**Abstract**

Apex Code is the Force.com programming language used to write custom, robust business logic. As with any programming language, there are key coding principles and best practices that will help you write efficient, scalable code. This article illustrates many of the key best practices for writing and designing Apex Code solutions on the Force.com platform.

Best Practice #1: Bulkify your Code

Bulkifying Apex code refers to the concept of making sure the code properly handles more than one record at a time. When a batch of records initiates Apex, a single instance of that Apex code is executed, but it needs to handle all of the records in that given batch. For example, a trigger could be invoked by an Force.com SOAP API call that inserted a batch of records. So if a batch of records invokes the same Apex code, all of those records need to be processed as a bulk, in order to write scalable code and avoid hitting governor limits.

Here is an example of poorly written code that only handles one record:

01

trigger accountTestTrggr on Account (before insert, before update) {

02

03

//This only handles the first record in the Trigger.new collection

04

//But if more than one Account initiated this trigger, those additional records

05

//will not be processed

06

Account acct = Trigger.new[0];

07

List<Contact> contacts = [select id, salutation, firstname, lastname, email

08

from Contact where accountId = :acct.Id];

09

10

}

The issue is that only one Account record is handled because the code explicitly accesses only the first record in the Trigger.new collection by using the syntax Trigger.new[0]. Instead, the trigger should properly handle the entire collection of Accounts in the Trigger.new collection.

Here is a sample of how to handle all incoming records:

01

trigger accountTestTrggr on Account (before insert, before update) {

02

03

List<String> accountNames = new List<String>{};

04

05

//Loop through all records in the Trigger.new collection

06

for(Account a: Trigger.new){

07

//Concatenate the Name and billingState into the Description field

08

a.Description = a.Name + ':' + a.BillingState

09

}

10

11

}

Notice how this revised version of the code iterates across the entire Trigger.new collection with a for loop. Now if this trigger is invoked with a single Account or up to 200 Accounts, all records are properly processed.

Best Practice #2: Avoid SOQL Queries or DML statements inside FOR Loops

The previous Best Practice talked about the importance of handling all incoming records in a bulk manner. That example showed use of a for loop to iterate over all of the records in the Trigger.new collection. A common mistake is that queries or DML statements are placed inside a for loop. There is a governor limit that enforces a maximum number of SOQL queries. There is another that enforces a maximum number of DML statements (insert, update, delete, undelete). When these operations are placed inside a for loop, database operations are invoked once per iteration of the loop making it very easy to reach these governor limits.

Instead, move any database operations outside of for loops. If you need to query, query once, retrieve all the necessary data in a single query, then iterate over the results. If you need to modify the data, batch up data into a list and invoke your DML once on that list of data.

Here is an example showing both a query and a DML statement inside a for loop:

01

trigger accountTestTrggr on Account (before insert, before update) {

02

03

//For loop to iterate through all the incoming Account records

04

for(Account a: Trigger.new) {

05

06

//THIS FOLLOWING QUERY IS INEFFICIENT AND DOESN'T SCALE

07

//Since the SOQL Query for related Contacts is within the FOR loop, if this trigger is initiated

08

//with more than 100 records, the trigger will exceed the trigger governor limit

09

//of maximum 100 SOQL Queries.

10

11

List<Contact> contacts = [select id, salutation, firstname, lastname, email

12

from Contact where accountId = :a.Id];

13

14

for(Contact c: contacts) {

15

System.debug('Contact Id[' + c.Id + '], FirstName[' + c.firstname + '],

16

LastName[' + c.lastname +']');

17

c.Description=c.salutation + ' ' + c.firstName + ' ' + c.lastname;

18

19

//THIS FOLLOWING DML STATEMENT IS INEFFICIENT AND DOESN'T SCALE

20

//Since the UPDATE dml operation is within the FOR loop, if this trigger is initiated

21

//with more than 150 records, the trigger will exceed the trigger governor limit

22

//of 150 DML Operations maximum.

23

24

update c;

25

}

26

}

27

}

Since there is a SOQL query within the for loop that iterates across all the Account objects that initiated this trigger, a query will be executed for each Account. An individual Apex request gets a maximum of 100 SOQL queries before exceeding that governor limit. So if this trigger is invoked by a batch of more than 100 Account records, the governor limit will throw a runtime exception.

In this example, because there is a limit of 150 DML operations per request, a governor limit will be exceeded after the 150th contact is updated.

Here is the optimal way to 'bulkify' the code to efficiently query the contacts in a single query and only perform a single update DML operation.

01

trigger accountTestTrggr on Account (before insert, before update) {

02

//This queries all Contacts related to the incoming Account records in a single SOQL query.

03

//This is also an example of how to use child relationships in SOQL

04

List<Account> accountsWithContacts = [select id, name, (select id, salutation, description,

05

firstname, lastname, email from Contacts)

06

from Account where Id IN :Trigger.newMap.keySet()];

07

08

List<Contact> contactsToUpdate = new List<Contact>{};

09

// For loop to iterate through all the queried Account records

10

for(Account a: accountsWithContacts){

11

// Use the child relationships dot syntax to access the related Contacts

12

for(Contact c: a.Contacts){

13

System.debug('Contact Id[' + c.Id + '], FirstName[' + c.firstname + '], LastName[' + c.lastname +']');

14

c.Description=c.salutation + ' ' + c.firstName + ' ' + c.lastname;

15

contactsToUpdate.add(c);

16

}

17

}

18

19

//Now outside the FOR Loop, perform a single Update DML statement.

20

update contactsToUpdate;

21

}

Now if this trigger is invoked with a single account record or up to 200 account records, only one SOQL query and one update statement is executed.

Best Practice #3: Bulkify your Helper Methods

This best practice is similar to the previous one: make sure any code that runs a query or DML operation does it in a bulk manner and doesn't execute within an iteration or a for loop. Executing queries or DML operations within an iteration adds risk that the governor limits will be exceeded. This is also true for any helper or utility methods an Apex request executes.

As discussed in the SFDC:ApexGovernorLimits article, governor limits are calculated at runtime. After the request is initiated (Trigger, Visualforce page, etc.), any Apex code executed in that transaction applies and shares the governor limits. So if a trigger uses some Apex methods written in a helper class, it's important that those shared Apex methods are properly designed to handle bulk records. These methods should be written to be invoked with a set of records, especially if the method has a SOQL query or DML operation.

For example, if the Apex method performs a SOQL query, that method should receive a collection (Array, List, Set, etc.) of records so when it performs the query, it can perform the query for all records in the Apex transaction. Otherwise, if the Apex method is called individually for each record being processed, the Apex transaction will inefficiently run queries and possibly exceed the allowed number of queries allowed in that transaction. The same is true for DML statements in Apex methods.

So please make sure any utility or helper methods are efficiently written to handle collections of records. This will avoid unnecessarily executing inefficient queries and DML operations.

Best Practice #4: Using Collections, Streamlining Queries, and Efficient For Loops

It is important to use Apex Collections to efficiently query data and store the data in memory. A combination of using collections and streamlining SOQL queries can substantially help writing efficient Apex code and avoid governor limits.

Here is a sample that uses collections inefficiently:

01

trigger accountTrigger on Account (before delete, before insert, before update) {

02

03

//This code inefficiently queries the Opportunity object in two seperate queries

04

List<Opportunity> opptysClosedLost = [select id, name, closedate, stagename

05

from Opportunity where

06

accountId IN :Trigger.newMap.keySet() and StageName='Closed - Lost'];

07

08

List<Opportunity> opptysClosedWon = [select id, name, closedate, stagename

09

from Opportunity where

10

accountId IN :Trigger.newMap.keySet() and StageName='Closed - Won'];

11

12

for(Account a : Trigger.new){

13

14

//This code inefficiently has two inner FOR loops

15

//Redundantly processes the List of Opportunity Lost

16

for(Opportunity o: opptysClosedLost){

17

if(o.accountid == a.id)

18

System.debug('Do more logic here...');

19

}

20

21

//Redundantly processes the List of Opportunity Won

22

for(Opportunity o: opptysClosedWon){

23

if(o.accountid == a.id)

24

System.debug('Do more logic here...');

25

}

26

}

27

}

The main issue with the previous snippet is the unnecessary querying of the opportunity records in two separate queries. Use the power of the SOQL where clause to query all data needed in a single query. Another issue here is the use of two inner for loops that redundantly loop through the list of opportunity records just trying to find the ones related to a specific account. Look at the following revision:

01

trigger accountTrigger on Account (before delete, before insert, before update) {

02

03

//This code efficiently queries all related Closed Lost and

04

//Closed Won opportunities in a single query.

05

List<Account> accountWithOpptys = [select id, name, (select id, name, closedate,

06

stagename from Opportunities where accountId IN :Trigger.newMap.keySet()

07

and (StageName='Closed - Lost' or StageName = 'Closed - Won'))

08

from Account where Id IN :Trigger.newMap.keySet()];

09

10

//Loop through Accounts only once

11

for(Account a : accountWithOpptys){

12

13

//Loop through related Opportunities only once

14

for(Opportunity o: a.Opportunities){

15

if(o.StageName == 'Closed - Won'){

16

System.debug('Opportunity Closed Won...do some more logic here...');

17

}else if(o.StageName =='Closed - Lost'){

18

System.debug('Opportunity Closed Lost...do some more logic here...');

19

}

20

}

21

22

}

23

}

This revised sample only executes one query for all related opportunities and only has one inner for loop to apply the same logic, but in a much more efficient, governor-friendly manner.

Best Practice #5: Streamlining Multiple Triggers on the Same Object

It is important to avoid redundancies and inefficiencies when deploying multiple triggers on the same object. If developed independently, it is possible to have redundant queries that query the same dataset or possibly have redundant for statements.

Note that it is very important to detail exactly how governor limits are applied when multiple triggers are deployed on the same object. For starters, you do not have any explicit control over which trigger gets initiated first. Secondly, each trigger that is invoked does not get its own governor limits. Instead, all code that is processed, including the additional triggers, share those available resources.

So instead of only the one trigger getting a maximum of 100 queries, all triggers on that same object will share those 100 queries. That is why it is critical to ensure that the multiple triggers are efficient and no redundancies exist.

Best Practice #6: Querying Large Data Sets

The total number of records that can be returned by SOQL queries in a request is 50,000. If returning a large set of queries causes you to exceed your heap limit, then a SOQL query for loop must be used instead. It can process multiple batches of records through the use of internal calls to query and queryMore.

For example, if the results are too large, the syntax below causes a runtime exception:

1

//A runtime exception is thrown if this query returns enough records to exceed your heap limit.

2

Account[] accts = [SELECT id FROM account];

Instead, use a SOQL query for loop as in one of the following examples:

01

// Use this format for efficiency if you are executing DML statements

02

// within the for loop. Be careful not to exceed the 150 DML statement limit.

03

04

Account[] accts = new Account[];

05

06

for (List<Account> acct : [SELECT id, name FROM account

07

WHERE name LIKE 'Acme']) {

08

// Your logic here

09

accts.add(acct);

10

}

11

12

update accts;

Let the Force.com platform chunk your large query results into batches of 200 records by using this syntax where the SOQL query is in the for loop definition, and then handle the individual datasets in the for loop logic.

Best Practice #7: Use of the Limits Apex Methods to Avoid Hitting Governor Limits

Apex has a System class called Limits that lets you output debug messages for each governor limit. There are two versions of every method: the first returns the amount of the resource that has been used in the current context, while the second version contains the word limit and returns the total amount of the resource that is available for that context.

The following example shows how to embed these types of statements in your code and ultimately determine if or when you are about to exceed any governor limits. Using either the System Log or Debug Logs, you can evaluate the output to see how the specific code is performing against the governor limits. Additionally, you can embed logic in the Apex code directly to throw error messages before reaching a governor limit. The code sample below has an IF statement to evaluate if the trigger is about to update too many Opportunities.

Here is an example of how you can use a combination of System.debug statements and the Limits Apex class to generate some very useful output as it relates to governor limits and the overall efficiency of your code.

01

trigger accountLimitExample on Account (after delete, after insert, after update) {

02

03

System.debug('Total Number of SOQL Queries allowed in this Apex code context: ' + Limits.getLimitQueries());

04

System.debug('Total Number of records that can be queried in this Apex code context: ' + Limits.getLimitDmlRows());

05

System.debug('Total Number of DML statements allowed in this Apex code context: ' + Limits.getLimitDmlStatements() );

06

System.debug('Total Number of CPU usage time (in ms) allowed in this Apex code context: ' + Limits.getLimitCpuTime());

07

08

// Query the Opportunity object

09

List<Opportunity> opptys =

10

[select id, description, name, accountid, closedate, stagename from Opportunity where accountId IN: Trigger.newMap.keySet()];

11

12

System.debug('1. Number of Queries used in this Apex code so far: ' + Limits.getQueries());

13

System.debug('2. Number of rows queried in this Apex code so far: ' + Limits.getDmlRows());

14

System.debug('3. Number of DML statements used so far: ' + Limits.getDmlStatements());

15

System.debug('4. Amount of CPU time (in ms) used so far: ' + Limits.getCpuTime());

16

17

//NOTE:Proactively determine if there are too many Opportunities to update and avoid governor limits

18

if (opptys.size() + Limits.getDMLRows() > Limits.getLimitDMLRows()) {

19

System.debug('Need to stop processing to avoid hitting a governor limit. Too many related Opportunities to update in this trigger');

20

System.debug('Trying to update ' + opptys.size() + ' opportunities but governor limits will only allow ' + Limits.getLimitDMLRows());

21

for (Account a : Trigger.new) {

22

a.addError('You are attempting to update the addresses of too many accounts at once. Please try again with fewer accounts.');

23

}

24

}

25

26

else{

27

System.debug('Continue processing. Not going to hit DML governor limits');

28

System.debug('Going to update ' + opptys.size() + ' opportunities and governor limits will allow ' + Limits.getLimitDMLRows());

29

for(Account a : Trigger.new){

30

System.debug('Number of DML statements used so far: ' + Limits.getDmlStatements());

31

32

33

for(Opportunity o: opptys){

34

if (o.accountid == a.id)

35

o.description = 'testing';

36

}

37

38

}

39

update opptys;

40

System.debug('Final number of DML statements used so far: ' + Limits.getDmlStatements());

41

System.debug('Final heap size: ' + Limits.getHeapSize());

42

}

43

}

And here is a sample output after running the trigger by updating an account record through the user interface. This was generated in a debug log:

DEBUG|Total Number of SOQL Queries allowed in this Apex code context: 100

DEBUG|Total Number of records that can be queried in this Apex code context: 10000

DEBUG|Total Number of DML statements allowed in this Apex code context: 150

DEBUG|Total Number of CPU usage time (in ms) allowed in this Apex code context: 10000

DEBUG|1. Number of Queries used in this Apex code so far: 1

DEBUG|2. Number of rows queried in this Apex code so far: 0

DEBUG|3. Number of DML statements used so far: 0

DEBUG|4. Amount of CPU time (in ms) used so far: 9

DEBUG|Continue processing. Not going to hit DML governor limits

DEBUG|Going to update 3 opportunities and governor limits will allow 10000

DEBUG|Number of DML statements used so far: 0

DEBUG|Final number of DML statements used so far: 1

DEBUG|Final heap size: 1819

This example illustrates how valuable the Limits Apex class can be when debugging and analyzing the efficiency of your code. It also demonstrates how you can proactively check if you are going to run into governor limits and better handle those scenarios.

Apex Governor Limit Warning Emails

Additionally, you can enable Apex governor limit warning emails.

When an end-user invokes Apex code that surpasses more than 50% of any governor limit, you can specify a user in your organization to receive an email notification of the event with additional details. To enable email warnings:

1

1. Log in to Salesforce as an administrator user.

2

2. Click Setup | Manage Users | Users.

3

3. Click Edit next to the name of the user who should receive the email notifications.

4

4. Select the Send Apex Warning Emails option.

5

5. Click Save.

Best Practice #8: Use @future Appropriately

As articulated throughout this article, it is critical to write your Apex code to efficiently handle bulk or many records at a time. This is also true for asynchronous Apex methods (those annotated with the @future keyword). The differences between synchronous and asynchronous Apex can be found Governors in Apex Code#Synchronous\_vs\_Asynchronous\_Apex. Even though Apex written within an asynchronous method gets its own independent set of higher governor limits, it still has governor limits. Additionally, no more than ten @future methods can be invoked within a single Apex transaction.

Here is a list of governor limits specific to the @future annotation:

• No more than 10 method calls per Apex invocation

• No more than 200 method calls per Salesforce license per 24 hours

• The parameters specified must be primitive dataypes, arrays of primitive datatypes, or collections of primitive datatypes.

• Methods with the future annotation cannot take sObjects or objects as arguments.

• Methods with the future annotation cannot be used in Visualforce controllers in either getMethodName or setMethodName methods, nor in the constructor.

It's important to make sure that the asynchronous methods are invoked in an efficient manner and that the code in the methods is efficient. In the following example, the Apex trigger inefficiently invokes an asynchronous method for each Account record it wants to process:

1

trigger accountAsyncTrigger on Account (after insert, after update) {

2

for(Account a: Trigger.new){

3

// Invoke the @future method for each Account

4

// This is inefficient and will easily exceed the governor limit of

5

// at most 10 @future invocation per Apex transaction

6

asyncApex.processAccount(a.id);

7

}

8

}

Here is the Apex class that defines the @future method:

01

global class asyncApex {

02

03

@future

04

public static void processAccount(Id accountId) {

05

List<Contact> contacts = [select id, salutation, firstname, lastname, email

06

from Contact where accountId = :accountId];

07

08

for(Contact c: contacts){

09

System.debug('Contact Id[' + c.Id + '], FirstName[' + c.firstname + '], LastName[' + c.lastname +']');

10

c.Description=c.salutation + ' ' + c.firstName + ' ' + c.lastname;

11

}

12

update contacts;

13

}

14

}

Since the @future method is invoked within the for loop, it will be called N-times (depending on the number of accounts being processed). So if there are more than ten accounts, this code will throw an exception for exceeding a governor limit of only ten @future invocations per Apex transaction.

Instead, the @future method should be invoked with a batch of records so that it is only invoked once for all records it needs to process:

1

trigger accountAsyncTrigger on Account (after insert, after update) {

2

//By passing the @future method a set of Ids, it only needs to be

3

//invoked once to handle all of the data.

4

asyncApex.processAccount(Trigger.newMap.keySet());

5

}

And now the @future method is designed to receive a set of records:

01

global class asyncApex {

02

03

@future

04

public static void processAccount(Set<Id> accountIds) {

05

List<Contact> contacts = [select id, salutation, firstname, lastname, email from Contact where accountId IN :accountIds];

06

for(Contact c: contacts){

07

System.debug('Contact Id[' + c.Id + '], FirstName[' + c.firstname + '], LastName[' + c.lastname +']');

08

c.Description=c.salutation + ' ' + c.firstName + ' ' + c.lastname;

09

}

10

update contacts;

11

}

12

}

Notice the minor changes to the code to handle a batch of records. It doesn't take a whole lot of code to handle a set of records as compared to a single record, but it's a critical design principle that should persist across all of your Apex code - regardless if it's executing synchronously or asynchronously.

Best Practice #9: Writing Test Methods to Verify Large Datasets

Since Apex code executes in bulk, it is essential to have test scenarios to verify that the Apex being tested is designed to handle large datasets and not just single records. To elaborate, an Apex trigger can be invoked either by a data operation from the user interface or by a data operation from the Force.com SOAP API. The API can send multiple records per batch, leading to the trigger being invoked with several records. Therefore, it is key to have test methods that verify that all Apex code is properly designed to handle larger datasets and that it does not exceed governor limits.

The example below shows you a poorly written trigger that does not handle bulk properly and therefore hits a governor limit. Later, the trigger is revised to properly handle bulk datasets.

Here is the poorly written contact trigger. For each contact, the trigger performs a SOQL query to retrieve the related account. The invalid part of this trigger is that the SOQL query is within the for loop and therefore will throw a governor limit exception if more than 100 contacts are inserted/updated.

01

trigger contactTest on Contact (before insert, before update) {

02

03

for(Contact ct: Trigger.new){

04

05

Account acct = [select id, name from Account where Id=:ct.AccountId];

06

if(acct.BillingState=='CA'){

07

System.debug('found a contact related to an account in california...');

08

ct.email = 'test\_email@testing.com';

09

//Apply more logic here....

10

}

11

}

12

13

}

Here is the test method that tests if this trigger properly handles volume datasets:

01

public class sampleTestMethodCls {

02

03

static testMethod void testAccountTrigger(){

04

05

//First, prepare 200 contacts for the test data

06

Account acct = new Account(name='test account');

07

insert acct;

08

09

Contact[] contactsToCreate = new Contact[]{};

10

for(Integer x=0; x<200;x++){

11

Contact ct = new Contact(AccountId=acct.Id,lastname='test');

12

contactsToCreate.add(ct);

13

}

14

15

//Now insert data causing an contact trigger to fire.

16

Test.startTest();

17

insert contactsToCreate;

18

Test.stopTest();

19

}

20

}

This test method creates an array of 200 contacts and inserts them. The insert will, in turn, cause the trigger to fire. When this test method is executed, a System.Exception will be thrown when it hits a governor limit. Since the trigger shown above executes a SOQL query for each contact in the batch, this test method throws the exception 'Too many SOQL queries: 101'. A trigger can only execute at most 100 queries.

Note the use of Test.startTest and Test.stopTest. When executing tests, code called before Test.startTest and after Test.stopTest receive a separate set of governor limits than the code called between Test.startTest and Test.stopTest. This allows for any data that needs to be setup to do so without affecting the governor limits available to the actual code being tested.

Now let's correct the trigger to properly handle bulk operations. The key to fixing this trigger is to get the SOQL query outside the for loop and only do one SOQL Query:

01

trigger contactTest on Contact (before insert, before update) {

02

03

Set<Id> accountIds = new Set<Id>();

04

for(Contact ct: Trigger.new)

05

accountIds.add(ct.AccountId);

06

07

//Do SOQL Query

08

Map<Id, Account> accounts = new Map<Id, Account>(

09

[select id, name, billingState from Account where id in :accountIds]);

10

11

for(Contact ct: Trigger.new){

12

if(accounts.get(ct.AccountId).BillingState=='CA'){

13

System.debug('found a contact related to an account in california...');

14

ct.email = 'test\_email@testing.com';

15

//Apply more logic here....

16

}

17

}

18

19

}

Note how the SOQL query retrieving the accounts is now done once only. If you re-run the test method shown above, it will now execute successfully with no errors and 100% code coverage.

Best Practices #10: Avoid Hardcoding IDs

When deploying Apex code between sandbox and production environments, or installing Force.com AppExchange packages, it is essential to avoid hardcoding IDs in the Apex code. By doing so, if the record IDs change between environments, the logic can dynamically identify the proper data to operate against and not fail.

Here is a sample that hardcodes the record type IDs that are used in an conditional statement. This will work fine in the specific environment in which the code was developed, but if this code were to be installed in a separate org (ie. as part of an AppExchange package), there is no guarantee that the record type identifiers will be the same.

01

for(Account a: Trigger.new){

02

03

//Error - hardcoded the record type id

04

if(a.RecordTypeId=='012500000009WAr'){

05

//do some logic here.....

06

}else if(a.RecordTypeId=='0123000000095Km'){

07

//do some logic here for a different record type...

08

}

09

10

}

Now, to properly handle the dynamic nature of the record type IDs, the following example queries for the record types in the code, stores the dataset in a map collection for easy retrieval, and ultimately avoids any hardcoding.

01

//Query for the Account record types

02

List<RecordType> rtypes = [Select Name, Id From RecordType

03

where sObjectType='Account' and isActive=true];

04

05

//Create a map between the Record Type Name and Id for easy retrieval

06

Map<String,String> accountRecordTypes = new Map<String,String>{};

07

for(RecordType rt: rtypes)

08

accountRecordTypes.put(rt.Name,rt.Id);

09

10

for(Account a: Trigger.new){

11

12

//Use the Map collection to dynamically retrieve the Record Type Id

13

//Avoid hardcoding Ids in the Apex code

14

if(a.RecordTypeId==accountRecordTypes.get('Healthcare')){

15

//do some logic here.....

16

}else if(a.RecordTypeId==accountRecordTypes.get('High Tech')){

17

//do some logic here for a different record type...

18

}

19

20

}

By ensuring no IDs are stored in the Apex code, you are making the code much more dynamic and flexible - and ensuring that it can be deployed safely to different environments.