<https://developer.salesforce.com/docs/atlas.en-us.apexcode.meta/apexcode/apex_batch_interface.htm>

## Testing Batch Apex

When testing your batch Apex, you can test only one execution of the execute method. Use the scope parameter of the executeBatch method to limit the number of records passed into the execute method to ensure that you aren’t running into governor limits.

The executeBatch method starts an asynchronous process. When you test batch Apex, make certain that the asynchronously processed batch job is finished before testing against the results. Use the Test methods startTest and stopTest around the executeBatch method to ensure that it finishes before continuing your test. All asynchronous calls made after the startTest method are collected by the system. When stopTest is executed, all asynchronous processes are run synchronously. If you don’t include the executeBatch method within the startTest and stopTest methods, the batch job executes at the end of your test method. This execution order applies for Apex saved using API version 25.0 and later, but not for earlier versions.

For Apex saved using API version 22.0 and later, exceptions that occur during the execution of a batch Apex job invoked by a test method are passed to the calling test method. As a result, these exceptions cause the test method to fail. If you want to handle exceptions in the test method, enclose the code in try and catch statements. Place the catch block after the stopTest method. However, with Apex saved using Apex version 21.0 and earlier, such exceptions don’t get passed to the test method and don’t cause test methods to fail.

Note

Asynchronous calls, such as @future or executeBatch, called in a startTest, stopTest block, do not count against your limits for the number of queued jobs.

The following example tests the [OwnerReassignment class](https://developer.salesforce.com/docs/" \l "batch_apex_examples).

public static testMethod void testBatch() {

user u = [SELECT ID, UserName FROM User

WHERE username='testuser1@acme.com'];

user u2 = [SELECT ID, UserName FROM User

WHERE username='testuser2@acme.com'];

String u2id = u2.id;

// Create 200 test accounts - this simulates one execute.

// Important - the Salesforce.com test framework only allows you to

// test one execute.

List <Account> accns = new List<Account>();

for(integer i = 0; i<200; i++){

Account a = new Account(Name='testAccount'+'i',

Ownerid = u.ID);

accns.add(a);

}

insert accns;

Test.StartTest();

OwnerReassignment reassign = new OwnerReassignment();

reassign.query='SELECT ID, Name, Ownerid ' +

'FROM Account ' +

'WHERE OwnerId=\'' + u.Id + '\'' +

' LIMIT 200';

reassign.email='admin@acme.com';

reassign.fromUserId = u.Id;

reassign.toUserId = u2.Id;

ID batchprocessid = Database.executeBatch(reassign);

Test.StopTest();

System.AssertEquals(

database.countquery('SELECT COUNT()'

+' FROM Account WHERE OwnerId=\'' + u2.Id + '\''),

200);

}

}

## Batch Apex Governor Limits

Keep in mind the following governor limits for batch Apex.

* Up to 5 batch jobs can be queued or active concurrently.
* Up to 100 Holding batch jobs can be held in the Apex flex queue.
* In a running test, you can submit a maximum of 5 batch jobs.
* The maximum number of batch Apex method executions per 24-hour period is 250,000, or the number of user licenses in your org multiplied by 200—whichever is greater. Method executions include executions of the start, execute, and finish methods. This limit is for your entire org and is shared with all asynchronous Apex: Batch Apex, Queueable Apex, scheduled Apex, and future methods. To check how many asynchronous Apex executions are available, make a request to REST API limits resource. See [List Organization Limits](https://developer.salesforce.com/docs/atlas.en-us.224.0.api_rest.meta/api_rest/dome_limits.htm) in the [REST API Developer Guide](https://developer.salesforce.com/docs/atlas.en-us.224.0.api_rest.meta/api_rest/). The licenses that count toward this limit are full Salesforce user licenses or App Subscription user licenses. Chatter Free, Chatter customer users, Customer Portal User, and partner portal User licenses aren’t included.
* The batch Apex start method can have up to 15 query cursors open at a time per user. The batch Apex execute and finish methods each have a limit of five open query cursors per user.
* A maximum of 50 million records can be returned in the Database.QueryLocator object. If more than 50 million records are returned, the batch job is immediately terminated and marked as Failed.
* If the start method of the batch class returns a QueryLocator, the optional scope parameter of Database.executeBatch can have a maximum value of 2,000. If set to a higher value, Salesforce chunks the records returned by the QueryLocator into smaller batches of up to 2,000 records. If the start method of the batch class returns an iterable, the scope parameter value has no upper limit. However, if you use a high number, you can run into other limits.
* If no size is specified with the optional scope parameter of Database.executeBatch, Salesforce chunks the records returned by the start method into batches of 200. The system then passes each batch to the execute method. Apex governor limits are reset for each execution of execute.
* The start, execute, and finish methods can implement up to 100 callouts each.
* Only one batch Apex job's start method can run at a time in an org. Batch jobs that haven’t started yet remain in the queue until they're started. Note that this limit doesn’t cause any batch job to fail and execute methods of batch Apex jobs still run in parallel if more than one job is running.

## Batch Apex Best Practices

* Use extreme care if you are planning to invoke a batch job from a trigger. You must be able to guarantee that the trigger does not add more batch jobs than the limit. In particular, consider API bulk updates, import wizards, mass record changes through the user interface, and all cases where more than one record can be updated at a time.
* When you call Database.executeBatch, Salesforce only places the job in the queue. Actual execution can be delayed based on service availability.
* When testing your batch Apex, you can test only one execution of the execute method. Use the scope parameter of the executeBatch method to limit the number of records passed into the execute method to ensure that you aren’t running into governor limits.
* The executeBatch method starts an asynchronous process. When you test batch Apex, make certain that the asynchronously processed batch job is finished before testing against the results. Use the Test methods startTest and stopTest around the executeBatch method to ensure that it finishes before continuing your test.
* Use Database.Stateful with the class definition if you want to share instance member variables or data across job transactions. Otherwise, all member variables are reset to their initial state at the start of each transaction.
* Methods declared as future aren’t allowed in classes that implement the Database.Batchable interface.
* Methods declared as future can’t be called from a batch Apex class.
* When a batch Apex job is run, email notifications are sent to the user who submitted the batch job. If the code is included in a managed package and the subscribing org is running the batch job, notifications are sent to the recipient listed in the Apex Exception Notification Recipient field.
* Each method execution uses the standard governor limits anonymous block, Visualforce controller, or WSDL method.
* Each batch Apex invocation creates an AsyncApexJob record. To construct a SOQL query to retrieve the job’s status, number of errors, progress, and submitter, use the AsyncApexJob record’s ID. For more information about the AsyncApexJob object, see [AsyncApexJob](https://developer.salesforce.com/docs/atlas.en-us.224.0.object_reference.meta/object_reference/sforce_api_objects_asyncapexjob.htm" \o "HTML (New Window)" \t "_blank) in the Object Reference for Salesforce .
* For each 10,000 AsyncApexJob records, Apex creates an AsyncApexJob record of type BatchApexWorker for internal use. When querying for all AsyncApexJob records, we recommend that you filter out records of type BatchApexWorker using the JobType field. Otherwise, the query returns one more record for every 10,000 AsyncApexJob records. For more information about the AsyncApexJob object, see [AsyncApexJob](https://developer.salesforce.com/docs/atlas.en-us.224.0.object_reference.meta/object_reference/sforce_api_objects_asyncapexjob.htm" \o "HTML (New Window)" \t "_blank) in the Object Reference for Salesforce .
* All methods in the class must be defined as global or public.
* For a sharing recalculation, we recommend that the execute method delete and then re-create all Apex managed sharing for the records in the batch. This process ensures that sharing is accurate and complete.
* Batch jobs queued before a Salesforce service maintenance downtime remain in the queue. After service downtime ends and when system resources become available, the queued batch jobs are executed. If a batch job was running when downtime occurred, the batch execution is rolled back and restarted after the service comes back up.
* Minimize the number of batches, if possible. 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. 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 batch jobs execute as fast as possible. 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.
* If you use batch Apex with Database.QueryLocator to access external objects via an OData adapter for Salesforce Connect:
  + You must enable Request Row Counts on the external data source, and each response from the external system must include the total row count of the result set.
  + We recommend enabling Server Driven Pagination on the external data source and having the external system determine page sizes and batch boundaries for large result sets. Typically, server-driven paging can adjust batch boundaries to accommodate changing data sets more effectively than client-driven paging.

When Server Driven Pagination is disabled on the external data source, the OData adapter controls the paging behavior (client-driven). If external object records are added to the external system while a job runs, other records can be processed twice. If external object records are deleted from the external system while a job runs, other records can be skipped.

* + When Server Driven Pagination is enabled on the external data source, the batch size at runtime is the smaller of the following:
    - Batch size specified in the scope parameter of Database.executeBatch. Default is 200 records.
    - Page size returned by the external system. We recommend that you set up your external system to return page sizes of 200 or fewer records.
* Batch Apex jobs run faster when the start method returns a QueryLocator object that doesn't include related records via a subquery. Avoiding relationship subqueries in a QueryLocator allows batch jobs to run using a faster, chunked implementation. If the start method returns an iterable or a QueryLocator object with a relationship subquery, the batch job uses a slower, non-chunking, implementation. For example, if the following query is used in the QueryLocator, the batch job uses a slower implementation because of the relationship subquery:

|  |  |
| --- | --- |
| 1 | SELECT Id, (SELECT id FROM Contacts) FROM Account |

* A better strategy is to perform the subquery separately, from within the execute method, which allows the batch job to run using the faster, chunking implementation.

## Chaining Batch Jobs

Starting with API version 26.0, you can start another batch job from an existing batch job to chain jobs together. Chain a batch job to start a job after another one finishes and when your job requires batch processing, such as when processing large data volumes. Otherwise, if batch processing isn’t needed, consider using [Queueable Apex](https://developer.salesforce.com/docs/atlas.en-us.apexcode.meta/apexcode/apex_queueing_jobs.htm" \o "Take control of your asynchronous Apex processes by using the Queueable interface. This interface enables you to add jobs to the queue and monitor them. Using the interface is an enhanced way of running your asynchronous Apex code compared to using future methods.).

You can chain a batch job by calling Database.executeBatch or System.scheduleBatch from the finish method of the current batch class. The new batch job will start after the current batch job finishes.

For previous API versions, you can’t call Database.executeBatch or System.scheduleBatch from any batch Apex method. The version that’s used is the version of the running batch class that starts or schedules another batch job. If the finish method in the running batch class calls a method in a helper class to start the batch job, the API version of the helper class doesn’t matter.

Using Callouts in Batch Apex

To use a [callout](https://developer.salesforce.com/docs/atlas.en-us.apexcode.meta/apexcode/apex_callouts.htm) in batch Apex, specify Database.AllowsCallouts in the class definition. For example:

|  |  |  |
| --- | --- | --- |
| 1 | global class SearchAndReplace implements Database.Batchable<sObject>, | |
| 2 | Database.AllowsCallouts{ |

|  |  |
| --- | --- |
| 3 | } |

# Using Batch Apex

To use batch Apex, write an Apex class that implements the Salesforce-provided interface Database.Batchable and then invoke the class programmatically.

To monitor or stop the execution of the batch Apex job, from Setup, enter Apex Jobs in the Quick Find box, then select **Apex Jobs**.

## Implementing the Database.Batchable Interface

The Database.Batchable interface contains three methods that must be implemented.

start method:

|  |  |
| --- | --- |
| 1 | global (Database.QueryLocator | Iterable<sObject>) start(Database.BatchableContext bc) {} |

* execute method:

|  |  |
| --- | --- |
| 1 | global void execute(Database.BatchableContext BC, list<P>){} |

* To do the required processing for each chunk of data, use the execute method. This method is called for each batch of records that you pass to it.
* 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.

Batches of records tend to execute in the order in which they’re received from the start method. However, the order in which batches of records execute depends on various factors. The order of execution isn’t guaranteed.

* finish method:

|  |  |
| --- | --- |
| 1 | global void finish(Database.BatchableContext BC){} |

* To send confirmation emails or execute post-processing operations, use the finish method. This method is called after all batches are processed.

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 from Database.executeBatch is considered five transactions of 200 records each. The Apex governor limits are reset for each transaction. If the first transaction succeeds but the second fails, the database updates made in the first transaction are not rolled back.

## Using Database.BatchableContext

All the methods in the Database.Batchable interface require a reference to a Database.BatchableContext object. Use this object to track the progress of the batch job.

The following is the instance method with the Database.BatchableContext object:

| **Name** | **Arguments** | **Returns** | **Description** |
| --- | --- | --- | --- |
| getJobID |  | ID | Returns the ID of the [AsyncApexJob](https://developer.salesforce.com/docs/atlas.en-us.224.0.object_reference.meta/object_reference/sforce_api_objects_asyncapexjob.htm" \t "_blank" \o "HTML (New Window)) object associated with this batch job as a string. Use this method to track the progress of records in the batch job. You can also use this ID with the [System.abortJob](https://developer.salesforce.com/docs/atlas.en-us.apexcode.meta/apexcode/apex_methods_system_system.htm" \l "apex_System_System_abortJob" \o "Stops the specified job. The stopped job is still visible in the job queue in the Salesforce user interface.) method. |

global void finish(Database.BatchableContext BC){

// Get the ID of the AsyncApexJob representing this batch job

// from Database.BatchableContext.

// Query the AsyncApexJob object to retrieve the current job's information.

AsyncApexJob a = [SELECT Id, Status, NumberOfErrors, JobItemsProcessed,

TotalJobItems, CreatedBy.Email

FROM AsyncApexJob WHERE Id =

:BC.getJobId()];

// Send an email to the Apex job's submitter notifying of job completion.

Messaging.SingleEmailMessage mail = new Messaging.SingleEmailMessage();

String[] toAddresses = new String[] {a.CreatedBy.Email};

mail.setToAddresses(toAddresses);

mail.setSubject('Apex Sharing Recalculation ' + a.Status);

mail.setPlainTextBody

('The batch Apex job processed ' + a.TotalJobItems +

' batches with '+ a.NumberOfErrors + ' failures.');

Messaging.sendEmail(new Messaging.SingleEmailMessage[] { mail });

}

Using Database.QueryLocator to Define Scope

The start method can return either a Database.QueryLocator object that contains the records to use in the batch job or an iterable.

The following example uses a Database.QueryLocator:

global class SearchAndReplace implements Database.Batchable<sObject>{

global final String Query;

global final String Entity;

global final String Field;

global final String Value;

global SearchAndReplace(String q, String e, String f, String v){

Query=q; Entity=e; Field=f;Value=v;

}

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

return Database.getQueryLocator(query);

}

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

for(sobject s : scope){

s.put(Field,Value);

}

update scope;

}

global void finish(Database.BatchableContext BC){

}

}

## Using the Database.executeBatch Method to Submit Batch Jobs

You can use the Database.executeBatch method to programmatically begin a batch job.

Important

When you call Database.executeBatch, Salesforce adds the process to the queue. Actual execution can be delayed based on service availability.

The 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. Use this parameter when you have many operations for each record being passed in and are running into governor limits. By limiting the number of records, you are limiting the operations per transaction. This value must be greater than zero. If the start method of the batch class returns a QueryLocator, the optional scope parameter of Database.executeBatch can have a maximum value of 2,000. If set to a higher value, Salesforce chunks the records returned by the QueryLocator into smaller batches of up to 2,000 records. If the start method of the batch class returns an iterable, the scope parameter value has no upper limit. However, if you use a high number, you can run into other limits.

The Database.executeBatch method returns the ID of the AsyncApexJob object, which you can use to track the progress of the job. For example:

|  |  |  |
| --- | --- | --- |
| 1 | ID batchprocessid = Database.executeBatch(reassign); | |
| 2 |  |

|  |  |  |  |
| --- | --- | --- | --- |
| 3 | AsyncApexJob aaj = [SELECT Id, Status, JobItemsProcessed, TotalJobItems, NumberOfErrors | | |
| 4 | | FROM AsyncApexJob WHERE ID =: batchprocessid ]; |

You can also use this ID with the [System.abortJob](https://developer.salesforce.com/docs/atlas.en-us.apexcode.meta/apexcode/apex_methods_system_system.htm" \l "apex_System_System_abortJob" \o "Stops the specified job. The stopped job is still visible in the job queue in the Salesforce user interface.) method.

For more information, see [AsyncApexJob](https://developer.salesforce.com/docs/atlas.en-us.224.0.object_reference.meta/object_reference/sforce_api_objects_asyncapexjob.htm" \t "_blank" \o "HTML (New Window)) in the Object Reference for Salesforce .

## Holding Batch Jobs in the Apex Flex Queue

With the Apex flex queue, you can submit up to 100 batch jobs.

The outcome of Database.executeBatch is as follows.

* The batch job is placed in the Apex flex queue, and its status is set to Holding.
* If the Apex flex queue has the maximum number of 100 jobs, Database.executeBatch throws a LimitException and doesn’t add the job to the queue.

Note

If your org doesn’t have Apex flex queue enabled, Database.executeBatch adds the batch job to the batch job queue with the Queued status. If the concurrent limit of queued or active batch job has been reached, a LimitException is thrown, and the job isn’t queued.

**Reordering Jobs in the Apex Flex Queue**

While submitted jobs have a status of Holding, you can reorder them in the Salesforce user interface to control which batch jobs are processed first. To do so, from Setup, enter Apex Flex Queue in the Quick Find box, then select **Apex Flex Queue**.

Alternatively, you can use Apex methods to reorder batch jobs in the flex queue. To move a job to a new position, call one of the [System.FlexQueue methods](https://developer.salesforce.com/docs/atlas.en-us.apexcode.meta/apexcode/apex_class_System_FlexQueue.htm" \l "apex_System_FlexQueue_methods). Pass the method the job ID and, if applicable, the ID of the job next to the moved job’s new position. For example:

|  |  |
| --- | --- |
| 1 | Boolean isSuccess = System.FlexQueue.moveBeforeJob(jobToMoveId, jobInQueueId); |

You can reorder jobs in the Apex flex queue to prioritize jobs. For example, you can move a batch job up to the first position in the holding queue to be processed first when resources become available. Otherwise, jobs are processed “first-in, first-out”—in the order in which they’re submitted.

When system resources become available, the system picks up the next job from the top of the Apex flex queue and moves it to the batch job queue. The system can process up to five queued or active jobs simultaneously for each organization. The status of these moved jobs changes from Holding to Queued. Queued jobs get executed when the system is ready to process new jobs. You can monitor queued jobs on the Apex Jobs page.

## Batch Job Statuses

The following table lists all possible statuses for a batch job along with a description of each.

| **Status** | **Description** |
| --- | --- |
| Holding | Job has been submitted and is held in the Apex flex queue until system resources become available to queue the job for processing. |
| Queued | Job is awaiting execution. |
| Preparing | The start method of the job has been invoked. This status can last a few minutes depending on the size of the batch of records. |
| Processing | Job is being processed. |
| Aborted | Job aborted by a user. |
| Completed | Job completed with or without failures. |
| Failed | Job experienced a system failure. |