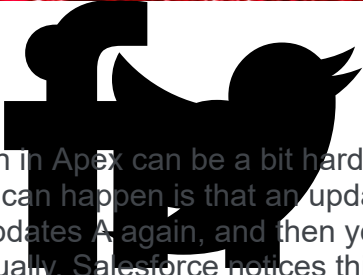
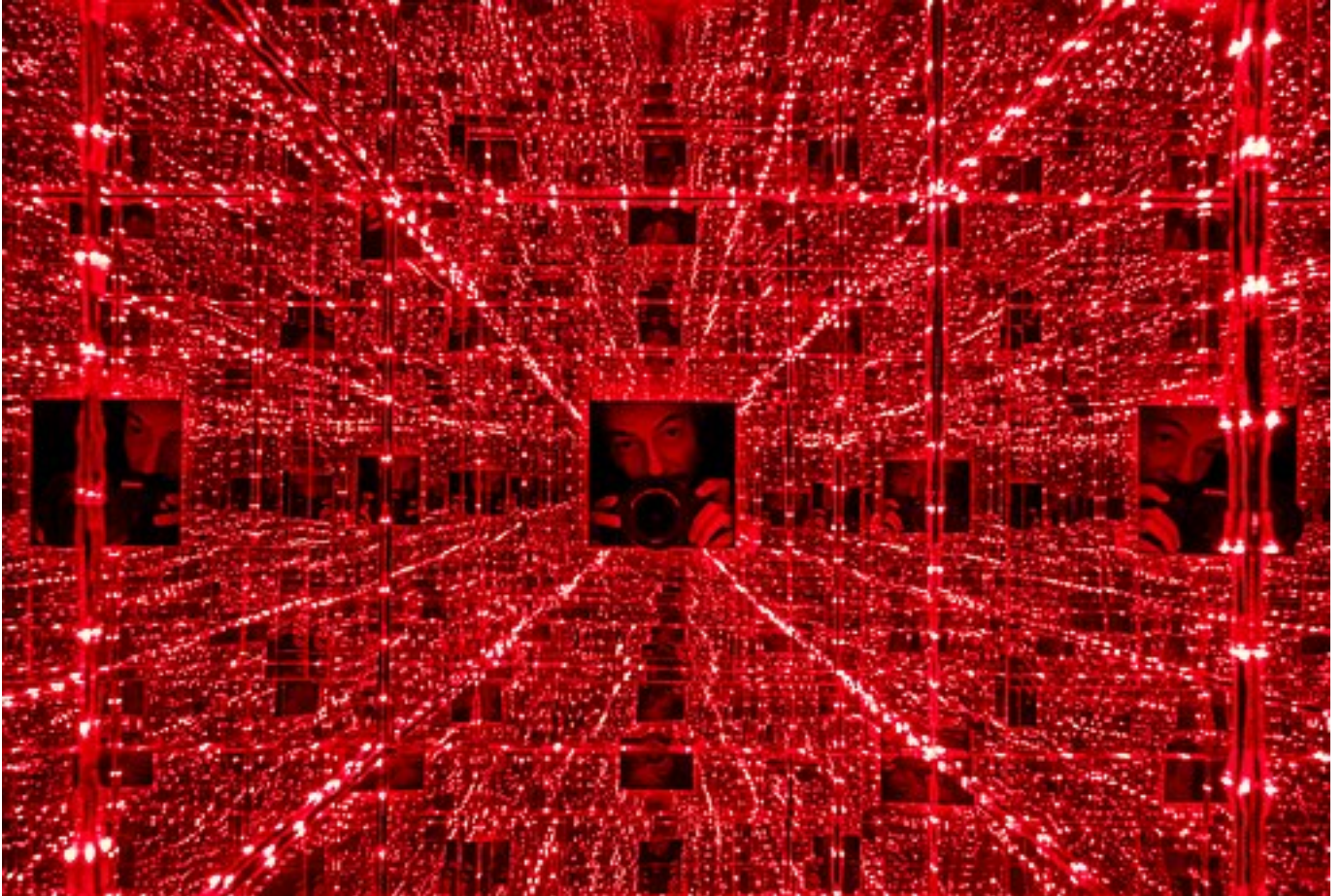


Please don't use static flags to control Apex Trigger recursion!



by Aidan Harding
February 01, 2018



Trigger recursion in Apex can be a bit hard to get your head around. Often it is indirect, so hard to diagnose. What can happen is that an update is made on object A, then a trigger on A updates B, then a trigger on B updates A again, and then you can end up with an infinite loop of trigger recursion. Eventually, Salesforce notices this and throws an error with “maximum trigger depth exceeded”.

There can be a very complex path that goes from A round the houses and back to A again. It could be a mix of Workflow, Process Builder, Flow, and Apex. It could go via a managed package. So, this can be an intimidating problem to fix.

However, I frequently see the incorrect answer being provided on developer forums and sometimes in code we have inherited from elsewhere. The incorrect approach is to have a static flag somewhere which we check to make sure that the trigger only runs once. The aim of this post is to explain why that doesn't work, and to propose a couple of alternatives that do work.

To illustrate the point, we will use a somewhat fake scenario where the trigger recursion is just within a single trigger. Suppose we want to have a field on Lead called “Last Modified Date” which is of

type Date , rather than the standard DateTime field. Further, suppose we don't want to use a formula field for this (which you absolutely would use, in the real world). Even more crazily, suppose we don't want to do this in a before trigger, we want to do it in an after trigger. This sets up the potential recursion, we trigger on insert + update of Lead, writing back to the "Last Modified Date" field. Which will call the trigger again. The trick is to have some termination condition for this trigger.

The trigger on Lead just delegates to a helper class (using MyTriggerNameHelper to hold all the logic is still not a great idea, but let's ignore that for now):

```
trigger LeadSetLastModifiedDate on Lead (after insert, after update) {  
    LeadSetLastModifiedDateHelper.handleTrigger(Trigger.new);  
}
```

And a test class which sets up the data, showing via its assertions, what we hope to happen:

```
@isTest  
private class LeadSetLastModifiedDateTest {  
  
    private static List<Lead> insertedLeads;  
  
    private static void insertLeads() {  
        insertedLeads = new List<Lead>();  
  
        for(Integer i=0; i < 400; i++) {  
            insertedLeads.add(new Lead(LastName = ''+ i, Company = '' + i));  
        }  
  
        insert insertedLeads;  
    }  
  
    private static void checkLeads() {  
        Map<Id, Lead> afterInsert = new Map<Id, Lead>([SELECT Id, LastModifiedDate  
  
        for(Integer i=0; i < afterInsert.size(); i++) {  
            Lead l = afterInsert.get(insertedLeads[i].Id);  
            System.assertEquals(l.LastModifiedDate.date(), l.Last_Modified_Date__  
        }  
    }  
  
    @isTest private static void noRecursionHandling() {  
        LeadSetLastModifiedDateHelper.recursionControlMethod = LeadSetLastModifie  
  
        insertLeads();  
        checkLeads();  
    }  
  
    @isTest private static void staticFlagRecursionHandling() {  
        LeadSetLastModifiedDateHelper.recursionControlMethod = LeadSetLastModifie  
  
        insertLeads();  
        checkLeads();  
    }  
  
    @isTest private static void staticSetRecursionHandling() {  
        LeadSetLastModifiedDateHelper.recursionControlMethod = LeadSetLastModifie  
  
        insertLeads();  
        checkLeads();  
    }  
}
```

```

    }
    @isTest private static void checkRecordRecursionHandling() {
        LeadSetLastModifiedDateHelper.recursionControlMethod = LeadSetLastModifiedDateHelper.RECURSION_CONTROL_METHOD_INSERT;

        insertLeads();
        checkLeads();
    }
}

```

In these tests, we create 400 leads (the large number matters, we'll see that later), set a flag on the trigger handler class to choose the method of handling trigger recursion, then assert that all 400 have their `Last_Modified_Date__c` field set to the right value.

1. No recursion handling at all

```

List<Lead> toUpdate = new List<Lead>();

for(Lead l : newLeads) {
    toUpdate.add(new Lead(Id = l.Id, Last_Modified_Date__c = l.lastModifiedDate.d
}
update toUpdate;

```

This is the dumbest handler of all. Loop through all the Leads that have changed, set their `Last_Modified_Date__c` field, then call `update`. It fails with “maximum trigger depth exceeded”.

2. Using a static flag

```

private static Boolean alreadyRunOnce = false;

/* ... */

if(!alreadyRunOnce) {
    alreadyRunOnce = true;
    List<Lead> toUpdate = new List<Lead>();

    for(Lead l : newLeads) {
        toUpdate.add(new Lead(Id = l.Id, Last_Modified_Date__c = l.lastModifiedDate.d
    }
    update toUpdate;
}

```

This is the method often suggested on the developer forums, but **it does not work**. The main problem with this approach is that Salesforce really runs triggers on a maximum of 200 records at-a-time. So, if we insert 400 Leads, SF runs the trigger twice. Once on the first 200, then again for the second 200. Our static flag keeps its value through those invocations, so you now won't be surprised to hear that this test fails with “Assertion Failed: Failed on row 200: Expected: 2018-02-01 00:00:00, Actual: null”.

3. Using a static set

One way we could improve the previous method is to use a set of Ids to see which ones we have processed already, only skipping records if we have already processed them in this transaction:

```

private static Set<Id> alreadyProcessed = new Set<Id>();

```

```

/* ... */

List<Lead> toUpdate = new List<Lead>();

for(Lead l : newLeads) {

    if(!alreadyProcessed.contains(l.Id)) {
        toUpdate.add(new Lead(Id = l.Id, Last_Modified_Date__c = l.lastModifiedDa
        alreadyProcessed.add(l.Id);
    }
}

update toUpdate;

```

This passes the test! It's not yet my preferred way, but it does the job in this case.

4. Check whether the trigger actually needs to do anything

This is my preferred method. If we were writing a recursive function, then we would have some conditional in the function, checking whether the current state means that we need to recurse again, or whether we can stop now. This approach is that, but spread across trigger invocations:

```

List<Lead> toUpdate = new List<Lead>();

for(Lead l : newLeads) {
    Date nextLastModifiedDate = l.lastModifiedDate.date();
    if(l.Last_Modified_Date__c != nextLastModifiedDate) {
        toUpdate.add(new Lead(Id = l.Id, Last_Modified_Date__c = nextLastModified
    }
}

update toUpdate;

```

This also passes the test! This is my preferred method. Not only does it prevent recursion, but it also stops your code from doing an unnecessary update if the Lead is updated more than once in the same day.

Conclusions

Don't ever use static flags to control recursion. It is asking for data inconsistencies when the code appears to work for months, then starts to go wrong when someone starts using Data Loader.

My preference is to check the record before acting on it. If that's not possible, then using a static set of Ids can be a suitable alternative.

And of course, try to just avoid trigger recursion at the design stage, unless you **really, really** need it.