Apex Code Quality Rules

Rule #01: Avoid DML statements in loops

Avoid DML statements inside loops to avoid hitting the DML governor limit. Instead, try to batch up the data into a list and invoke your DML once on that list of data outside the loop.

Database class methods, DML operations, SOQL queries, SOSL queries, Approval class methods, Email sending, async scheduling, or queueing within loops can cause governor limit exceptions. Instead, try to batch up the data into a list and invoke the operation once on that list of data outside the loop.

Rule #02: Avoid SOQL in loops

SOQL calls within loops can cause governor limit exceptions. Avoid using SOQL query in the loop, instead use collections like Map, List, and Set.

Rule#03: Avoid SOSL in loops

SOSL calls within loops can cause governor limit exceptions. Avoid using SOsL query in the loop, instead use collections like Map, List, and Set.

Rule#04: Apex unit test class should have asserts

Apex unit tests should include at least one assertion. This makes the tests more robust, and using assert with messages provides the developer a clearer idea of what the test does.

Rule#05: Avoid logic in the trigger

As triggers do not allow methods like regular classes they are less flexible and suited to apply a good encapsulation style. Therefore delegate the triggers to work to a regular class (often called Trigger handler class).

To avoid putting your logic in a trigger, use a single trigger per object template and write a handler and helper class. See the screenshots below for an AccounTrigger example.

```
/**
 * @description AccountTrigger.
 * @Created By: Arun Kumar
 * @Last Modified Date : 05-04-2022
*/
trigger AccountTrigger on Account (before insert,before update, before delete, after insert, after update, after delete,after undelete) {

        switch on Trigger.operationType {
            when BEFORE_INSERT {
                //call before insert handler method
            }
            when BEFORE_UPDATE {
                //call before update handler method
            }
            when BEFORE_DELETE{
                //call before delete handler method
            }
            when AFTER_INSERT{
                //call after insert handler method
            }
            when AFTER_UPDATE{
                //call after update handler method
            }
            when AFTER_DELETE{
                //call after delete handler method
            }
            when AFTER_UNDELETE{
                //call after undelete handler method
            }
        }
}
```

```
public interface TriggerHandler {

    // ********** before methods ******************
    void beforeInsert(List<sObject> newRecords);
    void beforeUpdate(List<sObject> newRecords, List<sObject> oldRecords, Map<Id,sObject> newRecordMap, Map<Id,sObject> oldRecordMap);
    void beforeDelete(List<sObject> oldRecords, Map<Id,sObject> oldRecordMap);

    // ********** after methods ******************
    void afterInsert(List<sObject> newRecords, Map<Id,sObject> newRecordMap);
    void afterUpdate(List<sObject> newRecords, List<sObject> oldRecords, Map<Id,sObject> newRecordMap, Map<Id,sObject> oldRecordMap);
    void afterDelete(List<sObject> oldRecords, Map<Id, sObject> oldRecordMap);
    void afterUndelete(List<sObject> newRecords, Map<Id,sObject> newRecordMap);

}
```

```apex
public without sharing class AccountTriggerHandler implements TriggerHandler{
    private boolean triggerIsExecuting;
    private integer triggerSize;
    public AccountTriggerHelper helper;
    public AccountTriggerHandler(Boolean triggerIsExecuting, integer triggerSize) {
        this.triggerIsExecuting = triggerIsExecuting;
        this.triggerSize = triggerSize;
        this.helper = new AccountTriggerHelper();
    }

    public void beforeInsert(List<Account> newAccounts){
        //call AccountTriggerHelper methods
    }
    public void beforeUpdate(List<Account> newAccounts, List<Account> oldAccopunts, Map<Id,sObject> newAccountMap, Map<Id,sObject> oldAccountMap
      //call AccountTriggerHelper methods
    }

    public void beforeDelete(List<Account> oldAccounts, Map<Id,SObject> oldAccountMap){
        //call AccountTriggerHelper methods
    }

    public void afterInsert(List<Account> newAccounts, Map<Id,SObject> newAccountMap){
        //call AccountTriggerHelper methods

    }
    public void afterUpdate(List<Account> newAccounts, List<Account> oldAccounts, Map<Id,SObject> newAccountMap, Map<Id,SObject> oldAccountMap){
        //call AccountTriggerHelper methods
    }

    public void afterDelete(List<Account> oldAccounts, Map<Id,SObject> oldAccountMap){
        //call AccountTriggerHelper methods
    }

    public void afterUndelete(List<Account> newAccounts, Map<Id,SObject> newAccountMap){
        //call AccountTriggerHelper methods
    }
}
```

```apex
public without sharing class AccountTriggerHelper {
    //Constructor
    public AccountTriggerHelper() {
        system.debug('Inside AccounttriggerHelper Constructor');
    }
    /**
     * @description updateAccountOwner method updates accounts owner records.
     * @param accList
    */
    public void updateAccountOwner(List<Account> accList){
        //write logic to update the account owner
    }
    /**
     * @description updateAddresses method updates contact address if account address is changed,
     * if more than record found with same zipcode then it gets random record.
     * @param accList
    */
    public void updateContactAddress(List<Account> accList) {
        //write update contacts logic below
    }
    //Write other methods here as per your requirements

    /********************* ADD MORE METHODS************************/
}
```

Rule#06: Debugs should use logging level

The first parameter of System.debug, when using the signature with two parameters, is a LoggingLevel enum.

Having the Logging Level specified provides a cleaner log, and improves the readability.

Rule#07: Apex unit test should not use SeeAllData True

Apex unit tests should not use @isTest(seeAllData=true) because it opens up the existing database data for unexpected modification by tests.

Rule#08: Apex unit test method should have @IsTest annotation

Apex test methods should have @isTest annotation instead of the testMethod keyword, as testMethod is deprecated. Salesforce advises to use @isTest annotation for test classes and methods.

Rule#9: Class naming conventions

This rule reports type declarations that do not match the regex that applies to their specific kind (e.g. enum or interface).

By default, this rule uses the standard Apex naming convention (Pascal case).

```
public class FooClass { } // This is in pascal case, so it's ok
```

```
public class fooClass { } // This will be reported an an issue
```

REGEX: [A-Z][a-zA-Z0-9_]*

Rule#10: Field naming conventions

This rule reports variable declarations which do not match the regex that applies to their specific kind —e.g. constants (static final), static field, final field.

By default, this rule uses the standard Apex naming convention (Camel case).

```
public class Foo {
   Integer instanceField; // This is in camel case, so it's ok
   Integer INSTANCE_FIELD; // This will be reported as an issue
}
```

Rule#11: For loops must use braces

Avoid using ‘for’ statements without using surrounding braces. If the code formatting or indentation is lost then it becomes difficult to separate the code being controlled from the rest.

```
for (int i = 0; i < 42; i++) // not recommended
   foo();
for (int i = 0; i < 42; i++) { // preferred approach
   foo();
}
```

Rule#12: While loops must use braces

Avoid using ‘while’ statements without using braces to surround the code block. If the code formatting or indentation is lost then it becomes difficult to separate the code being controlled from the rest.

```
while (true)    // not recommended
   x++;

while (true) {  // preferred approach
   x++;
}
```

Rule#13: Variable naming conventions

This rule reports variable declarations which do not match the regex that applies to their specific kind (e.g. local variable, or final local variable)

Rule#14: Local variable naming conventions

This rule reports variable declarations which do not match the regex that applies to their specific kind (e.g. local variable, or final local variable)

Camel case

```
public class Foo {
    Integer instanceField; // This is in camel case, so it's ok
    Integer INSTANCE_FIELD; // NOT OK
}
```

Rule#15: Method naming conventions

"This rule reports method declarations which do not match the regex that applies to their specific kind (e.g. static method, or test method)

```
public class Foo {

   public void instanceMethod() { } // This is in camel case, so it's ok


   public void INSTANCE_METHOD() { } // This will be reported as an issue

}
```

REGEX: [a-z][a-zA-Z0-9]*"

Rule#16: If-Else statements must use braces

Avoid using if..else statements without using surrounding braces. If the code formatting or indentation is lost then it becomes difficult to separate the code being controlled from the rest.

Rule#17: If Statements must use braces

Avoid using if statements without using braces to surround the code block. If the code formatting or indentation is lost then it becomes difficult to separate the code being controlled from the rest.

Rule#18: Apex doc

- ★ This rule validates that:
- ★ ApexDoc comments are present for classes, methods, and properties that are public or global, excluding overrides and test classes (as well as the contents of test classes).
- ★ ApexDoc comments are present for classes, methods, and properties that are protected or private, depending on the properties reportPrivate and reportProtected.
- ★ ApexDoc comments should contain @description.
- ★ ApexDoc comments on non-void, non-constructor methods should contain @return.
- ★ ApexDoc comments on void or constructor methods should not contain @return.
- ★ ApexDoc comments on methods with parameters should contain @param for each parameter, in the same order as the method signature.

Rule#19: Avoid hardcoding Id

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.

```
// KO
public static void foo() {

        if (opp.RecordTypeId == '016500000005WAr') {
            //do some logic here.....
        } else if (opp.RecordTypeId == '016500000008WAr') {
            //do some logic here for a different record type...
        }
    }
```

If you run the above code in different environments, it will fail. The code snippet below is the best way to implement this type of requirement.

```
// OK
public static void foo() {

        if (opp.RecordTypeId ==
Schema.sObjectType.Opportunity.getRecordTypeInfosByName().get('Local') ) {
            //do some logic here.....
        } else if (opp.RecordTypeId ==
Schema.sObjectType.Opportunity.getRecordTypeInfosByName().get('Regional')) {
            //do some logic here for a different record type...
        }
}
```

Rule#20: Empty If Statement

Empty If Statement finds instances where a condition is checked but nothing is done about it.

Rule#21: Apex sharing violations

Detect classes declared without explicit sharing mode if DML methods are used. This forces the developer to take access restrictions into account before modifying objects.

Rule#22: Apex SOQL injection

Detects the usage of untrusted / unescaped variables in DML queries.

ex. SELECT Id FROM Contact WHERE (IsDeleted = false AND Name LIKE '%test%') OR (Name LIKE '%')

the results show all contacts, not just the non-deleted ones. A SOQL Injection flaw can be used to modify the intended logic of any vulnerable query.

To prevent a SOQL injection attack, avoid using dynamic SOQL queries. Instead, use static queries and binding variables. The vulnerable example above can be re-written using static SOQL as follows:

```
public class SOQLController {
    public String name {
        get { return name;}
        set { name = value;}
            }
    public PageReference query() {
        String queryName = '%' + name + '%';
        queryResult = [SELECT Id FROM Contact WHERE
          (IsDeleted = false and Name like :queryName)];
        return null;
            }
}
```

If you must use dynamic SOQL, use the escapeSingleQuotes method to sanitize user-supplied input. This method adds the escape character (\) to all single quotation marks in a string that is passed in from a user. The method ensures that all single quotation marks are treated as enclosing strings, instead of database commands."