Entry Points. Triggers, Batches, Controllers etc.

Overview

There is a category of Apex types that are in effect entry points into the business logic of any org. Each of these classes are designed and intended to run in very specific contexts. This means that any code in these is not re-usable anywhere else. This is not really different from many other systems, but it is worth mentioning some of the Salesforce specific areas.

Unfortunately it is very common to see developers simply put the code they need to write in these classes, which then causes issues later on.

Triggers

An important consideration of Apex development is how to code with triggers in mind, as they are a key entry point into a lot of logic. To a large extent Salesforce requires that a lot of business logic runs at the point records are written to the database; and for Apex that means triggers; elsewhere it may be Flows, Process Builders, CDC etc.

Best practise on Salesforce is to only have 1 trigger per object - that of course means of the ones you can control, as you have little control over what managed packages install. Also, good practise is to have as little logic in the trigger as possible and instead hand over to some trigger handling class or service as soon as possible. If this is done well then most developers can more or less ignore triggers after that, as they will just be writing and maintaining the business logic in service/domain style classes without regard to trigger semantics.

An example of a bad trigger from our org:

```
trigger bg_Contract_au on Contract(after update)
{

if(iHASCOUtility.userDivison == 'iHASCO'){
        System.debug('bg_Contract_ad.userDivison'+iHASCOUtility.
userDivison);
      bg_Contract_Helper.handleiHASCOContractUpdate(trigger.new,
trigger.oldMap);
    }
    Else{

    if(!ByPassUtils.isByPassed('ContractAfterUpdate')){
        bg_Contract_Helper.setContractOnQuote(trigger.new, trigger.
oldMap);

        // START SDT-4706: Skip after update for Citation Partnership
Record type
        Id citationPartnershipRecordTypeId = Schema.SObjectType.
Contract.getRecordTypeInfosByName().get('Citation Partnership').
getRecordTypeId();
        List<Contract> newList = new List<Contract>();
        Map<Id, Contract> newMap = new Map<Id, Contract>();
        Map<Id, Contract> oldMap = new Map<Id, Contract>();

        for(Contract objContract : trigger.new) {
            if(objContract.RecordtypeId !=
citationPartnershipRecordTypeId) {
                newMap.put(objContract.Id, objContract);
                oldMap.put(objContract.Id, trigger.oldmap.get
(objContract.Id));
            }
            else if(objContract.Status != Trigger.oldmap.get
(objContract.Id).Status){
                newList.add(objContract);
            }
        }

        //Update related Account -> Is_Active_Partner as True if we
have active 'Citation Partnership' contracts.
        bg_Contract_Helper.runPartnerAccount(newList);

        bg_Contract_Helper.checkAccountToUpdateCall(newMap,oldMap);

        [... and more ...]
```

There is business logic in this trigger - routing based on user division, potentially some quoting logic, routing based on record type, or whether a specific field has changed. Due to the separation between triggers and the Apex classes it will not be obvious that this is going on when anyone is adding features or maintaining features. If someone ever expects to be able to do the same sort of things within a UI then there is no chance of re-use, which means we may get duplication of code and bugs due to inconsistency etc.

There are multiple triggers for contract, and several call this `bg_Contract_Helper` helper class. Working out the overall process is somewhat awkward. Also 2 of these triggers (the above and one other) are fired on afterUpdate, so both triggers will fire in an unknown order.

A better trigger might look something like this (ignore whether the code looks like other examples I've shown elsewhere, it is the concept that is important):

```
trigger ContractTrigger on Contract (before insert, before update,
before delete,  after insert, after update, after delete, after
undelete)
{
    ITriggerHandler handler = DefaultTriggerHandler.getInstance(new
ContractService());

    if (Trigger.isBefore)
    {
        if(Trigger.isInsert)
        {
            handler.beforeInsert(Trigger.new);
        }

        if(Trigger.isUpdate)
        {
            handler.beforeUpdate(Trigger.newMap, Trigger.oldMap);
        }

        if(Trigger.isDelete)
        {
            handler.beforeDelete(Trigger.old);
        }
    }
    else if (Trigger.isAfter)
    {
        if(Trigger.isInsert)
        {
            handler.afterInsert(Trigger.new);
        }

        if(Trigger.isUpdate)
        {
            handler.afterUpdate(Trigger.newMap, Trigger.oldMap);
        }

        if(Trigger.isDelete)
        {
            handler.afterDelete(Trigger.old);
        }

        if(Trigger.isUndelete)
        {
            handler.afterUndelete(Trigger.new);
        }
    }
}
```

Note the the above is not the only way of doing this, but it shows the general idea; that is that the trigger simply instantiates some Handler for dealing with Apex logic, from where the flexibility and programming structures of the language make it easier to write readable and maintainable code, and to see in a single place all the logic that might happen on any given trigger event.

As is obvious from this, once this trigger is written there will be little reason to revisit it, as there is little else it can do. Any changes will be in the trigger handler, although even that is somewhat areas they tend to be quite stable, and is more likely the service class will change for new features etc. There will be no need to have many triggers, as the trigger handler will deal with all requests.

Batches

Another very common scenario in Salesforce is batch processing, and the use of Apex Batch interfaces. Generally speaking a batch process is not doing anything that should only ever be done in a batch; it is quite common that at some point what a batch does over many many records will want to be done in some other context with fewer (maybe 1) records, triggered from a UI button for example. However, it is very common to see Batch classes full of business logic. This prevents re-use of such logic and makes it harder to test due to certain limits on Apex tests and Asynchronous processing.

Best practise for Salesforce is that a batch do nothing other than load up records and pass them straight into a service or domain style class to handle. This service should in no way be batch specific. This allows that logic to be re-used elsewhere and more easily tested outside the issues that batch testing imposes.

An example of a bad Batch from our Org (only the execute shown):

```
global void execute (Database.BatchableContext bc ,List<Opportunity>
scope){
        Map<String,String> prospectToMarketingChannel=new Map<String,
String>();
        Map<String,String> ProspectToSalesChannel=new Map<String,
String>();
        Set<String> channels=new Set<String>();
        List<Prospect_Source_Data__mdt> prospectSourceMDT=new
List<Prospect_Source_Data__mdt>();
        prospectSourceMDT=[SELECT Label,Marketing_Channel__c,
Sales_Channel__c from Prospect_Source_Data__mdt];
        for(Prospect_Source_Data__mdt pmdt:prospectSourceMDT){
            prospectToMarketingChannel.put(pmdt.Label,pmdt.
Marketing_Channel__c);
            ProspectToSalesChannel.put(pmdt.Label,pmdt.
Sales_Channel__c);
            channels.add(pmdt.Sales_Channel__c);
        }
        List<Opportunity> oppties=new List<Opportunity>();
        for(Opportunity opp:scope){
            boolean scupdated=false;
            if(opp.Type==null){
                 opp.Type='Existing Business';
               }
            if(opp.LeadSource!=null && prospectToMarketingChannel.
containsKey(opp.LeadSource)){
                    opp.Marketing_Channel__c=prospectToMarketingChannel.
get(opp.LeadSource);
                }
                if(opp.LeadSource==null || opp.LeadSource==''|| (opp.
LeadSource!=null && !prospectToMarketingChannel.containsKey(opp.
LeadSource))){
                    opp.Marketing_Channel__c='';
```

```
                }
                if(opp.CampaignId!=null){
                    if(opp.Campaign.Name.Contains('EPM') || opp.
Campaign.Name.Contains('QMS') || opp.Campaign.Name.Contains('SMAS') ||
opp.Campaign.Name.Contains('P&R') || opp.Campaign.Name.Contains('SM
UK') || opp.Campaign.Name.Contains('HS Direct') || opp.Campaign.Name.
Contains('EL Direct') || opp.Campaign.Name.Contains('Southalls') || opp.
Campaign.Name.Contains('Avec') || opp.Campaign.Name.Contains('Food
Alert')){
                        opp.Sales_Channel__c='Cross Sell';
                        scupdated=true;
                    }
                }
                if(opp.LeadSource!=null && (opp.LeadSource.contains
('EPM') || opp.LeadSource.contains('QMS')|| opp.LeadSource.contains
('P&R') || opp.LeadSource.contains('SMAS')|| opp.LeadSource.contains
('SM UK')|| opp.LeadSource.contains('HS Direct')|| opp.LeadSource.
contains('EL Direct')|| opp.LeadSource.contains('Southalls')|| opp.
LeadSource.contains('Avec')|| opp.LeadSource.contains('Food Alert')) &&
!scupdated){
                    opp.Sales_Channel__c='Cross Sell';
                    scupdated=true;
                }
                if(opp.Referred_By_Citation_Company__c!=null && (opp.
Referred_By_Citation_Company__c.contains('EPM') || opp.
Referred_By_Citation_Company__c.contains('QMS')|| opp.
Referred_By_Citation_Company__c.contains('P&R') || opp.
Referred_By_Citation_Company__c.contains('SMAS')|| opp.
Referred_By_Citation_Company__c.contains('SM UK')|| opp.
Referred_By_Citation_Company__c.contains('HS Direct')|| opp.
Referred_By_Citation_Company__c.contains('EL Direct')|| opp.
Referred_By_Citation_Company__c.contains('Southalls')|| opp.
Referred_By_Citation_Company__c.contains('Avec')|| opp.
Referred_By_Citation_Company__c.contains('Food Alert')) && !scupdated){
                    opp.Sales_Channel__c='Cross Sell';
                    scupdated=true;
                }
                if(opp.LeadSource!=null && (opp.LeadSource.contains
('SEM') || opp.LeadSource.contains('Seminar')) && !scupdated){
                    opp.Sales_Channel__c='Seminar';
                    scupdated=true;
                }
                if(opp.LeadSource!=null && opp.Sales_Criterion__c!=null
&& ProspectToSalesChannel.containsKey(opp.LeadSource)
                    && ProspectToSalesChannel.get(opp.LeadSource).equals
('Outbound or Inbound') && !scupdated){
                    opp.Sales_Channel__c=opp.Sales_Criterion__c;
                    scupdated=true;
                }
                if(opp.LeadSource!=null && ProspectToSalesChannel.
```

```
containsKey(opp.LeadSource) && !scupdated){
                    opp.Sales_Channel__c=ProspectToSalesChannel.get(opp.
LeadSource);
                    scupdated=true;
                }
                if(!scupdated){
                    opp.Sales_Channel__c='Data Incomplete';
                }
            oppties.add(opp);
        }
        if(!oppties.isEmpty())
        database.update(oppties,false);
    }
```

This code is really business logic around opportunities. Because of its location it can only really be executed by firing off the batch. If this logic was in a standalone service it could be called by this batch as well as elsewhere - e.g. a UI or trigger. As it happens this code has been duplicated more or less verbatim elsewhere as the logic was desired outside of this batch process. That may have happened the other way around, the other location is so un-reusable and it was maybe copied here later, but still not done properly. It appears that if we needed elsewhere then current practise is copy->paste. This is extremely bad practise as it makes it likely that over time the logic in the multiple locations will diverge accidently, resulting in bugs and inconsistent behaviour.

A better version of the above would probably looks something like this:

```
public void execute (Database.BatchableContext bc ,List<Opportunity>
scope)
{
  Set<Id> oppIds = Map<Id, Opportunity>(scope).keySet();
  New OpportunityService(new OpportunityChannelSelector()).
calculateChannel(oppIds);
}
```

Note, the name of the method called may be something else, I can’t be sure exactly what the function is really doing without more knowledge. However, the important part is that the batch just calls into a service that might also be called from elsewhere in different use cases.

It might be noted that the code above only pulled the Ids from the scope and passed them to the service. This is another general good practise of batch processing, but it also ensures the service has the most generic and safest method signature - it will query the Opportunities itself so as to avoid any required field knowledge leaking and having to be maintained outside the service.

Controllers

Controllers, be they Visualforce or Lightning, are intended to manage the UI. The code in these should only have logic that is directly responsible for maintaining UI state; e.g. toggle state, button visibility etc. no business logic should appear in these classes.

Again, however, it is very common in our org to see huge controllers performing significant logic, which can be awkward to test and cannot be readily reused when the logic is desired outside of that specific UI.

An example of a bad controller in our org (this is only a singe method in a controller):

```
public PageReference validateStripePayementDetails() {
        try {
            String encodedString = ApexPages.currentPage().
getParameters().get('contractOrderId');
            Blob decodeBlob = EncodingUtil.base64Decode(encodedString);
```

```
            String decodedString;
            decodedString = decodeBlob.toString();
            System.debug('Decoded-String ' + decodedString);
            contractOrderId = decodedString.substringBefore('-');
            DateTime expiryDateTime = DateTime.valueOf(decodedString.
substringAfter('-'));
            System.debug('::contractOrderId '+contractOrderId );
            System.debug('::expiryDateTime '+expiryDateTime );
            String objName = contractOrderId.getsobjecttype().
getDescribe().getName();
            if (String.isEmpty(contractOrderId) && (objName !=
'Contract' || objName != 'Order')) {
                return redirectToErrorPage();
            }
            if  (objName == 'Contract') {
                Contract[] cont = [
                    SELECT Id, Name, Status, Source_Opportunity__c,
SBQQ__Opportunity__c,
                        Total_Contract_Value__c, CustomerSignedId,
CustomerSigned.Name,
                        CustomerSigned.AccountId, CustomerSigned.
Account.Name, Payment_Status__c,
                        StripePaymentLinkExpiryDateTime__c,
SBQQ__Quote__r.name,SBQQ__Quote__c,
                            Account.BillingStreet,Account.Billingcity,
Account.BillingState,Account.BillingCountry,
                            CustomerSigned.email,CustomerSigned.phone
                    FROM Contract
                    WHERE Id = :contractOrderId AND
StripePaymentLinkExpiryDateTime__c = :expiryDateTime
                    LIMIT 1
                ];


                if ((cont == null && cont.size() == 0) ||
                    (cont != null && cont.size() > 0 && ((cont[0].
Payment_Status__c != 'Payment Link Sent') && (cont[0].Payment_Status__c
!= 'Payment Failed') )
                     || DateTime.now() > cont[0].
StripePaymentLinkExpiryDateTime__c) ) {
                    return redirectToErrorPage();
                }
                // Populate CustomerBilling details
                customerBillingWrap = new CustomerAndBillingDetailsWrap
(cont[0], true);
            } else if (objName == 'Order') {
                 Order[] ord = [
                    SELECT Id, Name, Status,
                        TotalAmount, CustomerAuthorizedById,
CustomerAuthorizedBy.Name,
```

```
                        CustomerAuthorizedBy.AccountId,
CustomerAuthorizedBy.Account.Name,
                        StripePaymentLinkExpiryDateTime__c,
Payment_Status__c,
                            SBQQ__Quote__r.name,Account.BillingStreet,
Account.Billingcity,Account.BillingState,
                            Account.BillingCountry,
CustomerAuthorizedBy.email,CustomerAuthorizedBy.phone
                    FROM Order
                    WHERE Id = :contractOrderId AND
StripePaymentLinkExpiryDateTime__c = :expiryDateTime
                    LIMIT 1
                ];


                if ((Ord == null && Ord.size() == 0) ||
                    (Ord != null && Ord.size() > 0 && ((Ord[0].
Payment_Status__c != 'Payment Link Sent') && (Ord[0].Payment_Status__c !
= 'Payment Failed') )
                     || DateTime.now() > Ord[0].
StripePaymentLinkExpiryDateTime__c) ) {
                    return redirectToErrorPage();
                }
                customerBillingWrap = new CustomerAndBillingDetailsWrap
(ord[0], true);
            }
        } Catch(Exception ex) {
            return  redirectToErrorPage();
        }

        return null;
    }
```

This method validates something about payment details. This seems like a fairly important piece of logic that could well be required from any code path, kicked off from any entry point. The UI specific parts should be in here, e.g. page redirects, but it should be calling some payment validation service and making such choices based on the response.

A better version of the above would look something like below:

```
public PageReference getPayementDetails()
{
  PaymentDetails paymentDetails = getPaymentDetails();  String
encodedString = ApexPages.currentPage().getParameters().get
('contractOrderId');
  recordId = paymentDetails.getPaymentDetails();
  DateTime expiryDateTime = paymentDetails.getExpiryDateTime();

  SObject record;
  try
  {
    record = new StripePaymentService().getPaymentDetails(recordId,
expiryDateTime);
  }
  catch(Exception e)
  {
    return redirectToErrorPage();
  }

  customerBillingWrap = new CustomerAndBillingDetailsWrap(record);

  return null;
}
```

Note, that I've slightly renamed the method, as the logic was more about retrieving data then validating. There was some validation that redirected to an error page, but that felt more incidental.

The controller now only does what it needs to, and leaves the logic itself to a re-useable service. So, the inputs to the service are a UI responsibility, as it knows they came in via some encoded HTTP parameter and knows how that will look. If there is some error then it knows to redirect to the error page, but whether there is an error is left to the service to signal via an exception. The exception may be a specific one, and it may contain more information, but this controller was not obviously interested in what went wrong.