**Running Apex Within Governor Execution Limits:**

Developing software in a multitenant cloud environment, the Force.com platform, relieves you from having to scale your code because the Force.com platform does it for you. Because resources are shared in a multitenant platform, the Apex runtime engine enforces a set of governor execution limits to ensure that no one transaction monopolizes shared resources. Your Apex code must execute within these predefined execution limits. If a governor limit is exceeded, a run-time exception that can’t be handled is thrown. By following best practices in your code, you can avoid hitting these limits. The governor execution limits are per transaction.

The following are some best practices for writing code that doesn’t exceed certain governor limits:

**Bulkifying DML Calls:** Making DML calls on lists of sObjects instead of each individual sObject makes it less likely to reach the DML statements limit.

**More Efficient SOQL Queries:** Placing SOQL queries inside for loop blocks isn’t a good practice because the SOQL query executes once for each iteration and may surpass the 100 SOQL queries limit per transaction.

**SOQL For Loops:** Use SOQL for loops to operate on records in batches of 200. This helps avoid the heap size limit of 6 MB. This limit is for code running synchronously and it is higher for asynchronous code execution.

**Scheduled Execution of Apex**

The Apex Scheduler lets you have Apex classes run at specified times. This is ideal for daily or weekly maintenance tasks. To take advantage of the scheduler, you need to write an Apex class that implements the Schedulable interface, and then schedule it for execution on a specific schedule.

Example:

global class MySchedulableClass implements Schedulable {

global void execute(SchedulableContext ctx) {

CronTrigger ct = [SELECT Id, CronExpression, TimesTriggered, NextFireTime FROM CronTrigger WHERE Id = :ctx.getTriggerId()];

System.debug(ct.CronExpression);

System.debug(ct.TimesTriggered);

Merchandise\_\_c m = new Merchandise\_\_c( Name='Scheduled Job Item', Description\_\_c='Created by the scheduler', Price\_\_c=1, Total\_Inventory\_\_c=1000);

insert m;

}

}

The declaration of the class contains an extra implements Schedulable at the end. This indicates that the class implements the Schedulable interface and must implement the only method that this interface contains, which is this execute method.

The parameter of this method is a SchedulableContext object. It provides the getTriggerId method that returns the ID of the CronTrigger API object. After a class has been scheduled, a CronTrigger object is created that represents the scheduled job.

The CronTrigger object is queried to get additional information about the scheduled job. The Cron expression and the number of times the job has been run already is written to the debug log.

Finally, the execute method creates a merchandise record.

**Adding a Test for the Schedulable Class:**

@isTest

private class TestSchedulableClass {

// CRON expression: midnight on March 15. Because this is a test, job executes immediately after Test.stopTest().

public static String CRON\_EXP = '0 0 0 15 3 ? 2022';

static testmethod void test() {

Test.startTest();

String jobId = System.schedule('ScheduleApexClassTest', CRON\_EXP, new MySchedulableClass()); CronTrigger ct = [SELECT Id, CronExpression, TimesTriggered, NextFireTime FROM CronTrigger WHERE id = :jobId];

System.assertEquals(CRON\_EXP, ct.CronExpression);

// Verify the job has not run

System.assertEquals(0, ct.TimesTriggered);

// Verify the next time the job will run System.assertEquals('2022-03-15 00:00:00', String.valueOf(ct.NextFireTime));

// Verify the scheduled job hasn't run yet.

Merchandise\_\_c[] ml = [SELECT Id FROM Merchandise\_\_c WHERE Name = 'Scheduled Job Item']; System.assertEquals(ml.size(),0);

Test.stopTest();

// Now that the scheduled job has executed after Test.stopTest(),fetch the new merchandise that got added.

ml = [SELECT Id FROM Merchandise\_\_c WHERE Name = 'Scheduled Job Item']; System.assertEquals(ml.size(), 1);

}

}

The test method schedules the MySchedulableClass class by calling the System.schedule method. The System.Schedule method takes three arguments: a name for the job, an expression used to represent the time and date the job is scheduled to run, and the name of the class. The System.schedule method uses the user's timezone for the basis of all schedules.

The call to System.schedule is included within the Test.startTest and Test.stopTest block. This ensures that the job gets executed after the Test.stopTest call regardless of the schedule specified in the cron expression. Any asynchronous code included within Test.startTest and Test.stopTest gets executed synchronously after Test.stopTest.

Finally, the test method verifies a new merchandise item got added by the scheduled class.

**Apex Batch Processing:**

Using batch Apex classes, you can process records in batches asynchronously. If you have a large number of records to process, for example, for data cleansing or archiving, batch Apex is your solution. Each invocation of a batch class results in a job being placed on the Apex job queue for execution.

The execution logic of the batch class is called once for each batch of records. The default batch size is 200 records. You can also specify a custom batch size. Furthermore, each batch execution is considered a discrete transaction. With each new batch of records, a new set of governor limits is in effect. In this way, it’s easier to ensure that your code stays within the governor execution limits. Another benefit of discrete batch transactions is to allow for partial processing of a batch of records in case one batch fails to process successfully, all other batch transactions aren’t affected and aren’t rolled back if they were processed successfully.

**Batch Apex Syntax:**

To write a batch Apex class, class must implement the Database.Batchable interface. Class declaration must include the implements keyword followed by Database.Batchable.

global class CleanUpRecords implements Database.Batchable {

code

}

It has three methods

Start method: The start method is called at the beginning of a batch Apex job. It collects the records or objects to be passed to the interface method execute.

Execute Method: The execute method is called for each batch of records passed to the method. Use this method to do all required processing for each chunk of data.

Finish method: The finish method is called after all batches are processed. Use this method to send confirmation emails or execute post-processing operations.