Apex: Consuming SObjects (Standard and Custom Objects, Platform Events and CDC)

Many companies have Apex trigger handlers that consume Standard and Custom Objects (SObjects) by subscribing to Before/After events. With the addition of Platform Events (High Volume and Change Data Capture) companies now must decide how to integrate the former trigger handling with Platform Events, if at all.

This document describes a Model to support these components seamlessly in an Apex environment. Please note, the term Sobject will refer to standard, custom objects as well as Platform Events (High-Volume and Standard) and Change Data Capture Platform Events.

This document DOES NOT get into the lower details of Platform Events (High Volume or CDC). This document **assumes** the reader has good understanding of Platform Events (High Volume or CDC) as well as simple Object Oriented principles.

Static Model

First, let's ensure there is a basic understanding of the underlying class model dealing with SObjects.

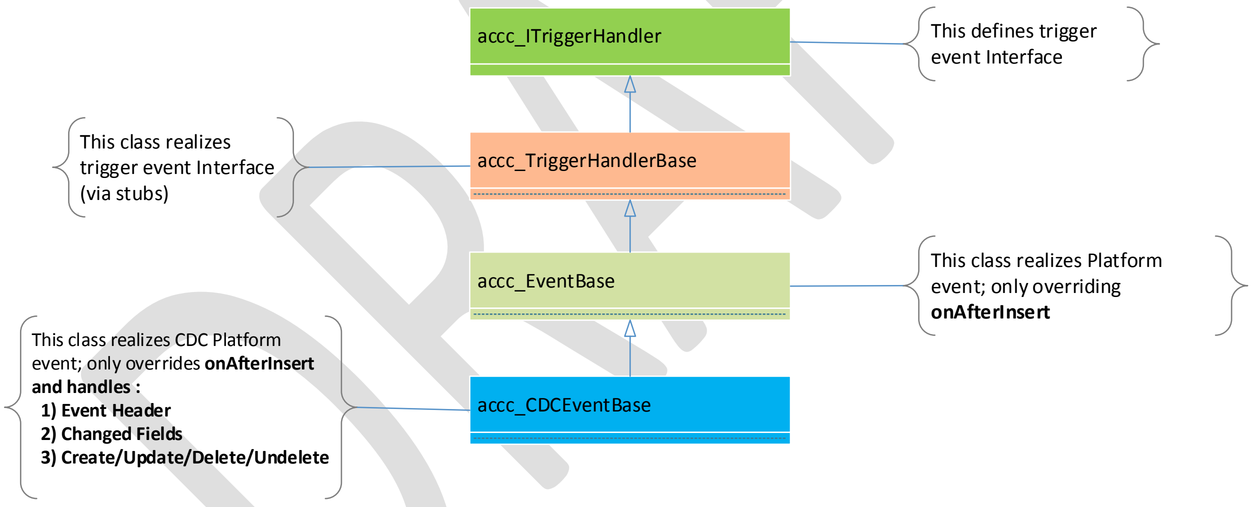

accc_ITriggerHandler
This defines trigger event Interface
This class realizes trigger event Interface (via stubs)
accc_TriggerHandlerBase
accc_EventBase
This class realizes Platform event; only overriding **onAfterInsert**
This class realizes CDC Platform event; only overrides **onAfterInsert** and handles : 1) Event Header 2) Changed Fields 3) Create/Update/Delete/Undelete
accc_CDCEventBase

Figure 1 High-Level Static Model

The above static class model provides the template for handling the different types of events. In order to provide a better realization of the benefits and breakdown we can further segment as follows:

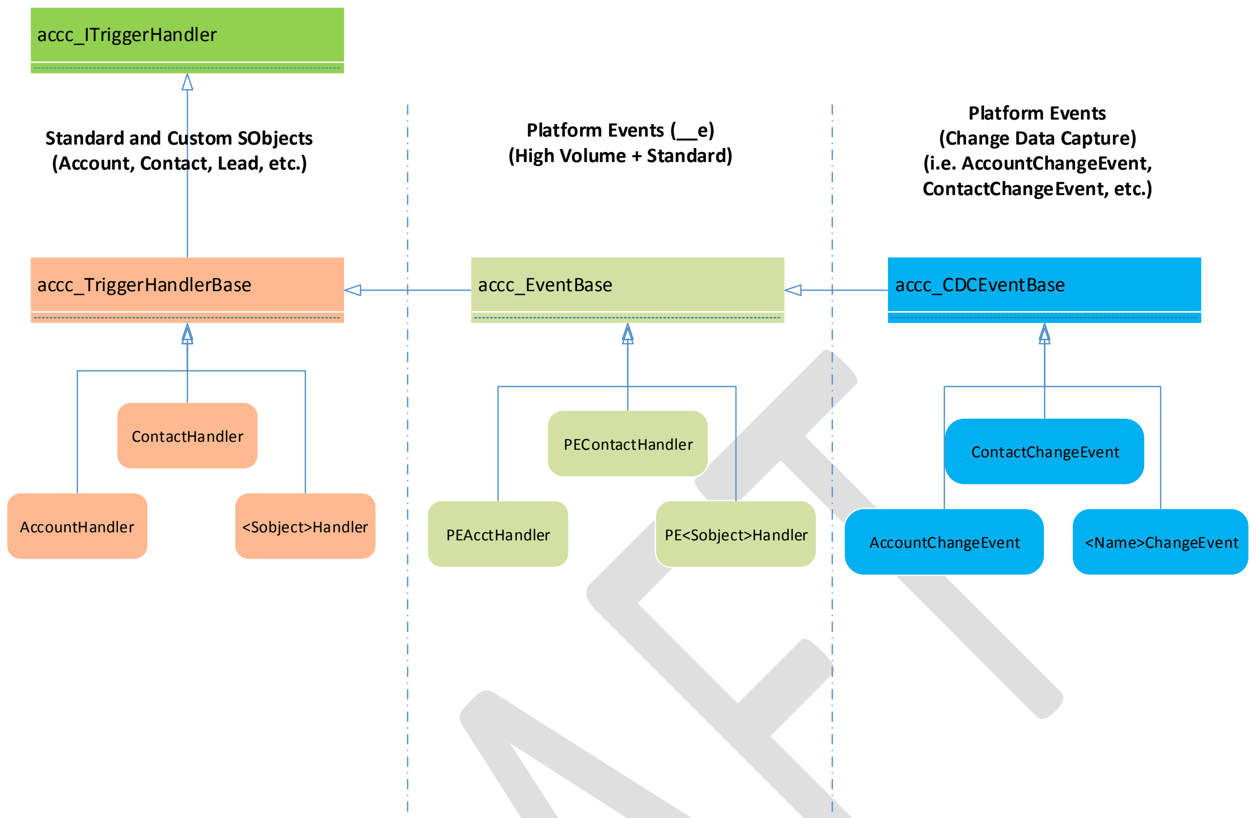


Figure 2 Segmentation of Trigger Handling

The handlers functionality are segmented based on their base class. For example, your standard and custom objects, handlers inherit from **accc_TriggerHandlerBase**. While your Platform Events (High Volume and Standard), handlers inherit from **accc_EventBase**. Finally, your CDC Events, handlers inherit from **accc_CDCEventBase.**

Advantages of the Class Hierarchy

The advantage of using this class hierarchy are as follows:

- A factory can be used to provide the handler by name (and/or category)
- Unit Testing can be augmented by defining a Test class per Handler to validate (and provide code coverage)
- Easier to maintain (Single Responsibility)
- Ability to change behavior via inheritance (i.e. reuse) (Open-Closed Principle)
- Injection of a **Mediator** (see accc_TriggerMediator).

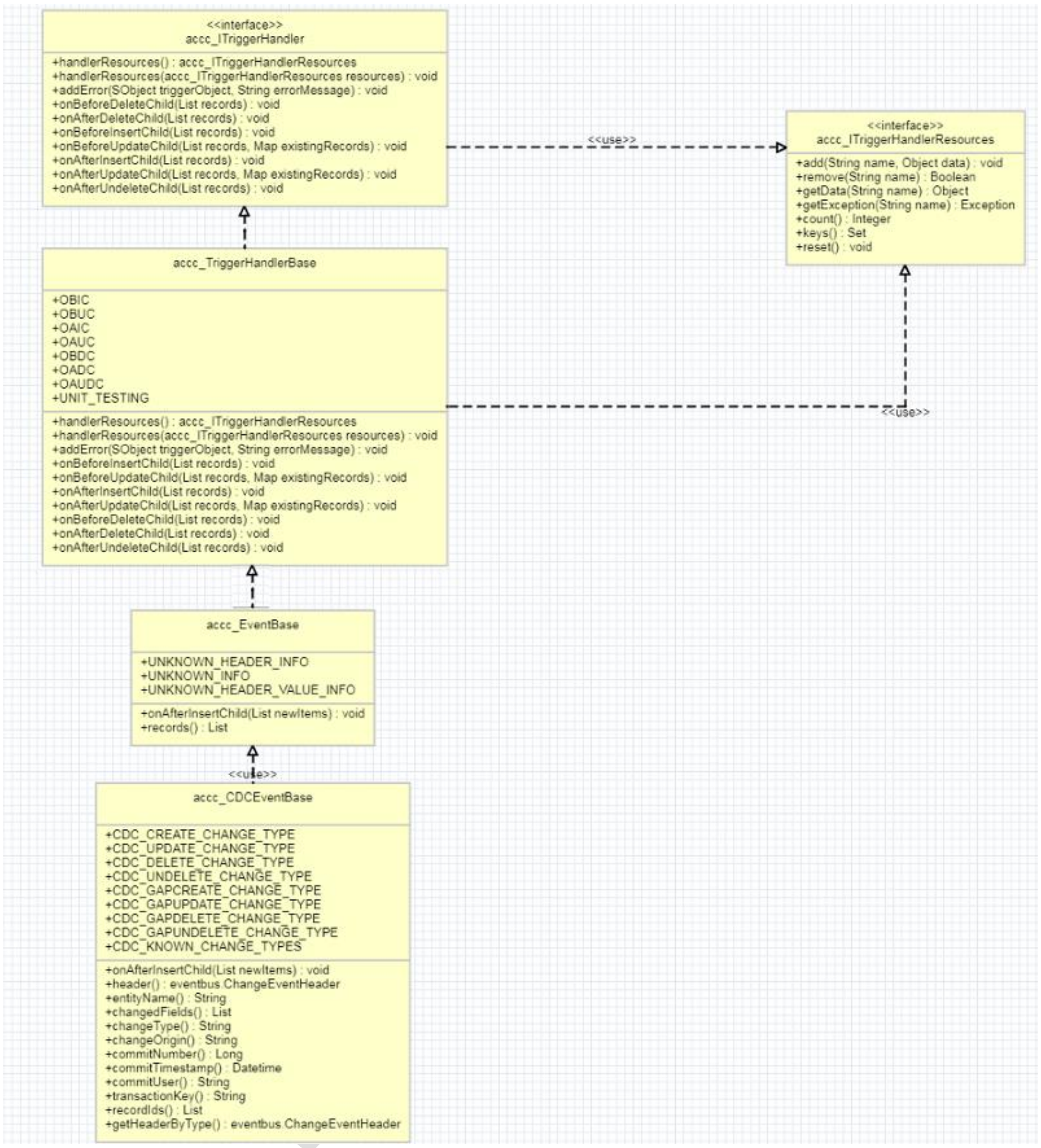

<<interface>>
accc_ITriggerHandler
+handlerResources() : accc_ITriggerHandlerResources
+handlerResources(accc_ITriggerHandlerResources resources) : void
+addError(SObject triggerObject, String errorMessage) : void
+onBeforeDeleteChild(List records) : void
+onAfterDeleteChild(List records) : void
+onBeforeInsertChild(List records) : void
+onBeforeUpdateChild(List records, Map existingRecords) : void
+onAfterInsertChild(List records) : void
+onAfterUpdateChild(List records, Map existingRecords) : void
+onAfterUndeleteChild(List records) : void
<<use>>
<<interface>>
accc_ITriggerHandlerResources
+add(String name, Object data) : void
+remove(String name) : Boolean
+getData(String name) : Object
+getException(String name) : Exception
+count() : Integer
+keys() : Set
+reset() : void
accc_TriggerHandlerBase
+OBIC
+OBUC
+OAIC
+OAUC
+OBDC
+OADC
+OAUDC
+UNIT_TESTING
+handlerResources() : accc_ITriggerHandlerResources
+handlerResources(accc_ITriggerHandlerResources resources) : void
+addError(SObject triggerObject, String errorMessage) : void
+onBeforeInsertChild(List records) : void
+onBeforeUpdateChild(List records, Map existingRecords) : void
+onAfterInsertChild(List records) : void
+onAfterUpdateChild(List records, Map existingRecords) : void
+onBeforeDeleteChild(List records) : void
+onAfterDeleteChild(List records) : void
+onAfterUndeleteChild(List records) : void
<<use>>
accc_EventBase
+UNKNOWN_HEADER_INFO
+UNKNOWN_INFO
+UNKNOWN_HEADER_VALUE_INFO
+onAfterInsertChild(List newItems) : void
+records() : List
<<use>>
accc_CDCEventBase
+CDC_CREATE_CHANGE_TYPE
+CDC_UPDATE_CHANGE_TYPE
+CDC_DELETE_CHANGE_TYPE
+CDC_UNDELETE_CHANGE_TYPE
+CDC_GAPCREATE_CHANGE_TYPE
+CDC_GAPUPDATE_CHANGE_TYPE
+CDC_GAPDELETE_CHANGE_TYPE
+CDC_GAPUNDELETE_CHANGE_TYPE
+CDC_KNOWN_CHANGE_TYPES
+onAfterInsertChild(List newItems) : void
+header() : eventbus.ChangeEventHeader
+entityName() : String
+changedFields() : List
+changeType() : String
+changeOrigin() : String
+commitNumber() : Long
+commitTimestamp() : Datetime
+commitUser() : String
+transactionKey() : String
+recordIds() : List
+getHeaderByType() : eventbus.ChangeEventHeader

Figure 3 Detail Static Class Diagram

Examples

Some of the examples provided are based on the [Apex Cross Cutting Concerns framework]() and the associated [trigger handler framework](). We walk-through each segment and how to wire the components together.

Please note, these are examples, and does not necessarily reflect the only way to craft the solution; nor, the best possible solution, as this will be dictated by user's ability, tools, requirements and overall understanding.

Standard and Custom Objects

Some companies use FFLIB for their domains; however, these examples, do not show that integration, though, it can, and has been, easily done.

The simple (contrived Use Case) is to log Account Names before they are created.

For this example, create trigger on Account (Before Inserted). We will use the Trigger Mediator (***accc_TriggerMediator***) which will read the register handlers from custom metadata. The trigger handler, *onBeforeInsertAccountsHandler,* would be registered in the custom metadata and invoked to handle the before insert event for Account. The three components are listed below.

Components

- **Trigger** – *OnBeforeInsertAccounts*

```
/* @description use the mediator to go find associated trigger handlers and invoke
*/
trigger OnBeforeInsertAccounts on Account (before insert) {
 (new accc_TriggerMediator(Account.class)).execute(accc_TriggerHelperClass.TriggerState.BeforeInsert);
} // end of  OnBeforeInsertAccounts
```

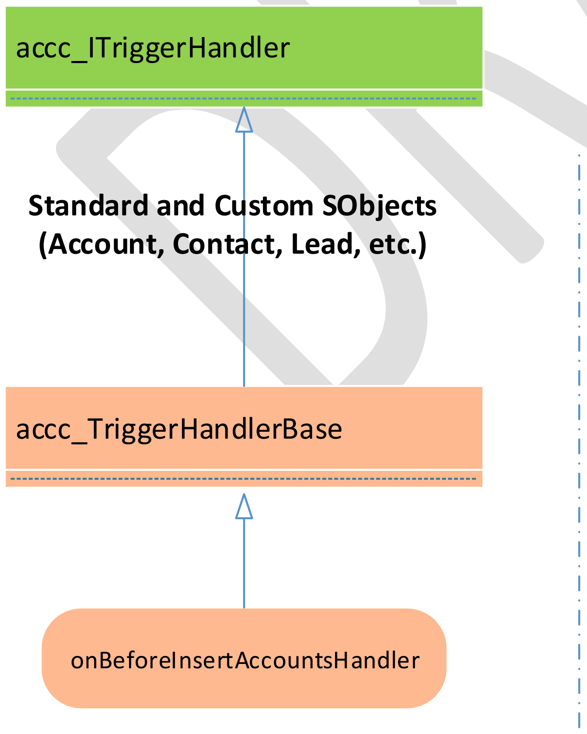
- **Event Dispatcher (see framework)**: *accc_TriggerMediator*[1]
- **Trigger Handler :** *onBeforeInsertAccountsHandler*

[1] You can find the code here.

```
public with sharing class onBeforeInsertAccountsHandler extends accc_TriggerHandlerBase {

  ////////////////////////////////////////////////////////////////////////
  //
  // We are ONLY overriding the one method. You override
  // any trigger event you are interested
  //
  ////////////////////////////////////////////////////////////////////////

  /**
   * @description On Before Insert - We override this to perform  processing
   * @param records the current records associated with the trigger event
   **/
  public override void onBeforeInsertChild(List<SObject> records) {
    for ( SObject record : records ){
          // :
    }
  } // end of onBeforeInsertChild

} // end of onBeforeInsertAccountsHandler
```

Breakdown of Solution

We breakdown the solution starting with the static class diagram. We are wanting to get a Trigger events on the Account object (namely*, before insert*); thus, we inherit from ***accc_TriggerHandlerBase***. Once we get the log event, we decompose it into a custom object and send it out to be written.


accc_ITriggerHandler
Standard and Custom SObjects
(Account, Contact, Lead, etc.)
accc_TriggerHandlerBase
onBeforeInsertAccountsHandler

The Trigger Mediator and the trigger components we are not delving into as it is not the focus of this document. However, as you can see, one ONLY has to override the associated method(s), in this example, we only overrode, **onBeforeInsertChild**. Please note, all trigger events can be handled but for brevity, only one was done.

Note, **accc_TriggerHandlerBase** supports all trigger before or after insert, update, delete, and undelete operations.

Platform Events

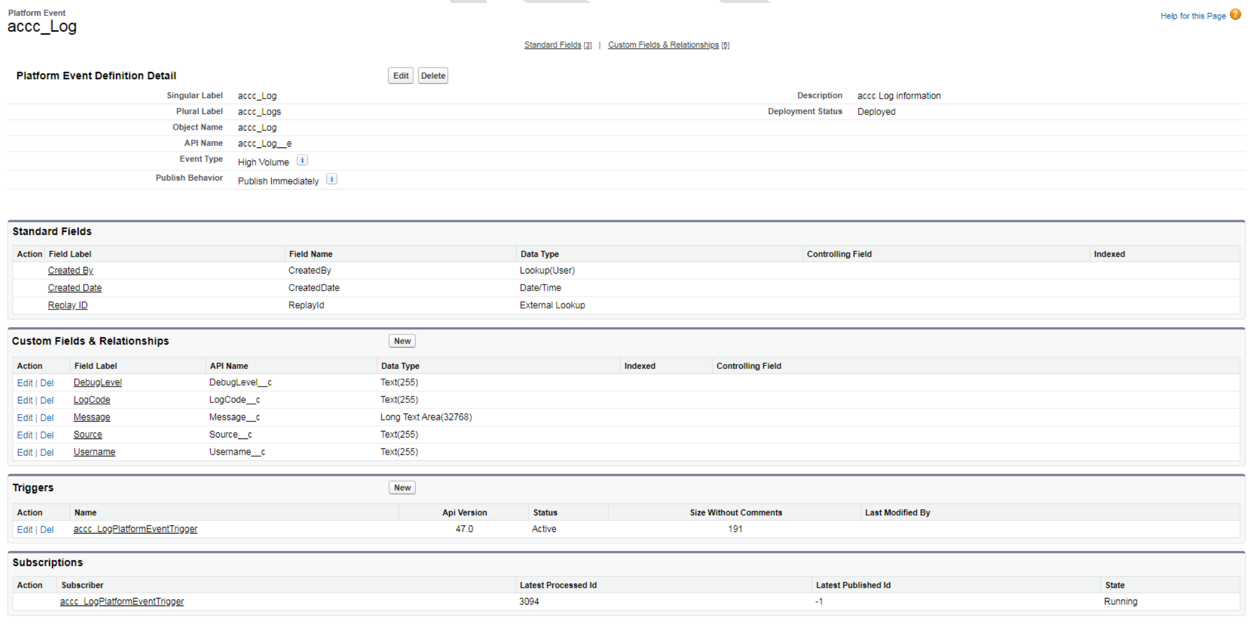
This example is actually implemented in the ACCC framework. The use case is as follows:

> As system administrator, I want all information run in Apex to be to logged into a custom object so that I can review any issues that may or may not have occurred.

This example used a simple Platform Event[2] (*accc_log__e*). In addition, we used a handler, a trigger and an event dispatcher (*accc_ApexPlatformEventDispatcher*).[3] The four components are listed below.

Components

- Platform Event - *accc_log__e:*



Platform Event
accc_Log

Help for this Page

Standard Fields [3] | Custom Fields & Relationships [5]

Platform Event Definition Detail

Singular Label	accc_Log	Description	accc Log information
Plural Label	accc_Logs	Deployment Status	Deployed
Object Name	accc_Log		
API Name	accc_Log__e		
Event Type	High Volume		
Publish Behavior	Publish Immediately		

Standard Fields

Action	Field Label	Field Name	Data Type	Controlling Field	Indexed
	Created By	CreatedBy	Lookup(User)		
	Created Date	CreatedDate	Date/Time		
	Replay ID	ReplayId	External Lookup		

Custom Fields & Relationships

Action	Field Label	API Name	Data Type	Indexed	Controlling Field
Edit | Del	DebugLevel	DebugLevel__c	Text(255)		
Edit | Del	LogCode	LogCode__c	Text(255)		
Edit | Del	Message	Message__c	Long Text Area(32768)		
Edit | Del	Source	Source__c	Text(255)		
Edit | Del	Username	Username__c	Text(255)		

Triggers

Action	Name	Api Version	Status	Size Without Comments	Last Modified By
Edit | Del	accc_LogPlatformEventTrigger	47.0	Active	191	

Subscriptions

Action	Subscriber	Latest Processed Id	Latest Published Id	State
	accc_LogPlatformEventTrigger	3094	-1	Running

- **Trigger** – *AcccLogEntryTrigger*

[2] This document does not walk through how to create a Platform Event in Salesforce

[3] All these items are available in the ACCC framework

```
/* @description if the site is configured to use accc_ApexPublishEventLogger,
 * log events are published and caught by this trigger. Upon arrival, the
 * accc_Log__e object is translated to a Application__c and inserted. Note, the
 * insert does not provide a UoW (Unit of Work Pattern) TBD.
 * @group Trigger Layer
 * @param handle insert of accc_Log__e events
 */
trigger accc_LogPlatformEventTrigger on accc_Log__e (after insert) {
        accc_ApexPlatformEventDispatcher.run(new accc_LogPlatformEventHandler());
} // end of accc_LogPlatformEventTrigger
```

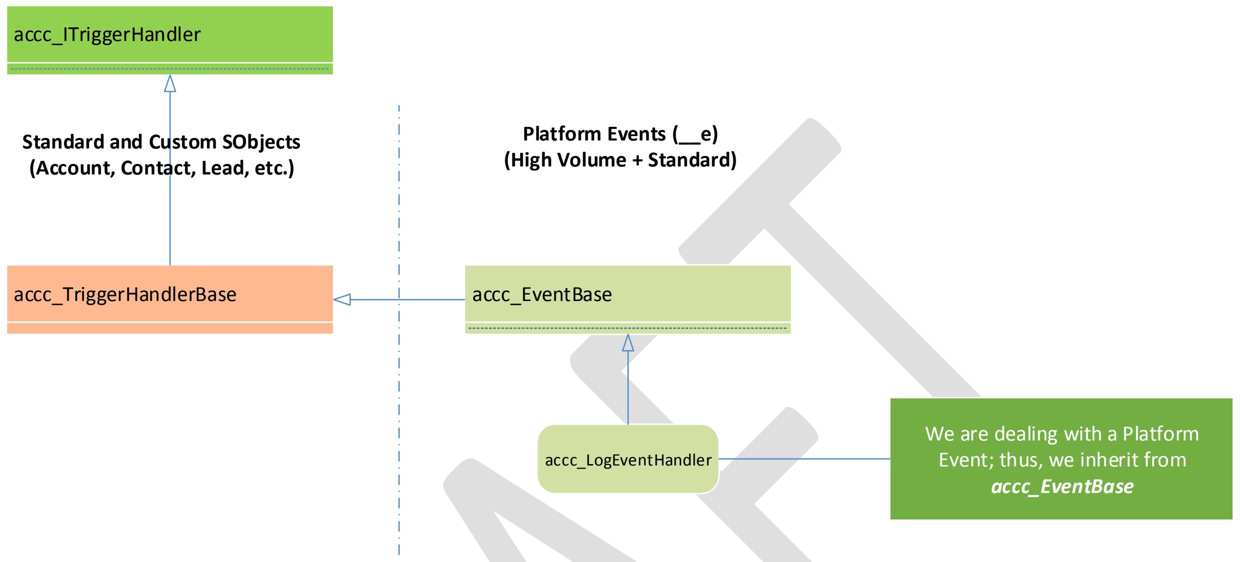
- **Event Dispatcher (see framework)**: *accc_ ApexPlatformEventDispatcher*[4]
- **Trigger Handler :** *accc_LogPlatformEventHandler*

```
public without sharing class accc_LogEventHandler extends accc_EventBase {
   /**
    * @description Capture the ACCC_Log_Event__e events -- only supports after insert
    *
    * @param List<SObject> list of new sobjects
    */
   public override void onAfterInsertChild(List<SObject> newItems) {
      try {

         List<ACCC_Log_Event__e> elogs = (List<ACCC_Log_Event__e>)newItems;
         List<AcccApplicationLog__c> theApplicationLogs =
accc_MessageTransformation.translate(elogs);
         // any data to add
         if ( theApplicationLogs != null && theApplicationLogs.size() > 0) {
            accc_SObjectBaseWriter writer = new accc_QueableSObjects(theApplicationLogs);
            writer.execute();
         }
      } catch (Exception excp) {
         // process exception, handle w/ care
         accc_ApexUtilities.log(excp);
      }
   } // end of onAfterInsertChild

} // end of accc_LogEventHandler
```

[4] You can find the code here.

Breakdown of Solution

We breakdown the solution starting with the static class diagram. We are listening for a Platform Event (namely, *accc_Log__e*); thus, we inherit from ***accc_EventBase***. Once we get the log event, we decompose it into a custom object and send it out to be written.


accc_ITriggerHandler
Standard and Custom SObjects (Account, Contact, Lead, etc.)
Platform Events (__e) (High Volume + Standard)
accc_TriggerHandlerBase
accc_EventBase
accc_LogEventHandler
We are dealing with a Platform Event; thus, we inherit from accc_EventBase

The dispatcher and the trigger components we are not delved into as it is not the focus of this document. However, as you can see, one ONLY has to override the **onAfterInsertChild** method.

Change Data Capture (CDC)

For this example, anytime a Account Name is ***updated*** we would like to record the operation. CDC handling is similar to High Volume Platform Events, though, other elements come into play. This document does not cover all the intricacies of CDC; the reader should take the initiative to understand this here.

Because the functionality is similar to High Volume Platform Events we will have many of the same components (though, a different inheritance scheme) but no user-defined Platform Event.

However, we will need a *handler*, a *trigger* and an *event* dispatcher (*accc_ApexPlatformEventDispatcher*).[5] The four components are listed below.

Components

- **Standard Platform Event** - *AccountChangeEvent*
- **Trigger** – *accc_cdcAccountHandler*

```
trigger accc_cdcAccountHandler on AccountChangeEvent (after insert) {
  // if you are using the accc_TriggerMediator (hook it up with you domain)
  accc_ApexPlatformEventDispatcher.run(AccountChangeEvent.class );
}
```

- **Event Dispatcher (see framework)**: *accc_ ApexPlatformEventDispatcher*[6]

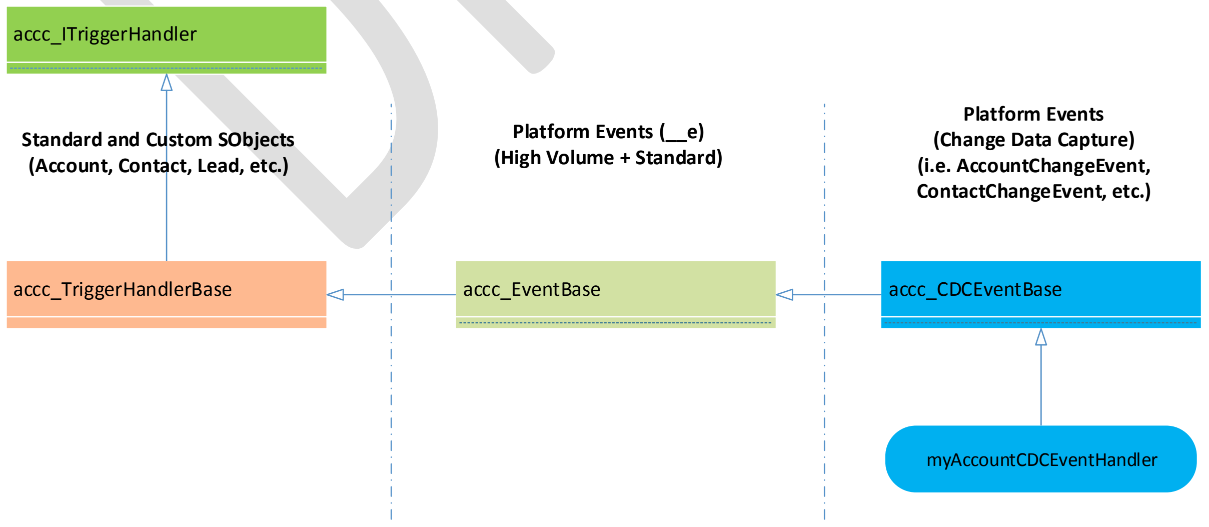
[5] All these items are available in the ACCC framework
[6] You can find the code here.

- **Trigger Handler :** *myAccountCDCEventHandler*

```
public without sharing class myAccountCDCEventHandler extends accc_CDCEventBase {

    /**
    * @description child handle update ( we ONLY CARED about UPDATEs)
    * @param SObject event
    * @return void
    */
   @TestVisible
   protected override void handleUpdate(SObject event){
      Boolean contains = this.getFieldsChangedInUpdate(event).containsKey('Name');
      if ( contains ) {
       accc_CDCEventBase.FieldUpdateState state =
this.getFieldsChangedInUpdate(event).get('Name');
       if (  state == accc_CDCEventBase.FieldUpdateState.UPDATED ){
            // Name has been updated ( you can pull off the event)
            // :
       }
      }

   }// end of handleUpdate
} // end of myAccountCDCEventHandler
```

Breakdown of Solution

We breakdown the solution starting with the static class diagram. We are wanting to get a CDC Platform Event (namely, *AccountChangeEvent*) when an Account Name changes; thus, we inherit from ***accc_CDCEventBase***. Once we get the CDC event, the platform event dispatcher (using Trigger Mediator) looks up the associated handler for *Account* and *AccountChangeEvent* and invokes, ***myAccountCDCEventHandler***.



The dispatcher and the trigger components we are not delving into as it is not the focus of this document. However, as you can see, one ONLY has to override the **handleUpdate** method. The CDC Base class, provides similar hooks into create, delete, undelete as well as GAP creates, updates, deletes and undeletes.

Where event, is an SObject (you can cast it accordingly, or use generic gets to retrieve data):

- handleCreate(event);
- handleUpdate(event);
- handleDelete(event);
- handleUnDelete(event);
- handleGAPCreate(event);
- handleGAPUpdate(event);
- handleGAPDelete(event);
- handleGAPUndelete(event);

Note, if you want a more granular control of CDC events, the developer has two choices:

1. Override - **public virtual void onAfterInsertChild(List<Sobject> records)**, or
2. Override - **protected virtual Boolean handleEventMethod(SObject event)**

Only chose to override these methods if you have **a firm understanding of the intricacies of CDC Events** as the above two methods give you greater control.

Summary

Some of the examples were contrived and may not represent the optimal code nor proper exception handling features (as that is not the purpose). Instead, the goal was to break down the class hierarchy and functionality found in the different base classes and provide a simple walk-through of components.