Triggers – workflow – recursion control – callouts – allOrNone

An insidious set of circumstances:

Starting condition:

- You have an afterUpdate trigger that if a condition when oldvalue changes to newValue, you want to do a callout
- You also have a workflow that updates the same object and that workflow's entry criteria is satisfied when the for the same update event

Now, you might not know this, but Workflow field updates will cause the trigger to re-execute AND, *the value of Trigger.old will be as it was when the record was initially updated*. Here's what it says from the well-thumbed Triggers and Order of Execution Apex Doc:

> *Trigger.old contains a version of the objects before the specific update that fired the trigger. However, there is an exception. When a record is updated and subsequently triggers a workflow rule field update, Trigger.old in the last update trigger won't contain the version of the object immediately prior to the workflow update, but the object before the initial update was made*

Thus, your future method will be called TWICE, and, if it is doing a callout, will callout twice. Here's a simple proof:

Apex trigger

```
trigger LeadTrigger on Lead (before insert, before update, after insert, after update) {

    if (Trigger.isAfter && Trigger.isUpdate)
        new LeadTriggerHandler().onAfterUpdate(Trigger.new,Trigger.oldMap);
}
```

Apex Trigger handler

```
public class LeadTriggerHandler {

    public void onAfterUpdate(Lead[] leads, map<ID,Lead> oldLeads) {
```

```
        for (Lead l: leads) {
            Lead oldLead = oldLeads.get(l.Id);
            if (l.Company != oldLead.Company) {
                System.debug(LoggingLevel.INFO,'company has changed from ' + oldLead.Company   +
                             'to ' + l.Company + ' .. request an @future to dowork');
                doCallout(l.Company);
            }
        }
    }

    @future
    static void doCallout(String company) {
        System.debug(LoggingLevel.INFO,'future method to do callout for ' + company);
        // .. callout details not important
    }
}
```

Workflow

- **Evaluation Criteria:** Evaluate the rule when a record is created, and any time it's edited to subsequently meet criteria
- **Rule Criteria:** If Lead.Company contains 'Changed'
- **Action:** Field update Lead.Mobile to '650-555-1212'

Anonymous apex to demonstrate

```
Lead[] leads = new list<Lead> {
    new Lead(Company = 'Foo00', LastName = 'LName00'),
    new Lead(Company = 'Foo01', LastName = 'LName01')
    };

insert leads;
leads[0].Company = 'Foo00Changed';
leads[1].Company = 'Foo01Changed';
update leads; // force future to execute in handler
```

Debug log(s)

Logs | Tests | Checkpoints | Query Editor | View State | Progress | Problems

User	Application	Operation	Time	Status
	Unknown	FutureHandler	12/16/2017, 7:33:55 AM	Success
	Unknown	FutureHandler	12/16/2017, 7:33:55 AM	Success
	Unknown	FutureHandler	12/16/2017, 7:33:55 AM	Success
	Unknown	FutureHandler	12/16/2017, 7:33:55 AM	Success
	Unknown	/services/data/v41.0/tooling/execu...	12/16/2017, 7:33:54 AM	Success

Ack! the future fired four(4) times! We should only have had two (2) as we updated only two records.

```
|USER_INFO|[EXTERNAL]|00540000000wbFS|cropredy@gmail.com|Pacific Standard Time|GMT-08:00
|EXECUTION_STARTED
//  lead inserted -- details omitted ...
//  Lead update event Lead.Company value changes
DML_BEGIN|[9]|Op:Update|Type:Lead|Rows:2
CODE_UNIT_STARTED|[EXTERNAL]|01q1W000000Tdah|LeadTrigger on Lead trigger event BeforeUpdate for [00Q1W00001Jh9VR, 00Q1W00001Jh9VS]
CODE_UNIT_FINISHED|LeadTrigger on Lead trigger event BeforeUpdate for [00Q1W00001Jh9VR, 00Q1W00001Jh9VS]
CODE_UNIT_STARTED|[EXTERNAL]|01q1W000000Tdah|LeadTrigger on Lead trigger event AfterUpdate for [00Q1W00001Jh9VR, 00Q1W00001Jh9VS]
USER_DEBUG|[7]|INFO|company has changed from Foo00to Foo00Changed .. request an @future to dowork
USER_DEBUG|[7]|INFO|company has changed from Foo01to Foo01Changed .. request an @future to dowork
CODE_UNIT_FINISHED|LeadTrigger on Lead trigger event AfterUpdate for [00Q1W00001Jh9VR, 00Q1W00001Jh9VS]
CODE_UNIT_STARTED|[EXTERNAL]|Workflow:Lead
WF_RULE_EVAL_BEGIN|Assignment
WF_SPOOL_ACTION_BEGIN|Assignment
WF_ACTION|.
WF_RULE_EVAL_END
WF_RULE_EVAL_BEGIN|Workflow
WF_CRITERIA_BEGIN|[Lead: LName00 00Q1W00001Jh9VR]|onUpdate - Set Field|01Q1W000000RGk3|ON_CREATE_OR_TRIGGERING_UPDATE|0
WF_RULE_FILTER|[Lead : Company contains Changed]
WF_RULE_EVAL_VALUE|Foo00Changed
WF_CRITERIA_END|true
WF_CRITERIA_BEGIN|[Lead: LName01 00Q1W00001Jh9VS]|onUpdate - Set Field|01Q1W000000RGk3|ON_CREATE_OR_TRIGGERING_UPDATE|0
WF_RULE_FILTER|[Lead : Company contains Changed]
WF_RULE_EVAL_VALUE|Foo01Changed
WF_CRITERIA_END|true
WF_SPOOL_ACTION_BEGIN|Workflow
WF_FIELD_UPDATE|[Lead: LName00 00Q1W00001Jh9VR]|Field:Lead: Mobile|Value:650-555-1212|Id=04Y1W000000PfJV|CurrentRule:onUpdate - Set Field (Id=01(
WF_FIELD_UPDATE|[Lead: LName01 00Q1W00001Jh9VS]|Field:Lead: Mobile|Value:650-555-1212|Id=04Y1W000000PfJV|CurrentRule:onUpdate - Set Field (Id=01(

 // Workflow updates the Leads with Field Update
WF_ACTION| Field Update: 2;
WF_RULE_EVAL_END

// before/after triggers on Lead re-fire (expected)
CODE_UNIT_STARTED|[EXTERNAL]|01q1W000000Tdah|LeadTrigger on Lead trigger event BeforeUpdate for [00Q1W00001Jh9VR, 00Q1W00001Jh9VS]
CODE_UNIT_FINISHED|LeadTrigger on Lead trigger event BeforeUpdate for [00Q1W00001Jh9VR, 00Q1W00001Jh9VS]
CODE_UNIT_STARTED|[EXTERNAL]|01q1W000000Tdah|LeadTrigger on Lead trigger event AfterUpdate for [00Q1W00001Jh9VR, 00Q1W00001Jh9VS]
```

```

// uh-oh, Trigger.old has values prior to the initial update DML,
// not the values as of the after update conclusion
USER_DEBUG|[7]|INFO|company has changed from Foo00to Foo00Changed .. request an @future to dowork
USER_DEBUG|[7]|INFO|company has changed from Foo01to Foo01Changed .. request an @future to dowork
CODE_UNIT_FINISHED|LeadTrigger on Lead trigger event AfterUpdate for [00Q1W00001Jh9VR, 00Q1W00001Jh9VS]
WF_ACTIONS_END| Field Update: 2;
CODE_UNIT_FINISHED|Workflow:Lead
DML_END|[9]
CODE_UNIT_FINISHED|execute_anonymous_apex
EXECUTION_FINISHED
```

Solution 1 (sounds good)

Just add a static recursion control variable to your handler

```
public class LeadTriggerHandler {
    static set<ID> leadIdsAlreadySentToFuture = new set<ID>(); // recursion control
    public void onAfterUpdate(Lead[] leads, map<ID,Lead> oldLeads) {
        for (Lead l: leads) {
            Lead oldLead = oldLeads.get(l.Id);
            if (l.Company != oldLead.Company &&
                !leadIdsAlreadySentToFuture.contains(l.Id)) { // have we already done this?
                System.debug(LoggingLevel.INFO,'company has changed from ' + oldLead.Company   +
                             'to ' + l.Company + ' .. request an @future to dowork');
                doCallout(l.Company);
                leadIdsAlreadySentToFuture.add(l.Id);
            }
        }
    }

    @future
    static void doCallout(String company) {
        System.debug(LoggingLevel.INFO,'future method to do callout for ' + company);
        // .. callout details not important
    }
}
```

This works as the debug log shows the future being called twice, once per Lead updated:

	Unknown	/services/data/v41.0/tooling/execu...	12/16/2017, 8:17:52 AM	Success
	Unknown	FutureHandler	12/16/2017, 8:17:52 AM	Success
	Unknown	FutureHandler	12/16/2017, 8:17:52 AM	Success

So, can I now work on my next JIRA ticket? Sorry ….

What if your Trigger/Handler is also invoked in a use case where partial success is allowed and one or more of the records fails to validate? AllOrNone = false can happen in many common use cases:

- Data Loader
- Any use of Apex Database.update(records,false); True also for the other Database.xxx methods.
- Bulk, SOAP, or REST APIs that either default AllOrNone to false or set explicitly if available.

Here, we run into another little-known SFDC feature of trigger retries in the AllOrNone = false (i.e. partial successes allowed) use case. This is documented in the Apex guide as:

Bulk DML Exception Handling

Exceptions that arise from a bulk DML call (including any recursive DML operations in triggers that are fired as a direct result of the call) are handled differently depending on where the original call came from:

- When errors occur because of a bulk DML call that originates directly from the Apex DML statements, or if the *allOrNone* parameter of a Database DML method was specified as `true`, the runtime engine follows the "all or nothing" rule: during a single operation, all records must be updated successfully or the entire operation rolls back to the point immediately preceding the DML statement.
- When errors occur because of a bulk DML call that originates from the SOAP API with default settings, or if the *allOrNone* parameter of a Database DML method was specified as `false`, the runtime engine attempts at least a partial save:
  1. During the first attempt, the runtime engine processes all records. Any record that generates an error due to issues such as validation rules or unique index violations is set aside.
  2. If there were errors during the first attempt, the runtime engine makes a second attempt that includes only those records that did not generate errors. All records that didn't generate an error during the first attempt are processed, and if any record generates an error (perhaps because of race conditions) it is also set aside.
  3. If there were additional errors during the second attempt, the runtime engine makes a third and final attempt which includes only those records that didn't generate errors during the first and second attempts. If any record generates an error, the entire operation fails with the error message, "Too many batch retries in the presence of Apex triggers and partial failures."

Note the following:

- During the second and third attempts, governor limits are reset to their original state before the first attempt. See Execution Governors and Limits.
- Apex triggers are fired for the first save attempt, and if errors are encountered for some records and subsequent attempts are made to save the subset of successful records, triggers are re-fired on this subset of records.

Going back to the Triggers and Order of Execution, there's one last tidbit as to why you can't use static variables for recursion control in an AllOrNone = false use case:

> *When a DML call is made with partial success allowed, more than one attempt can be made to save the successful records if the initial attempt results in errors for some records. For example, an error can occur for a record when a user-validation rule fails. Triggers are fired during the first attempt and are fired again during subsequent attempts. Because these trigger invocations are part of the same transaction,* ***static class variables that are accessed by the trigger aren't reset****. DML calls allow partial success when you set the allOrNone parameter of a Database DML method to false or when you call the SOAP API with default settings. For more details, see Bulk DML Exception Handling.*

So, if you do a bulk update of two records, and one fails the validation rule, the static recursion control variable will be set on the first attempt, any @future calls are rolled back, and, when SFDC makes the second attempt on the non-failed record, the recursion control prevents the callout attempt from even happening so you end up with no callouts!

Let's look at a proof:

Add a validation rule:

```
Website = 'www.failme.com'
```

Execute this code:

```
Lead[] leads = new list<Lead> {
    new Lead(Company = 'Foo00', LastName = 'LName00'),
    new Lead(Company = 'Foo01', LastName = 'LName01')
    };

insert leads;

leads[0].Company = 'Foo00Changed';
leads[1].Company = 'Foo01Changed';
leads[1].Website = 'www.failme.com';  // force partial success by failing this in VR
Database.SaveResult[] results = Database.update(leads,false); // allow partial success
```

Get this debug log

User	Application	Operation	Time	Status
	Unknown	/services/data/v41.0/tooling/execu...	12/16/2017, 9:09:40 AM	Success

No future logs! Future never happened!

```
CODE_UNIT_STARTED|[EXTERNAL]|execute_anonymous_apex
// 1st time trigger is executed - both Leads passed:
DML_BEGIN|[11]|Op:Update|Type:Lead|Rows:2
CODE_UNIT_STARTED|[EXTERNAL]|01q1W000000Tdah|LeadTrigger on Lead trigger event BeforeUpdate for [00Q1W00001Jh9Vv, 00Q1W00001Jh9Vw]
CODE_UNIT_FINISHED|LeadTrigger on Lead trigger event BeforeUpdate for [00Q1W00001Jh9Vv, 00Q1W00001Jh9Vw]
CODE_UNIT_STARTED|[EXTERNAL]|Validation:Lead:00Q1W00001Jh9Vv

// Validation rules execute
VALIDATION_RULE|03d1W000000Tdvy|Coerce_failure
VALIDATION_FORMULA|Website = 'www.failme.com'|Website=null
VALIDATION_PASS
CODE_UNIT_FINISHED|Validation:Lead:00Q1W00001Jh9Vv
CODE_UNIT_STARTED|[EXTERNAL]|Validation:Lead:00Q1W00001Jh9Vw
VALIDATION_RULE|03d1W000000Tdvy|Coerce_failure
VALIDATION_FORMULA|Website = 'www.failme.com'|Website=www.failme.com
// Fail the second Lead
VALIDATION_FAIL
CODE_UNIT_FINISHED|Validation:Lead:00Q1W00001Jh9Vw

// After update sees only the first, successful, Lead; future requested, static vbl set
CODE_UNIT_STARTED|[EXTERNAL]|01q1W000000Tdah|LeadTrigger on Lead trigger event AfterUpdate for [00Q1W00001Jh9Vv]
USER_DEBUG|[8]|INFO|company has changed from Foo00to Foo00Changed .. request an @future to dowork
CODE_UNIT_FINISHED|LeadTrigger on Lead trigger event AfterUpdate for [00Q1W00001Jh9Vv]
CODE_UNIT_STARTED|[EXTERNAL]|Workflow:Lead

// Workflow executes , causes field update on first lead
WF_RULE_EVAL_BEGIN|Workflow
WF_CRITERIA_BEGIN|[Lead: LName00 00Q1W00001Jh9Vv]|onUpdate - Set Field|01Q1W000000RGk3|ON_CREATE_OR_TRIGGERING_UPDATE|0
WF_RULE_FILTER|[Lead : Company contains Changed]
WF_RULE_EVAL_VALUE|Foo00Changed
WF_CRITERIA_END|true
WF_SPOOL_ACTION_BEGIN|Workflow
WF_FIELD_UPDATE|[Lead: LName00 00Q1W00001Jh9Vv]|Field:Lead: Mobile|Value:650-555-1212|Id=04Y1W000000PfJV|CurrentRule:onUpdate - Set Field (Id=01(
WF_ACTION| Field Update: 1;
WF_RULE_EVAL_END

// WF field update causes after trigger to re-execute (as expected)
CODE_UNIT_STARTED|[EXTERNAL]|01q1W000000Tdah|LeadTrigger on Lead trigger event BeforeUpdate for [00Q1W00001Jh9Vv]
CODE_UNIT_FINISHED|LeadTrigger on Lead trigger event BeforeUpdate for [00Q1W00001Jh9Vv]
// after trigger is NOP as recursion vbl says do nothing
CODE_UNIT_STARTED|[EXTERNAL]|01q1W000000Tdah|LeadTrigger on Lead trigger event AfterUpdate for [00Q1W00001Jh9Vv]
CODE_UNIT_FINISHED|LeadTrigger on Lead trigger event AfterUpdate for [00Q1W00001Jh9Vv]
WF_ACTIONS_END| Field Update: 1;
CODE_UNIT_FINISHED|Workflow:Lead
```

```

// SFDC retries the first record because AllOrNone=false; governor limits reset
//  But static variables are not reset
CODE_UNIT_STARTED|[EXTERNAL]|01q1W000000Tdah|LeadTrigger on Lead trigger event BeforeUpdate for [00Q1W00001Jh9Vv]
CODE_UNIT_FINISHED|LeadTrigger on Lead trigger event BeforeUpdate for [00Q1W00001Jh9Vv]
CODE_UNIT_STARTED|[EXTERNAL]|Validation:Lead:00Q1W00001Jh9Vv

// WF fires again, updates the first Lead but no callout done  as recursion vbl prevents
WF_RULE_EVAL_BEGIN|Workflow
WF_CRITERIA_BEGIN|[Lead: LName00 00Q1W00001Jh9Vv]|onUpdate - Set Field|01Q1W000000RGk3|ON_CREATE_OR_TRIGGERING_UPDATE|0
WF_RULE_FILTER|[Lead : Company contains Changed]
WF_RULE_EVAL_VALUE|Foo00Changed
WF_CRITERIA_END|true
WF_SPOOL_ACTION_BEGIN|Workflow
WF_FIELD_UPDATE|[Lead: LName00 00Q1W00001Jh9Vv]|Field:Lead: Mobile|Value:650-555-1212|Id=04Y1W000000PfJV|CurrentRule:onUpdate - Set Field (Id=01(
WF_ACTION| Field Update: 1;
WF_RULE_EVAL_END
CODE_UNIT_STARTED|[EXTERNAL]|01q1W000000Tdah|LeadTrigger on Lead trigger event BeforeUpdate for [00Q1W00001Jh9Vv]
CODE_UNIT_FINISHED|LeadTrigger on Lead trigger event BeforeUpdate for [00Q1W00001Jh9Vv]

// no callout request made in retry of first record
CODE_UNIT_STARTED|[EXTERNAL]|01q1W000000Tdah|LeadTrigger on Lead trigger event AfterUpdate for [00Q1W00001Jh9Vv]
CODE_UNIT_FINISHED|LeadTrigger on Lead trigger event AfterUpdate for [00Q1W00001Jh9Vv]
WF_ACTIONS_END| Field Update: 1;
CODE_UNIT_FINISHED|Workflow:Lead
DML_END|[11]
CODE_UNIT_FINISHED|execute_anonymous_apex
EXECUTION_FINISHED
```

So now what?

If we take the recursion static variable away, then the AllOrNone use case will still not pass – the future will get called twice on the successful record and never on the failed record.

```
... after the VR fails record[1] and before the WF executes
USER_DEBUG|[8]|INFO|company has changed from Foo00 to Foo00Changed .. request an @future to dowork
... after the WF updates record[0] .. our original issue
USER_DEBUG|[8]|INFO|company has changed from Foo00to Foo00Changed .. request an @future to dowork
.. SFDC retries the successful records in trigger.new; skips the failed ones
.. trigger re-executes as if none of the above ever happened
USER_DEBUG|[8]|INFO|company has changed from Foo00 to Foo00Changed .. request an @future to dowork
... after the WF updates record[0] .. our original issue
USER_DEBUG|[8]|INFO|company has changed from Foo00to Foo00Changed .. request an @future to dowork
```

User	Application	Operation	Time
	Unknown	/services/data/v41.0/tooling/execu...	12/16/2017, 9:26:22 AM
	Unknown	FutureHandler	12/16/2017, 9:26:22 AM
	Unknown	FutureHandler	12/16/2017, 9:26:22 AM

Clearly, static variables can't be used to control redundant callouts when workflows and AllOrNone = false are combined in the same use case.

Solution 2 (better, but fragile)

Go through your workflows that could update the Object where you are testing for make-a-callout conditions.

- Move the field updates out and put them in the before insert/update triggers.
- This way, the workflow will never force the trigger to re-execute with the original, start-of-transaction state of Trigger.old
- Hence, your doCallout logic will only execute once in the transaction

This is fragile because you or some colleague could add at some later time a new workflow+field update that causes the trigger's callout-evaluating condition to be re-assessed and you'll be making duplicate callouts again.

Solution 3 – what could it be?

Clearly, you need to have state that:

1. Persists across the trigger – workflow field update – trigger sequence
2. Is rolled back when SFDC retries in the AllorNone = false (partial success) use case

We've seen that static variables won't work. Platform cache would not work as it isn't rolled back in the AllOrNone = false scenario.

The only thing that meets both criteria would be Sobject fields. A general approach …

1. Trigger sets the field in Trigger.new (or updates some other Sobject with a known key)

2. Workflow field update is made, trigger re-executes. Using the values in trigger.new, looks to see if the work was already done and if yes, avoids asking to do it again
3. If trigger is running in a AllOrNone = false use case, and a record fails in the batch, the update made in step 1 is rolled back by SFDC. Thus, the trigger re-requests the work, persists the request in an sobject, and even though the workflow will re-fire, the persisted sobject can be inspected on the second trigger re-fire and the dowork request skipped

Now, what would you save in that sobject field?

1. Initially, I was tempted to save the sessionId (encrypted as a MAC digest) as a pseudo signal that the callout request was made. As an sobject, it would be rolled back in the AllOrNone-false with error use case. But, when the Bulk API is used, there is no sessionId — it is null.
2. Next idea was to exploit lastModifiedDate and lastModifiedById and compare the running user and current time to see if the trigger is being used in the trigger-workflow+field update-trigger use case as a sort of pseudo state variable. This seems problematic for several reasons like time skew and concurrent transactions coming from the same user
3. Another idea was an unconditional workflow field update to set a field called Is_WF_Update_in_Progress__c. Thus, the second time through the trigger, the code would inspect the Is_WF_Update_in_Progress__c in Trigger.new, say, “ha”, I’m being called in a workflow field update-induced trigger invocation and bypass the request to do a callout. But, then the new field would have to be cleared (another DML) and, we’d be unnecessarily forcing the trigger to re-execute even if no other workflow’s rule criteria were satisfied. This slows down performance. This is complicated and doesn’t scale well. Every SObjectType needs its own clone of this if involved in workflows + triggers that compare trigger.old to trigger.new

A Workable, albeit a bit heavyweight solution

Create a Custom Object Transaction_State__c

One relevant field:

- Name – This will be the transactionId

Create a Transaction Service class

```
public class TransactionService {

    /**
    *   Meat of the TransactionService
    *
    *       1 - Set/get of transactionId
    *       2 - Track visitedIds by scopeKey to avoid recursion
    *       3 - Track whether a "context" is enabled or disabled - especially useful for testmethods to switch of async handling
    **/

```

```
    @TestVisible private static String transactionId;

    /**
    *   get(set)TransactionId - use to record some identifier for the transaction. Particularly useful for incoming REST calls
    *                           so methods can reference without having to pass around in arguments
    **/

    public virtual String getTransactionId() {
        return transactionId == null ? transactionId = String.valueOf(System.currentTimeMillis()) + '_' + UserInfo.getName() : transactionId;
    }

    public virtual void setTransactionId(String txnId) {
        transactionId = txnId;
    }

    public virtual Boolean hasTransactionId()
    {
        return transactionId == null ? false : true;
    }

    private static map<String,Boolean>    enablementByContext = new map<String,Boolean> ();

    /**
    *   isDisabled (String  context) - returns true if this context has been switched off
    *       future enhancement - read from custom metadata to allow external (dis)(en)ablement
    **/
    public virtual Boolean isDisabled(String context) {
        return enablementByContext.containsKey(context)
            ? !enablementByContext.get(context)     // in map, return whether enabled or disabled
            : false;  // no entry, hence enabled
    }
    /**
    *   isEnabled (String  context) - returns true if this context has been switched on or never entered
    *       future enhancement - read from custom metadata to allow external (dis)(en)ablement
    **/
    public virtual Boolean isEnabled(String context) {
        return enablementByContext.containsKey(context)
            ? enablementByContext.get(context)      // in map, return whether enabled or disabled
            : true;  // no entry, hence enabled
    }

    /**
    *   setEnablement(String context, Boolean isEnabled)
    **/
    public virtual void setEnablement(String context, Boolean isEnabled) {
        if (isEnabled == null)
            throw new TransactionService.TransactionServiceException('setEnablement isEnabled argument can not be null');
        enablementByContext.put(context,isEnabled);
    }



    static ID txnStateIdAsProxyForStateTrust;   // beacon to tell us if we can trust static variables

    /**
```

```
66      *   establishStateTrust - Transaction_State__c is an sobject
67      *                           1 - so, it is rolled back on allOrNone = false retry
68      *                           2 - hence we point at it with a static variable that isn't rolled back on retry
69      *                           3 - If the two don't agree, we know we are retrying and static map must be reset to empty
70      **/
71      private void establishStateTrust() {
72          if (txnStateIdAsProxyForStateTrust == null) { // no trust yet setup
73              resetStateTrust();
74          }
75          else {
76              //  if we have an sobject, has it been rolled back because we are in an AllOrNone = false
77              //  (partial success) SFDC-initiated retry use case on the "successes"?
78              Transaction_State__c[] txnStates = [select Id, Name from Transaction_State__c where Id = : txnStateIdAsProxyForStateTrust];
79              if (txnStates.isEmpty()) { // static vbl points at sobject that has been rolled back
80                  resetStateTrust();
81              }
82              else {} // if the static variable we established points at an existing Transaction_State__c,
83                      // that means we are not in an AllOrNone = false retry step and the static variables
84                      // maintaining state can be relied on. Thus, triggers re-executed
85                      // as part of a workflow/Process Builder can avoid repeating logic
86          }
87      }
88
89      private void resetStateTrust(){
90         Transaction_State__c txnState = new Transaction_State__c(Name = transactionid);
91         insert txnState;
92         txnStateIdAsProxyForStateTrust = txnState.Id;
93         clearVisitedCaches();
94      }
95
96      /**
97      *   Map takes care of visited Ids by scopeKey and is valid up until the point that a retry
98      *   is detected; then map is cleared and we start afresh
99      **/
100     static map<String,Set<ID>>  visitedIdsThisTxnByScopeKey = new map<String,set<ID>> ();
101
102     public virtual set<ID> getUnvisitedIdsThisTxn(String scopeKey, set<ID> proposedIds) {
103
104         establishStateTrust();
105         if (visitedIdsThisTxnByScopeKey.containsKey(scopeKey)) {
106             set<ID> unvisitedIds = new set<ID>(proposedIds);                    // start with proposedIds as unvisited
107             unvisitedIds.removeAll(visitedIdsThisTxnByScopeKey.get(scopeKey));  // remove any Ids we've already seen
108             visitedIdsThisTxnByScopeKey.get(scopeKey).addAll(proposedIds);      // update visited set
109             return unvisitedIds;
110         }
111         else {                                                                  // new scopeKey, hence all ids are unvisited
112             visitedIdsThisTxnByScopeKey.put(scopeKey,new set<ID>(proposedIds));
113             return proposedIds;
114         }
115     }
116
117     /**
118     *   peekVisitedIdsThisTxn - Inspect visitedIDs this Transaction without affecting set (for a given scope key)
119     **/
120     public virtual set<ID> getVisitedIdsThisTxn(String scopeKey) {
```

```
        return visitedIdsThisTxnByScopeKey.containsKey(scopeKey) ? visitedIdsThisTxnByScopeKey.get(scopeKey) : new set<ID>();
    }

    /**
    *   getVisitedIdsThisTxn - Inspect visitedIDs this Transaction without affecting set (all scope keys)
    **/
    public virtual map<String,set<ID>> getVisitedIdsThisTxn() {
        return visitedIdsThisTxnByScopeKey;
    }


    /**
    *   clearVisitedCache() - Clears specific visited ID cache
    **/
    public virtual void clearVisitedCache(String scopeKey) {
        if (visitedIdsThisTxnByScopeKey.containsKey(scopekey))
            visitedIdsThisTxnByScopeKey.get(scopekey).clear();
        else
            throw new TransactionService.TransactionServiceException('Invalid scopeKey: ' + scopeKey + ' for clearVisitedCaches');
    }
    /**
    *   clearVisitedCaches()    - Clears all visited ID caches; useful for testmethods
    **/
    public virtual void clearVisitedCaches() {
        visitedIdsThisTxnByScopeKey.clear();
    }


}
```

Modify the triggerhandler code as follows

```
public class LeadTriggerHandler {
    public void onAfterUpdate(Lead[] leads, map<ID,Lead> oldLeads) {

        set<ID> unvisitedIds = TransactionService.getVisitedIdsThisContext('LeadDoFuture',oldLeads.keySet());
        for (Lead l: leads) {
            Lead oldLead = oldLeads.get(l.Id);
            if (!unvisitedIds.contains(l.Id) && l.Company != oldLead.Company)    {
                System.debug(LoggingLevel.INFO,'company has changed from ' + oldLead.Company   +
                             'to ' + l.Company + ' .. request an @future to dowork');
                doCallout(l.Company);

            }
        }
    }

    @future
    static void doCallout(String company) {
        System.debug(LoggingLevel.INFO,'future method to do callout for ' + company);
        // .. callout details not important
```

```
    }
}
```

The triggerhandler asks the Transaction Service to get all unvisited Ids for some context scope. Behind the scenes, the TransactionService saves the Ids + context scope + TransactionId in the database, thus creating a persistent store for the AllOrNone = true use case and a rolback-able store for the AllOrNone = false use case.

Now, if you run an AllOrNone = true use case

```
Lead[] leads = new list<Lead> {
    new Lead(Company = 'Foo00', LastName = 'LName00'),
    new Lead(Company = 'Foo01', LastName = 'LName01')
    };

insert leads;
leads[0].Company = 'Foo00Changed';
leads[1].Company = 'Foo01Changed';
update leads; // force future to execute in handler
```

You see the future is called twice, once per record

	Unknown	/services/data/v41.0/tooling/execu...	12/16/2017, 5:47:24 PM	Success
	Unknown	FutureHandler	12/16/2017, 5:47:24 PM	Success
	Unknown	FutureHandler	12/16/2017, 5:47:24 PM	Success

If you run in an AllOrNone = false use case

```
Lead[] leads = new list<Lead> {
    new Lead(Company = 'Foo00', LastName = 'LName00'),
    new Lead(Company = 'Foo01', LastName = 'LName01')
    };

insert leads;

leads[0].Company = 'Foo00Changed';
leads[1].Company = 'Foo01Changed';
leads[1].Website = 'www.failme.com';  // force partial success by failing this in VR
Database.SaveResult[] results = Database.update(leads,false); // allow partial success
```

You see the future is only called once for the record that does not fail validation rules

	Unknown	FutureHandler	12/16/2017, 5:47:51 PM	Success
	Unknown	/services/data/v41.0/tooling/execu...	12/16/2017, 5:47:50 PM	Success