Per-Transaction Apex Limits

| **Description** | **Synchronous Limit** | **Asynchronous Limit** |
| --- | --- | --- |
| Total number of SOQL queries issued1 | 100 | 200 |
| Total number of records retrieved by SOQL queries | 50,000 | |
| Total number of records retrieved by Database.getQueryLocator | 10,000 | |
| Total number of SOSL queries issued | 20 | |
| Total number of records retrieved by a single SOSL query | 2,000 | |
| Total number of DML statements issued2 | 150 | |
| Total number of records processed as a result of DML statements, Approval.process, or database.emptyRecycleBin | 10,000 | |
| Total stack depth for any Apex invocation that recursively fires triggers due to insert, update, or delete statements3 | 16 | |
| Total number of callouts (HTTP requests or Web services calls) in a transaction | 100 | |
| Maximum cumulative timeout for all callouts (HTTP requests or Web services calls) in a transaction | 120 seconds | |
| Maximum number of methods with the future annotation allowed per Apex invocation | 50 | |
| Maximum number of Apex jobs added to the queue with System.enqueueJob | 50 | |
| Total number of sendEmail methods allowed | 10 | |
| Total heap size4 | 6 MB | 12 MB |
| Maximum CPU time on the Salesforce servers5 | 10,000 milliseconds | 60,000 milliseconds |
| Maximum execution time for each Apex transaction | 10 minutes | |
| Maximum number of push notification method calls allowed per Apex transaction | 10 | |
| Maximum number of push notifications that can be sent in each push notification method call | 2,000 | |

Static Apex Limits

| **Description** | **Limit** |
| --- | --- |
| Default timeout of callouts (HTTP requests or Web services calls) in a transaction | 10 seconds |
| Maximum size of callout request or response (HTTP request or Web services call)1 | 6 MB for synchronous Apex or 12 MB for asynchronous Apex |
| Maximum SOQL query run time before Salesforce cancels the transaction | 120 seconds |
| Maximum number of class and trigger code units in a deployment of Apex | 5,000 |
| For loop list batch size | 200 |
| Maximum number of records returned for a Batch Apex query in Database.QueryLocator | 50 million |

**Email Limits**

Inbound Email Limits

|  |  |
| --- | --- |
| Email Services: Maximum Number of Email Messages Processed  (Includes limit for On-Demand Email-to-Case) | Number of user licenses multiplied by 1,000; maximum 1,000,000 |
| Email Services: Maximum Size of Email Message (Body and Attachments) | 10 MB1 |
| On-Demand Email-to-Case: Maximum Email Attachment Size | 25 MB |
| On-Demand Email-to-Case: Maximum Number of Email Messages Processed  (Counts toward limit for Email Services) | Number of user licenses multiplied by 1,000; maximum 1,000,000 |

1 The maximum size of email messages for Email Services varies depending on language and character set. The size of an email message includes the email headers, body, attachments, and encoding. As a result, an email with a 25 MB attachment likely exceeds the 25 MB size limit for an email message after accounting for the headers, body, and encoding..

When defining email services, note the following:

* An email service only processes messages it receives at one of its addresses.
* Salesforce limits the total number of messages that all email services combined, including On-Demand Email-to-Case, can process daily. Messages that exceed this limit are bounced, discarded, or queued for processing the next day, depending on how you configure the [failure response settings](https://help.salesforce.com/apex/HTViewHelpDoc?id=code_email_services_editing.htm&language=en_US#FailureResponseSettings) for each email service. Salesforce calculates the limit by multiplying the number of user licenses by 1,000; maximum 1,000,000. For example, if you have 10 licenses, your org can process up to 10,000 email messages a day.
* Email service addresses that you create in your [sandbox](https://help.salesforce.com/apex/HTViewHelpDoc?id=create_test_instance.htm&language=en_US) cannot be copied to your production org.
* For each email service, you can tell Salesforce to [send error email messages to a specified address](https://help.salesforce.com/apex/HTViewHelpDoc?id=code_email_services_editing.htm&language=en_US#route_errors) instead of the sender's email address.
* Email services reject email messages and notify the sender if the email (combined body text, body HTML, and attachments) exceeds approximately 10 MB (varies depending on language and character set).

Outbound Email: Limits for Single and Mass Email Sent Using Apex

Using the API or Apex, you can send single emails to a maximum of 5,000 external email addresses per day based on Greenwich Mean Time (GMT). Single emails sent using the email author or composer in Salesforce don't count toward this limit. There’s no limit on sending individual emails to contacts, leads, person accounts, and users in your org directly from account, contact, lead, opportunity, case, campaign, or custom object pages.

When sending single emails, keep in mind:

* You can specify up to 100 recipients for the To field and up to 25 recipients for the CC and BCC fields in eachSingleEmailMessage.
* If you use SingleEmailMessage to email your org’s internal users, specifying the user’s ID in setTargetObjectId means the email doesn’t count toward the daily limit. However, specifying internal users’ email addresses in setToAddressesmeans the email does count toward the limit.

You can send mass email to a maximum of 5,000 external email addresses per day per org based on Greenwich Mean Time (GMT)

Note

* The single and mass email limits don't take unique addresses into account. For example, if you havejohndoe@example.com in your email 10 times, that counts as 10 against the limit.
* You can send an unlimited amount of email to your org’s internal users, which includes portal users.
* You can send mass emails only to contacts, person accounts, leads, and your org’s internal users.
* In Developer Edition orgs and orgs evaluating Salesforce during a trial period, you can send mass email to no more than 10 external email addresses per day. This lower limit doesn’t apply if your org was created before the Winter ’12 release and already had mass email enabled with a higher limit. Additionally, your org can send single emails to a maximum of 15 email addresses per day.
* Maximum of 10 sendEmail methods per transaction. Use the [Limits methods](https://developer.salesforce.com/docs/atlas.en-us.apexcode.meta/apexcode/apex_methods_system_limits.htm#apex_methods_system_limits) to verify the number of sendEmailmethods in a transaction.
* Transaction based errors
  + We can send 100 emails per SingleEmailMessage
* Per day (24 hour) email limit
  + We can send 1000 emails per day
  + You can send mass email to a maximum of 5,000 external email addresses per day per org based on Greenwich Mean Time (GMT).

Links:-

<http://www.jitendrazaa.com/blog/salesforce/resolve-24-hour-apex-email-limit-error-in-salesforce/>

https://developer.salesforce.com/docs/atlas.en-us.apexcode.meta/apexcode/apex\_gov\_limits.htm#email\_limits\_section

**Asynchronous Apex**

Apex offers multiple ways for running your Apex code asynchronously. Choose the asynchronous Apex feature that best suits your needs.

This table lists the asynchronous Apex features and when to use each.

| **Asynchronous Apex Feature** | **When to Use** |
| --- | --- |
| [Future Methods](https://developer.salesforce.com/docs/atlas.en-us.apexcode.meta/apexcode/apex_invoking_future_methods.htm) | * When you have a long-running method and need to prevent delaying an Apex transaction * When you make callouts to external Web services * To segregate DML operations and bypass the mixed save DML error |
| [Queueable Apex](https://developer.salesforce.com/docs/atlas.en-us.apexcode.meta/apexcode/apex_queueing_jobs.htm) | * To start a long-running operation and get an ID for it * To pass complex types to a job * To chain jobs |
| [Batch Apex](https://developer.salesforce.com/docs/atlas.en-us.apexcode.meta/apexcode/apex_batch.htm) | * For long-running jobs with large data volumes that need to be performed in batches, such as database maintenance jobs * For jobs that need larger query results than regular transactions allow |
| [Scheduled Apex](https://developer.salesforce.com/docs/atlas.en-us.apexcode.meta/apexcode/apex_scheduler.htm) | * To schedule an Apex class to run on a specific schedule |

Relationship Level[parent to child and Child to parent].

|  |  |  |
| --- | --- | --- |
| Relationship queries | Relationship query limits | * No more than 35 child-to-parent relationships can be specified in a query. A custom object allows up to 25 relationships, so you can reference all the child-to-parent relationships for a custom object in one query. * No more than 20 parent-to-child relationships can be specified in a query. * In each specified relationship, no more than five levels can be specified in a child-to-parent relationship. For example, Contact.Account.Owner.FirstName (three levels). * In each specified relationship, only one level of parent-to-child relationship can be specified in a query. For example, if the FROM clause specifies Account, the SELECTclause can specify only the Contact or other objects at that level. It could not specify a child object of Contact. |

**Visualforce Limits**

| **Limit** | **Value** |
| --- | --- |
| Maximum response size for a Visualforce page | Less than 15 MB |
| Maximum view state size in a Visualforce page | 135 KB |
| Maximum size of a Visualforce email template | 1 MB |
| Maximum file size for a file uploaded using a Visualforce page | 10 MB |
| Maximum size of HTML response *before* rendering, when Visualforce page is rendered as PDF | Less than 15 MB |
| Maximum PDF file size for a Visualforce page rendered as a PDF | 60 MB |
| Maximum total size of all images included in a Visualforce page rendered as a PDF | 30 MB |
| Maximum request size of a JavaScript remoting call | 4 MB |
| Maximum response size of a JavaScript remoting call | 15 MB |
| Default timeout for a JavaScript remoting call | 30,000 milliseconds (30 seconds) |
| Maximum timeout for a JavaScript remoting call | 120,000 milliseconds (120 seconds) |
| Maximum rows retrieved by queries for a single Visualforce page request | 50,000 |
| Maximum rows retrieved by queries for a single Visualforce page request in read-only mode | 1,000,000 |
| Maximum collection items that can be iterated in an iteration component such as<apex:pageBlockTable> and <apex:repeat> | 1,000 |
| Maximum collection items that can be iterated in an iteration component such as<apex:pageBlockTable> and <apex:repeat> in read-only mode | 10,000 |
| Maximum field sets that can be displayed on a single Visualforce page. | 50 |
| Maximum records that can be handled by StandardSetController | 10,000 |

**Sandbox Limits**

* You can refresh a full sandbox 29 days after you created or last refreshed it. If you delete a full sandbox within those 29 days, you need to wait until after the 29 day period, from the date of last refresh or creation, to replace it.
* You can refresh a Partial Copy sandbox 5 days after you created or last refreshed it. If you delete a Partial Copy sandbox within those 5 days, you need to wait until after the 5 day period, from the date of last refresh or creation, to replace it.
* You can refresh a Developer or Developer Pro sandbox once per day.
* Enterprise Edition includes licenses for 25 Developer sandboxes and 1 Partial Copy sandbox.
* Performance Edition includes licenses for 1 full sandbox, 5 Developer Pro sandboxes, 100 Developer sandbox, and 1 Partial Copy sandbox.
* Unlimited Edition includes licenses for 1 full sandbox, 5 Developer Pro sandboxes, 100 Developer sandboxes and 1 Partial Copy sandbox.
* Professional Edition includes licenses for 10 Developer sandboxes. (Change sets aren’t available.)
* If you need licenses for more sandboxes, contact Salesforce to order sandboxes for your organization.

Sandbox Storage Limits

* Partial Copy sandboxes have a 5 GB of files and a 5 GB of data storage limit.
* Developer Pro sandboxes have a 1 GB of files and a 1 GB of data storage limit.
* Developer sandboxes have a 200 MB of files and a 200 MB of data storage limit.
* Full sandboxes have the same storage limit as your production organization.
* Sandboxes don’t send email notifications when storage limits are reached. However, if you reach the storage limit of your sandbox, you can’t save new data in your sandbox. To check your storage limits, from Setup, enter Storage Usagein the Quick Find box, then select **Storage Usage** in your sandbox.

**Future Methods**

 A benefit of using future methods is that some governor limits are higher, such as SOQL query limits and heap size limits.

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. Methods with the futureannotation cannot take sObjects or objects as arguments.

The reason why sObjects can’t be passed as arguments to future methods is because the sObject might change between the time you call the method and the time it executes. In this case, the future method will get the old sObject values and might 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. EX:-

global class FutureMethodRecordProcessing

{

@future

public static void processRecords(List<ID> recordIds)

{

// Get those records based on the IDs

List<Account> accts = [SELECT Name FROM Account WHERE Id IN :recordIds];

// Process records

}

}

You can invoke future methods the same way you invoke any other method. However, a future method can’t invoke another future method.

Methods with the future annotation have the following limits:

* No more than 50 method calls per Apex invocation

The maximum number of future method invocations per a 24-hour period is 250,000 or the number of user licenses in your organization multiplied by 200, whichever is greater. This limit is for your entire org and is shared with all asynchronous Apex: Batch Apex, Queueable Apex, scheduled Apex, and future methods

Testing Future Methods

To test methods defined with the future annotation, call the class containing the method in a [startTest()](https://developer.salesforce.com/docs/atlas.en-us.apexcode.meta/apexcode/apex_methods_system_test.htm" \l "apex_System_Test_startTest" \o "Marks the point in your test code when your test actually begins. Use this method when you are testing governor limits.), [stopTest()](https://developer.salesforce.com/docs/atlas.en-us.apexcode.meta/apexcode/apex_methods_system_test.htm" \l "apex_System_Test_stopTest" \o "Marks the point in your test code when your test ends. Use this method in conjunction with the startTest method.)code block. All asynchronous calls made after the startTest method are collected by the system. When stopTest is executed, all asynchronous processes are run synchronously.

**Apex Scheduler**

To invoke Apex classes to run at specific times, first implement the Schedulable interface for the class, then specify the schedule using either the Schedule Apex page in the Salesforce user interface, or the System.schedule method.

You can only have 100 scheduled Apex jobs at one time. You can evaluate your current count by viewing the Scheduled Jobs page in Salesforce and creating a custom view with a type filter equal to “Scheduled Apex”. You can also programmatically query the CronTrigger and CronJobDetail objects to get the count of Apex scheduled jobs.

Use extreme care if you’re planning to schedule a class from a trigger. You must be able to guarantee that the trigger won’t add more scheduled classes 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.

If there are one or more active scheduled jobs for an Apex class, you cannot update the class or any classes referenced by this class through the Salesforce user interface. However, you can enable deployments to update the class with active scheduled jobs by using the Metadata API (for example, when using the Force.com IDE). See “Deployment Connections for Change Sets” in the Salesforce Help.

Testing the Apex Scheduler

The following is an example of how to test using the Apex scheduler.

The System.schedule method starts an asynchronous process. This means that when you test scheduled Apex, you must ensure that the scheduled job is finished before testing against the results. Use the Test methods startTest and stopTestaround the System.schedule method to ensure 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 System.schedule method within the startTest and stopTest methods, the scheduled job executes at the end of your test method for Apex saved using Salesforce API version 25.0 and later, but not in earlier versions.

Apex Scheduler Limits

* You can only have 100 scheduled Apex jobs at one time. You can evaluate your current count by viewing the Scheduled Jobs page in Salesforce and creating a custom view with a type filter equal to “Scheduled Apex”. You can also programmatically query the CronTrigger and CronJobDetail objects to get the count of Apex scheduled jobs.

Apex Scheduler Notes and Best Practices

Salesforce schedules the class for execution at the specified time. Actual execution may be delayed based on service availability.

Use extreme care if you’re planning to schedule a class from a trigger. You must be able to guarantee that the trigger won’t add more scheduled classes 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.

Though it's possible to do additional processing in the execute method, we recommend that all processing take place in a separate class.

Synchronous Web service callouts are not supported from scheduled Apex. To be able to make callouts, make an asynchronous callout by placing the callout in a method annotated with @future(callout=true) and call this method from scheduled Apex. However, if your scheduled Apex executes a batch job, callouts are supported from the batch class. See Using Batch Apex.

Apex jobs scheduled to run during a Salesforce service maintenance downtime will be scheduled to run after the service comes back up, when system resources become available. If a scheduled Apex job was running when downtime occurred, the job is rolled back and scheduled again after the service comes back up. Note that after major service upgrades, there might be longer delays than usual for starting scheduled Apex jobs because of system usage spikes.

**Batch Apex**

**Link:-** **https://developer.salesforce.com/docs/atlas.en-us.apexcode.meta/apexcode/apex\_batch\_interface.htm**

A developer can now employ batch Apex to build complex, long-running processes that run on thousands of records on the Force.com platform. Batch Apex operates over small batches of records, covering your entire record set and breaking the processing down to manageable chunks. For example, a developer could build an archiving solution that runs on a nightly basis, looking for records past a certain date and adding them to an archive. Or a developer could build a data cleansing operation that goes through all Accounts and Opportunities on a nightly basis and updates them if necessary, based on custom criteria.

Batch Apex is exposed as an interface that must be implemented by the developer. Batch jobs can be programmatically invoked at runtime using Apex.

You can only have five queued or active batch jobs at one time. You can evaluate your current count by viewing the Scheduled Jobs page in Salesforce or programmatically using SOAP API to query the AsyncApexJob object.

**Using Batch Apex**

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. For example, a batch Apex job for the Account object can return a QueryLocatorfor all account records (up to 50 million records) in an org. Another example is a sharing recalculation for the Contact object that returns a QueryLocator for all account records in an org.

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.

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.

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. |

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:

global class SearchAndReplace implements Database.Batchable<sObject>,

Database.AllowsCallouts{

}

Using State in Batch Apex

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.

global class SummarizeAccountTotal implements

Database.Batchable<sObject>, Database.Stateful{

}

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.

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 the REST API limits resource. See [List Organization Limits](https://developer.salesforce.com/docs/atlas.en-us.210.0.api_rest.meta/api_rest/dome_limits.htm) in the [Force.com REST API Developer Guide](https://developer.salesforce.com/docs/atlas.en-us.210.0.api_rest.meta/api_rest/). The licenses that count toward this limit are full Salesforce user licenses or Force.com 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.executeBatchcan 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 will 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.210.0.object_reference.meta/object_reference/sforce_api_objects_asyncapexjob.htm" \o "HTML (New Window)" \t "_blank) in the *Object Reference for Salesforce and Force.com.*
* 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.210.0.object_reference.meta/object_reference/sforce_api_objects_asyncapexjob.htm" \o "HTML (New Window)" \t "_blank) in the *Object Reference for Salesforce and Force.com.*
* 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.

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.

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.

**Queueable Apex**

**Link:-** **https://developer.salesforce.com/docs/atlas.en-us.apexcode.meta/apexcode/apex\_queueing\_jobs.htm**

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, which is an enhanced way of running your asynchronous Apex code compared to using future methods.

For Apex processes that run for a long time, such as extensive database operations or external Web service callouts, you can run them asynchronously by implementing the Queueable interface and adding a job to the Apex job queue. In this way, your asynchronous Apex job runs in the background in its own thread and doesn’t delay the execution of your main Apex logic. Each queued job runs when system resources become available. A benefit of using the Queueable interface methods is that some governor limits are higher than for synchronous Apex, such as heap size limits.

Queueable jobs are similar to future methods in that they’re both queued for execution, but they provide you with these additional benefits.

* Getting an ID for your job: When you submit your job by invoking the System.enqueueJob method, the method returns the ID of the new job. This ID corresponds to 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.
* Using 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.
* 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 processing that depends on another process to have run first.

Testing Queueable Jobs

This example shows how to test the execution of a queueable job in a test method. A queueable job is an asynchronous process. To ensure that this process runs within the test method, the job is submitted to the queue between the Test.startTest and Test.stopTest block. The system executes all asynchronous processes started in a test method synchronously after the Test.stopTest statement. Next, the test method verifies the results of the queueable job by querying the account that the job created.

Chaining Jobs

If you need to run a job after some other processing is done first by another job, you can chain queueable jobs. To chain a job to another job, submit the second job from the execute() method of your queueable class. You can add only one job from an executing job, which means that only one child job can exist for each parent job. For example, if you have a second class called SecondJob that implements the Queueable interface, you can add this class to the queue in the execute() method as follows:

Apex allows only one web service callout from chained queueable jobs, and it must come from the parent job.

Queueable Apex Limits

* The execution of a queued job counts once against the [shared limit for asynchronous Apex method executions](https://developer.salesforce.com/docs/atlas.en-us.apexcode.meta/apexcode/apex_gov_limits.htm#asyncExecutionLimit).
* You can add up to 50 jobs to the queue with System.enqueueJob in a single transaction. To check how many queueable jobs have been added in one transaction, call [Limits.getQueueableJobs()](https://developer.salesforce.com/docs/atlas.en-us.apexcode.meta/apexcode/apex_methods_system_limits.htm" \l "apex_System_Limits_apex_System_Limits_getQueueableJobs" \o "Returns the number of queueable jobs that have been added to the queue per transaction. A queueable job corresponds to a class that implements the Queueable interface.).
* No limit is enforced on the depth of chained jobs, which means that you can chain one job to another job and repeat this process with each new child job to link it to a new child job. For Developer Edition and Trial organizations, 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.
* When chaining jobs, you can add only one job from an executing job with System.enqueueJob, which means that only one child job can exist for each parent queueable job. Starting multiple child jobs from the same queueable job isn’t supported.

**Exposing Apex Classes as REST Web Services**

You can expose your Apex classes and methods so that external applications can access your code and your application through the REST architecture.

This is an overview of how to expose your Apex classes as REST web services. You'll learn about the class and method annotations and see code samples that show you how to implement this functionality.

**Introduction to Apex REST**

You can expose your Apex class and methods so that external applications can access your code and your application through the REST architecture. This is done by defining your Apex class with the @RestResource annotation to expose it as a REST resource. Similarly, add annotations to your methods to expose them through REST. For example, you can add the @HttpGet annotation to your method to expose it as a REST resource that can be called by an HTTP GET request. For more information, see [Apex REST Annotations](https://developer.salesforce.com/docs/atlas.en-us.apexcode.meta/apexcode/apex_classes_annotations_rest.htm)

These are the classes containing methods and properties you can use with Apex REST.

| **Class** | **Description** |
| --- | --- |
| [RestContext Class](https://developer.salesforce.com/docs/atlas.en-us.apexcode.meta/apexcode/apex_methods_system_restcontext.htm#apex_methods_system_restcontext) | Contains the RestRequest and RestResponse objects. |
| [request](https://developer.salesforce.com/docs/atlas.en-us.apexcode.meta/apexcode/apex_methods_system_restcontext.htm#apex_System_RestContext_request) | Represents an object used to pass data from an HTTP request to an Apex RESTful Web service method. |
| [response](https://developer.salesforce.com/docs/atlas.en-us.apexcode.meta/apexcode/apex_methods_system_restcontext.htm#apex_System_RestContext_response) | Represents an object used to pass data from an Apex RESTful Web service method to an HTTP response. |

**Apex REST Annotations**

Six new annotations have been added that enable you to expose an Apex class as a RESTful Web service.

* [@RestResource](https://developer.salesforce.com/docs/atlas.en-us.apexcode.meta/apexcode/apex_classes_annotation_rest_resource.htm)(urlMapping='/*yourUrl*')
* [@HttpDelete](https://developer.salesforce.com/docs/atlas.en-us.apexcode.meta/apexcode/apex_classes_annotation_http_delete.htm)
* [@HttpGet](https://developer.salesforce.com/docs/atlas.en-us.apexcode.meta/apexcode/apex_classes_annotation_http_get.htm)
* [@HttpPatch](https://developer.salesforce.com/docs/atlas.en-us.apexcode.meta/apexcode/apex_classes_annotation_http_patch.htm)
* [@HttpPost](https://developer.salesforce.com/docs/atlas.en-us.apexcode.meta/apexcode/apex_classes_annotation_http_post.htm)
* [@HttpPut](https://developer.salesforce.com/docs/atlas.en-us.apexcode.meta/apexcode/apex_classes_annotation_http_put.htm)

Apex REST Methods

Apex REST supports two formats for representations of resources: JSON and XML. JSON representations are passed by default in the body of a request or response, and the format is indicated by the Content-Type property in the HTTP header. You can retrieve the body as a Blob from the HttpRequest object if there are no parameters to the Apex method. If parameters are defined in the Apex method, an attempt is made to deserialize the request body into those parameters. If the Apex method has a non-void return type, the resource representation is serialized into the response body.

These return and parameter types are allowed:

Apex primitives (excluding sObject and Blob).

sObjects

Lists or maps of Apex primitives or sObjects (only maps with String keys are supported).

User-defined types that contain member variables of the types listed above.

Methods annotated with @HttpGet or @HttpDelete should have no parameters. This is because GET and DELETE requests have no request body, so there's nothing to deserialize.

A single Apex class annotated with @RestResource can't have multiple methods annotated with the same HTTP request method. For example, the same class can't have two methods annotated with @HttpGet.

Note-[while exposing the apex classes, we need to have a authentication either by 1.Outh 2.0 or 2.Session ID].

Rest API guide: https://developer.salesforce.com/docs/atlas.en-us.210.0.api\_rest.meta/api\_rest/quickstart.htm