How to Detect and Stop Infinite Automation Loops Before They Hit Limits

In the world of Salesforce automation, few things are more frustrating than hitting governor limits because of an infinite loop you didnt see coming. These loops can freeze your org, consume all your limits, and create records until storage is full. The worst part is they often slip into production because theyre tricky to catch during testing. In this blog we are going to see how to detect and prevent these loops before they cause damage, with real code examples you can apply immediately.

Analysing the root cause for loops

An infinite loop in Salesforce automation happens when a trigger, workflow, process builder, or flow causes itself to fire repeatedly without an exit condition. The classic example is a trigger that updates a record, which then fires the same trigger again on the updated record, creating a cycle that never ends. Salesforce has built-in circuit breakers—like the maximum trigger depth exceeded error but by the time you hit those, youve already consumed significant resources and potentially created data chaos. The goal is to stop the loop before it starts, or at least catch it early.

The Static Variable Guard Pattern

The most effective and widely used method to prevent trigger recursion is the static variable guard. This technique uses a constant variable in an Apex class to check if a trigger has already run within the same transaction. These variables in Apex maintain their value across trigger executions within a single transaction, but does not involve in different transactions, which makes them perfect for tracking recursion.

Now lets see a solid example. Imagine we have an Account trigger which automatically updates a custom field on related Contacts whenever the Accounts phone number field is updated .Without protection, if something else causes the Account to update again within the same transaction,this causes process to enter a loop.

First, we create a utility class to hold our static guard variable.

```
public class TriggerGuard {
    public static Boolean isAccountTriggerExecuting = false;
}
```

Now, lets implement our trigger using this guard.

```
trigger AccountPhoneUpdate on Account (after update) {
    // Check if were already in a trigger execution
    if (TriggerGuard.isAccountTriggerExecuting) {
        // If true, were in a recursive call. Exit immediately.
        System.debug(Prevented recursive Account trigger execution.);
        return;
    }
```

```
    // Set the guard to true for this transaction
    TriggerGuard.isAccountTriggerExecuting = true;

    try {
        List<Contact> contactsToUpdate = new List<Contact>();

        // Find related contacts for accounts where phone changed
        for (Account acc : Trigger.new) {
            Account oldAccount = Trigger.oldMap.get(acc.Id);
            if (acc.Phone != oldAccount.Phone) {
                // In a real scenario, youd query for contacts here
                // For this example, well simulate an update
                // This is where recursion risk exists if something
                // subsequently updates the Account again
            }
        }

        if (!contactsToUpdate.isEmpty()) {
            update contactsToUpdate;
        }

    } catch (Exception e) {
        // Always handle exceptions
        System.debug(Error in Account trigger:  + e.getMessage());
    } finally {
        // Consider whether to reset the guard here
        // For simple triggers, you might reset it. For complex chains, be careful.
        // TriggerGuard.isAccountTriggerExecuting = false;
    }
}
```

The important decision point whether to set guard to true or false is in the finally block. In this concise example, resetting the guard to false seems logical, but it may lead to another risk if you have other automation based on 1st automation it might cause loop again. commonly, keeping it true for the entire transaction is safer approach. This ensures the trigger cannot run a second time on any record in that transaction, which is usually what you want.

Enhanced Guard by Counting, Not Just Boolean

By using the above guard we would be able to run it only once as the guard will be set true after 1st iteration. For more complex scenarios where you need to allow a controlled number of executions (but not unlimited), use a counter instead of a boolean.

```
public class TriggerGuard {
    public static Integer accountTriggerCount = 0;
    public static final Integer MAX_EXECUTIONS = 3;
}
```

In the trigger, you would then check the count.

```
if (TriggerGuard.accountTriggerCount >= TriggerGuard.MAX_EXECUTIONS) {
   System.debug(Stopping potential infinite loop after  + TriggerGuard.MAX_EXECUTIONS +  executions.);
   return;
}
TriggerGuard.accountTriggerCount++;
```

This allows up to three executions per transaction, which can be useful for complex data operations that require multiple passes, but stops a true infinite loop.

The Per-Record Guard When You Need More Precision

Sometimes, you need to prevent re-processing the same record, but still allow the trigger to run for other records in the collection. For this, use a static Set<Id>.

```
public class TriggerGuard {
   public static Set<Id> processedAccountIds = new Set<Id>();
}
```

In the trigger, check and add each records ID.

```
List<Account> accountsToProcess = new List<Account>();
for (Account acc : Trigger.new) {
   if (!TriggerGuard.processedAccountIds.contains(acc.Id)) {
      accountsToProcess.add(acc);
      TriggerGuard.processedAccountIds.add(acc.Id);
   }
}
// Now process only accounts in accountsToProcess
```

This is particularly useful for before triggers or scenarios where you are modifying the same object that fired the trigger, but only want to apply logic once per record.

Handling Flows and Process Builder

Apex triggers arent the only source of loops. Declarative tools can create loops just as easily. Imagine a flow on Contact that updates a field on its parent Account, and a separate flow on Account that, when that field changes, updates a Contact field. This creates a cross-object loop.

The solution here is design discipline. First, map out your automation. For any field update, check if this update cause the same process, or another process that touches this record, to fire again? Second, use the Do you want to execute the actions only when specified changes are made to the record? Like entry condition in flow. This prevents the process from running unless your chosen fields are actually updated, reducing loop risk. You can also use a custom setting as store guard details like true or false and use it for execution like is_running .

Testing loops in Apex using @isTest annotation

You must test for potential loops. Write unit tests that simulate the worst case scenario.

```
@isTest
static void testPreventInfiniteLoop() {
    // Setup test data
    Account testAccount = new Account(Name=Test, Phone=555-1000);
    insert testAccount;

    // Simulate multiple update cycles in one transaction
    Test.startTest();
    // This update should trigger our code once
    testAccount.Phone = 555-1001;
    update testAccount;

    // Attempt a second update within same transaction context
    // The guard should prevent the trigger logic from running again
    testAccount.Phone = 555-1002;
    try {
        update testAccount;
        // If we get here, the guard worked and no exception was thrown
        System.assert(true, Guard prevented recursion.);
    } catch (Exception e) {
        // If recursion wasnt controlled, we might hit limits
        System.assert(false, Unexpected exception:  + e.getMessage());
    }
    Test.stopTest();

    // Verify expected outcomes
    Account updatedAccount = [SELECT Phone FROM Account WHERE Id = :testAccount.Id];
    System.assertEquals(555-1002, updatedAccount.Phone);
}
```

This test validates that our guard static variable prevents the trigger code from executing multiple times, even with multiple DML operations.

Architecture Decisions to Avoid Loops

Beyond code guards, consider your design. The Trigger Handler pattern is essential. Organize your trigger logic into a single, well-structured class per object. This gives you one place to manage execution order and guards.

```
public with sharing class AccountTriggerHandler {
    public static Boolean hasRun = false;

    public void onAfterUpdate(List<Account> newList, Map<Id, Account> oldMap) {
        if (hasRun) return;
```

```
        hasRun = true;

        // Your business logic here
        updateRelatedContacts(newList, oldMap);
    }

    private void updateRelatedContacts(List<Account> newList, Map<Id, Account> oldMap)
{
        // Specific logic isolated here
    }
}
```

Your trigger then becomes minimal.

```
trigger AccountTrigger on Account (after update) {
    AccountTriggerHandler handler = new AccountTriggerHandler();
    handler.onAfterUpdate(Trigger.new, Trigger.oldMap);
}
```

This structure makes it obvious where to place your guard logic and keeps everything maintainable.

Monitoring for Breakouts

Despite precautions, loops can still occur in complex orgs. Implement proactive monitoring. Create a custom object Loop_Error_Log and in your guard class, when you detect a potential loop (like hitting your max execution count), log the details: object type, record IDs, timestamp, and user. This creates a trackable record instead of just a debug log.

Schedule a weekly report on these logs. If you see a spike, investigate. Combine this with Salesforces standard Apex Limits email alerts, set to notify you when usage exceeds 80% of any governor limit. This gives you an early warning system.

The Human Factor Code Reviews and Documentation

Finally, the best technical guard can be defeated by poor communication. Implement mandatory code reviews for all automation, declarative and coded. The reviewers checklist must include: Is there recursion protection? Document your automation.

Infinite loops can cause high impact to Salesforce org stability, but they can be guarded. By configuring static variable guards, designing automation with awareness, continous testing, and establishing monitoring, you can stop loops even before they start. The key is to accept the repetetive nature of the platform and build your automation to be self-aware. Start by adding a simple guard class to your org today.