MyBatchClass myBatchObject = new MyBatchClass();
             Id batchId = Database.executeBatch(myBatchObject);
```

- You can also optionally pass a second scope parameter to specify the number of records that should be passed into the execute method for each batch. Pro tip: you might want to limit this batch size if you are running into governor limits:

```
Id batchId = Database.executeBatch(myBatchObject, 100);
```

- Each batch Apex invocation creates an AsyncApexJob record so that you can track the job's progress. You can view the progress via SOQL or manage your job in the Apex Job Queue.

```
AsyncApexJob job = [SELECT Id, Status, JobItemsProcessed, TotalJobItems, NumberOfErrors
FROM AsyncApexJob WHERE ID = :batchId];
```

Limitations of Batch Apex:

- We can have only 5 batch jobs running at a time.
- Execution may be delayed due to server availability.
- @Future methods are not allowed.
- Future methods cannot be called from Batch Apex.

Best Practices for batch Apex:

As with future methods, there are a few things you want to keep in mind when using Batch Apex. To ensure fast execution of batch jobs, minimize Web service callout times and tune queries used in your batch Apex code. The longer the batch job executes, the more likely other queued jobs are delayed when many jobs are in the queue. Best practices include:

- Only use Batch Apex if you have more than one batch of records. If you don't have enough records to run more than one batch, you are probably better off using Queueable Apex.
- Tune any SOQL query to gather the records to execute as quickly as possible.
- Minimize the number of asynchronous requests created to minimize the chance of delays.
- Use extreme care if you are planning to invoke a batch job from a trigger. You must be able to guarantee that the trigger won't add more batch jobs than the limit.

Example:
Write 'Test' at the end of the name of all accounts with batch apex.

BatchApexExample Class

```
public class accountNameUpdateBAtchApex implements Database.Batchable<sObject>{

public Database.QueryLocator start(Database.BatchableContext BC){
       // collect the batches of records or objects to be passed to execute

      String query = 'SELECT Id, Name FROM Account';
        return Database.getQueryLocator(query);
}
    public void execute(Database.BatchableContext BC, List<Account> accList){
       // process each batch of records - default size is 200.
      for(Account acc : accList) {
            // Update the Account Name
            acc.Name = acc.Name + 'Test';
        }
        try {
            // Update the Account Record
            update accList;
        } catch(Exception e) {
            System.debug(e);
          }
    }
    public void finish(Database.BatchableContext BC) {

       // execute any post-processing operations like sending email
    }

}
```

Anonymous Windows

```
accountNameUpdateBAtchApex UpdateAccountName = new accountNameUpdateBAtchApex();
  Id batchId = Database.executeBatch(UpdateAccountName);

  // Id batchId = Database.executeBatch(UpdateAccountName, Batch size);
```

You can view the progress via SOQL or manage your job in the Apex Job Queue.
Our org -->SET UP --> Home --> write '**Apex Jobs'** to quick find box --> monitor the status of Apex jobs and optionally, abort that are in progress.

Another Way of Anonymous Windows Execution:

Anonymous Windows

```
accountNameUpdateBAtchApex UpdateAccountName = new accountNameUpdateBAtchApex();
  Id batchId = Database.executeBatch(UpdateAccountName);

AsyncApexJob job = [SELECT Id, Status, JobItemsProcessed, TotalJobItems,
                    NumberOfErrors FROM AsyncApexJob WHERE ID = :batchId];

System.debug(job.id);
System.debug(job.JobType);
System.debug(job.status);
System.debug(job.JobItemsProcessed);
```

To see the AsyncApexJob records so that you can track the job's progress.

Example:

All Account of Companies must have their company's billing address as their mailing address.

BatchApexExample Class

```
public class ContactUpdateBatchApex implements Database.Batchable<sObject>, Database.Stateful
{
   integer recordContactUpdated=0;

   public Database.QueryLocator start(Database.BatchableContext BC) {

     return Database.getQueryLocator([SELECT Id, BillingCity, BillingState , BillingCountry,
         (SELECT id,MailingCity , MailingState , MailingCountry FROM Contacts)
         FROM Account WHERE BillingCountry = 'Turkiye']);
   }
   public void execute(Database.BatchableContext BC, List<Account> accList) {
          List<Contact> cntList = new List<contact>();

             for(Account acc : accList) {
                   for(Contact cnt: acc.contacts){
                       cnt.MailingCity = acc.BillingCity;
                       cnt.MailingCountry = acc.BillingCountry;
                       cnt.MailingState = acc.BillingState;
                       cntList.add(cnt);
                       recordContactUpdated = recordContactUpdated +1;
                   }
             }
             update cntList;
       }
   public void finish(Database.BatchableContext BC) {
         system.debug('Batch apex finished the job');
        system.debug(recordContactUpdated + ' contac updated succesfully');
   }

}
```

Anonymous Windows

```
ContactUpdateBatchApex updateContacts = new accountNameUpdateBAtchApex();
  Id batchId = Database.executeBatch(updateContacts);

  // Id batchId = Database.executeBatch(updateContacts, Batch size);
```

```
integer recordContactUpdated = 0;
recordContactUpdated = recordContactUpdated +1;
system.debug(recordContactUpdated + ' contac updated succesfully');
```

These codes are writen to see the counts of updated records. (Kaç tane record Update edildiğini görmek için yazıldı)

İn developer console by clicking '**log'** : we may see the execution of '**System.debug( )**', Because code is running in batchApex class.

İf we didn't write **Database.Stateful,** we couldn't see the count of updated record (`recordContactUpdated=0`). Because Batch Apex is Stateless. İt means: all static and instance member variables are set back to their original values.

QUEUEABLE APEX:

- Queueable apex is an asynchronous apex method. It's similar to the @future method. By using this **queueable interface**, we can process an Apex that runs **for a long time** (such as web service call outs and extensive database operations).
- **Queueable Apex** is more advanced and enhanced version of future methods with some extra features.
- It has **simplicity of future methods** and the **power of Batch Apex** and mixed them together to form Queueable Apex.
- It gives you a class structure that the platform serializes for you, a simplified interface without **start** and **finish** methods and even allows you to utilize more than just primitive arguments!
- We need to **implement Queueable interface** for performing queueable operation.

Queueable Apex allows you to submit jobs for asynchronous processing similar to future methods with the following additional benefits:

- **Non-primitive types**: Your Queueable class can contain member variables of non-primitive data types, such as **sObjects** or custom Apex types. Those objects can be accessed when the job executes.
- **Monitoring:** When you submit your job by invoking the **System.enqueueJob()** method, the method returns the **ID of the AsyncApexJob** record. You can use this ID to identify your job and **monitor its progress**, either through the Salesforce user interface in the Apex Jobs page, or programmatically by querying your record from AsyncApexJob.
- **Chaining jobs:** You can chain one job to another job by starting a second job from a running job. Chaining jobs is useful if you need to do some sequential processing. we can chain up to 50 jobs; However, for Developer Edition and Trial orgs, the maximum stack depth for chained jobs is 5, which means that you can chain jobs four times and the maximum number of jobs in the chain is 5, including the initial parent queueable job.

Queueable Apex asynchronously **runs in background** in its own thread. All jobs are added to a Queue and each job **runs when the system resource are available**. Also, we can monitor the progress of these jobs using the job id.

There are some **limitations of Queueable Apex** Class:

- Queueable can't handle millions of records in one job.
- Only one job can be scheduled at a time.
- In the developer edition stack depth for chaining the jobs is limited to 5 including the parent job.

Differences between Future and Queueable Apex:

Future Method	Queueable Job
1. Future will never use to work on SObjects or object types.	1. Queueable Jobs can contain the member variable as SObjects or custom Apex Types.
2. When using the future method we cannot monitor the jobs which are in process.	2. When using queueable jobs it will make the AsyncApexJob which we can monitor like Scheduled jobs.
3. The future method cannot be called inside the future or batch class.	3. Queueable Apex can be called from the future and batch class.
4. The future method will never be queued.	4. Using Queueable Apex will chain up to queueable jobs and in Developer Edition it is only 5 Jobs.

*** **Batch Apex** is used to run large jobs (think thousands or millions of records!) that would exceed normal processing limits. Using Batch Apex, you can process records asynchronously in batches (hence the name, “Batch Apex”) to stay within platform limits.

which one is best of future method or Queueable apex?

It's difficult to say which one is "best" because it depends on the specific use case. Both Future methods and Queueable Apex have their own advantages and disadvantages, and the choice between the two will depend on the requirements of your project.

Future methods are a type of asynchronous Apex that allow you to run a method in the background. They are useful for tasks that need to be performed asynchronously, such as sending emails or making API calls. Future methods are easy to use and are suitable for simple, short-running tasks.

Queueable Apex is also a type of asynchronous Apex, but it is more powerful and flexible than Future methods. Queueable Apex allows you to create complex, long-running processes that are executed in the background. It also provides additional features such as the ability to chain multiple Queueable jobs together and to specify the order in which they should be executed. However, Queueable Apex is more complex to use than Future methods and is better suited for more advanced use cases.

In general, if you have a simple, short-running task that needs to be performed asynchronously, you should use a Future method. If you have a more complex, long-running process that needs to be performed asynchronously, you should use Queueable Apex.

Queueable Syntax:

```
public class SomeClass implements Queueable {

      public void execute(QueueableContext context) {
            // Any code here
      }
}
```

To add this class as a job on the queue, call this method:

```
ID jobID = System.enqueueJob(new AsyncExecutionExample());
```

After you submit your queueable class for execution, the job is added to the queue and will be processed when system resources become available. You can monitor the status of your job programmatically by querying AsyncApexJob or through the user interface in Setup by entering Apex Jobs in the Quick Find box, then selecting **Apex Jobs**.

To query information about your submitted job, perform a SOQL query on AsyncApexJob by filtering on the job ID that the **System.enqueueJob** method returns. This example uses the jobID variable that was obtained in the previous example.

```
AsyncApexJob jobInfo = [SELECT Status,NumberOfErrors FROM AsyncApexJob WHERE Id=:jobID];
```

Similar to future jobs, queueable jobs don’t process batches, and so the number of processed batches and the number of total batches are always zero.

Queueable Apex has the ability to (1) start a long-running operation and **get an ID** for it, (2) pass complex data types like **sObjects** to a job, and (3) **chain jobs** in a sequence.

Unlike other methods, Queueable Apex has an interface that allows developers to **add jobs to the queue and monitor them**. Each queued job runs when system resources become available.

One benefit of using the interface is that some governor limits are higher than for synchronous Apex, such as heap size limits. However, keep in mind that you can add only up to 50 jobs within a single transaction.

Another huge benefit of Queueable Apex is the ability to chain jobs in a sequence. You can chain one queueable job to another by starting the second job from a running job. There are no limits on the depth of chained jobs. However, while chaining jobs, only one child job can exist for each parent job.

Example:

Get all account records. Set Test as parent account for each account and then update records in database.

Class

```
public class UpdateParentAccountQueueableClass implements system.Queueable{

   public void execute(QueueableContext context) {
        Account TestedAcc = [SELECT id,name FROM account WHERE name Like 'Test%' LIMIT 1];
        List<Account> accList = [SELECT id,name ,ParentId FROM Account ];
//List<Account> accList = [SELECT id,name ,ParentId FROM Account WHERE id !=:TestedAcc.Id ];

        for(Account acc: accList){
            if(acc.id != TestedAcc.id){
                acc.ParentId = TestedAcc.Id;
            }
        }
        update  accList;
    }
}
```

Anonymous Windows

```
UpdateParentAccountQueueableClass QC = new UpdateParentAccountQueuableClass();
 ID QueueableId = System.enqueueJob(QC);

AsyncApexJob job = [SELECT Id, Status, JobItemsProcessed, TotalJobItems,
                           NumberOfErrors FROM AsyncApexJob WHERE ID = : QueueableId];

System.debug(job.id);                    // id of job
System.debug(job.JobType);              // Queueable
System.debug(job.status);              //
System.debug(job.JobItemsProcessed);  // number of items to be processed
```

***

It's difficult to give a recommendation without more context about your specific use case. Here is a brief overview of each option:

- **Future methods** are used to run asynchronous Apex, meaning that the code is executed in a separate thread at a later time. This is useful when you need to run a long-running process or make a callout to an external service and don't want to block the user's interaction with the application. Future methods are best suited for relatively short-running processes that don't need to be invoked immediately.
- **Batch Apex** allows you to define a job that processes records in batches, rather than processing all records in a single transaction. This is useful when you have a large number of records that need to be processed and you want to break the processing down into smaller chunks to avoid reaching governor limits.
- **Queueable Apex** is similar to batch Apex in that it allows you to define a job that processes records in the background. However, queueable Apex is more flexible in that it allows you to chain jobs together and specify the order in which they should be executed. This is useful when you need to perform a series of dependent operations, such as sending an email after updating a record.

Ultimately, the choice between these options will depend on the specific requirements of your use case. You may want to consider factors such as the amount of data that needs to be processed, the complexity of the processing logic, and the required performance and scalability of the process.

***

Apex Scheduler:

- Scheduled apex is all about to run a piece of apex code at some particular time within a period of time.
- Apex Scheduler is an **ability to invoke an Apex class to run at specific times**.
- Schedule apex in Salesforce is a class that runs at a regular interval of time.
- To schedule an apex class, we need to **implement an interface Schedulable**.
- The parameter of this method is a **SchedulableContext** object.
    - **SchedulableContext** represents the parameter type of a method in a class that implements the Schedulable interface and contains the scheduled job ID using the **getTriggerID()** method. This interface is implemented internally by Apex.
    - With that ID, we can track the Progress of a Scheduled Job through a query on **CronTrigger**. The ID is of type **CronTrigger**. (Moreover, we can go to **Setup ---> Scheduled Jobs** too, for monitoring.)
    - After a class has been scheduled, a **CronTrigger** object is created that represents the scheduled job. It provides a **getTriggerId** method that returns the ID of a **CronTrigger** API object.

Use cases for Apex Scheduler:

- Sending Periodic notifications to users.
- Tasks related to real-time updates.
- Newsletter's on the 15th of every month.

Apex Scheduler Limits:

- You can only have **100** scheduled Apex jobs at one time.
- The maximum number of scheduled Apex executions per a 24-hour period is 250,000 or the number of user licenses in your organization **multiplied by 200, whichever is greater**.
- Synchronous **Web service callouts are not supported** from Scheduled Apex.
- In this way, you can schedule apex classes to execute at particular intervals of time by implementing the schedulable class. **Sometimes, execution may be delayed based on service availability**.

Basic scheduled Apex Syntax:

```
public class MySchedule implements schedule {
      public void execute (SchedulableContext SC) {
            /* Logic goes here */
      }
}
```

We can call schedule class in two way:

1. User interface
   (Click “Schedule Apex” button in Setup / Develop / Apex Classes to create a schedule apex job.)
2. Programmatically

Syntax:

```
MySchedule mine = new MySchedule ();
String cronExp = '20 30 8 10 2 ?';
String jobID = System.schedule('Any Job Name', cronExp, mine);
```

Another way of invoking:

```
system.schedule('CRON SET','0 0 12 * * ?', new MySchedule() );
```

CRON EXPRESSION is used to define the scheduling time. it has 6 to 7 inputs.

```
'.. .. .. .. .. .. ..'
```

Second	Minutes	Hours	Day of Month	Month	Day of Week	Year (Optional)
0 - 59	0 - 59	0 - 23	1 - 31	1 - 12	1 - 7	Null - 2022

Or the following:
Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, Dec

Or the following:
Sun, Mon, Tue, Wed, Thu, Fri, Sat

WiseQuarter

- **?** No Value (don't Care)
- ***** All Values
- **?** Specifies no specific value
- **/** Specifies increments. The number before the slash specifies when the intervals will begin, and the number after the slash is the interval amount.
- **#** Specifies the nth day of the month, in the format weekday#day_of_month
- **L** Specifies the end of a range (last)
- **W** Specifies the nearest weekday(Monday-Friday) of the given day

S.No	Scheduling a Job	CRON_EXP
1.	Run every day at 1 PM.	0 0 13 * * ?
2.	Run the last Friday of every month at 10 PM.	0 0 22 ? * 6L
3.	At 12:00 AM every day	0 0 0 ? * * *
4.	Run Monday through Friday at 10 AM.	0 0 10 ? * MON-FRI
5.	Run every day at 8 PM during the year 2024.	0 0 20 * * ? 2024
6.	At 12:00 PM every day	0 0 12 * * ?
7.	At 10.00 AM every day	0 0 10 ? * * 0 0 10 * * ? 0 0 10 * * ? *
8.	At 3:00 PM every day	0 0 15 ? * * *
9.	Every 30 minutes	0 30 * * * ?
10.	Runs the last Friday of every month at 7:00 PM.	0 0 19 ? * 6L
11.	At 5:25 AM on the 15th day of every month	0 25 5 15 * ?
12.	At 5:15 PM on the third Friday of every month	0 15 17 ? * 6#3
13.	Every minute starting at 3:00 PM and ending at 3:05 PM, every day	0 0-5 15 * * ?
14.	At 5:15 PM every Monday, Tuesday, Wednesday, Thursday and Friday	0 15 17 ? * MON-FRI
15.	Runs every day at 11:00 PM during the year 2022.	0 0 23 * * ? 2022

Example:

Create a list for open opportunities that should have closed by the current date

And

Create a task on each one to remind the owner to update the opportunity.

Schedulable Class

```
public class OpportunityController implements schedulable {

  public void execute(SchedulableContext sc){
       //   hem kanmamis hem de closedate i gecmis olan opp getirildi
    List<opportunity> oppList = [SELECT id,name,closeDate, OwnerId FROM opportunity
                                          WHERE closeDate < Today AND isClosed= False];

     //  her opp ile ilgili task olustur

        if( oppList.size() > 0  ) {

           List<Task> tskList = new List<Task>();

             for(Opportunity opp : oppList ){

                    Task tsk = new Task();
                    tsk.Subject = 'Opp Close date Passed';
                    tsk.Status = 'Not Started';
                    tsk.WhatId = opp.id;
                    tsk.OwnerId = opp.OwnerId;
                    tsk.ActivityDate = Date.today() + 3 ; // DueDate
                    tskList.add(tsk);

             }
             Database.insert(tskList);
       }
   }
}
```

Anonymous Windows

```
OpportunityController mySc = new OpportunityController ();

String cronExp = '20 30 8 10 2 ?';

      System.schedule('AnyJobName', cronExp, mySc);
```

We can see by clicking "**Scheduled Apex**" button in Setup-Home with this name. and also the status of scheduled can be seen by clicking '**Apex Jobs**' in setup-Home

*****When executed once; if we don't delete (deactivate) this schedulable class, it will run at scheduled time and we can't update the schedulable class. (To delete, click "Scheduled Apex" button in Setup-Home)**

Example:
Everyday check if there is an account including 'test' in the name.
İf so, delete all account which has 'test' in the name. (Scheduling BatchApex Class)

BatchApex Class

```
public class AccountDeleteBatchApex implements Database.Batchable<sObject> {

    public Database.QueryLocator  start(Database.BatchableContext bc) {
        // collect the batches of records or objects to be passed to execute

     return Database.getQueryLocator([ SELECT Id, Name FROM Account WHERE name LIKE '%test%' ]);
    }

    public void execute(Database.BatchableContext bc, List<Account> records){
        delete records;
    }

    public void finish(Database.BatchableContext bc){
        System.debug('Operation Completed');
    }
}
```

Schedule Class

```
public class AccountDeleteScheduleApex implements schedulable {

    public void execute(SchedulableContext sc){

        AccountDeleteBatchApex deletingSc = new AccountDeleteBatchApex();

          Database.executeBatch(deletingSc);
    }
}
```

Anonymous Windows

```
AccountDeleteScheduleApex mySc = new AccountDeleteScheduleApex ();

String cronExp = '20 30 8 10 2 ?';

      System.schedule('Deleting Set', cronExp, mySc);
```

Example: (System.scheduleBatch)