Salesforce Trigger Handler Management

Overview

The diagram below shows the Trigger Handling within the Salesforce environment. The basic premise is to incorporate various handlers based on the domains (i.e. *Account*, *Contact*, *Lead*, etc.). The handlers will look for the domain trigger handlers via the configuration information (contained in the custom metadata) based on their respective environments (*Test*, *Debug*, *Production*).

As longs as developers follow the design below new trigger handling domains can easily be injected in the environment with the ability to test and design without a trigger.

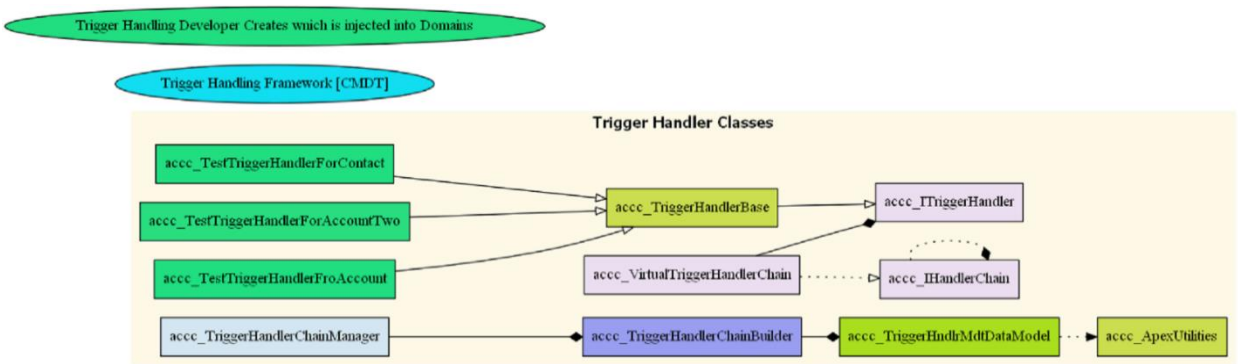
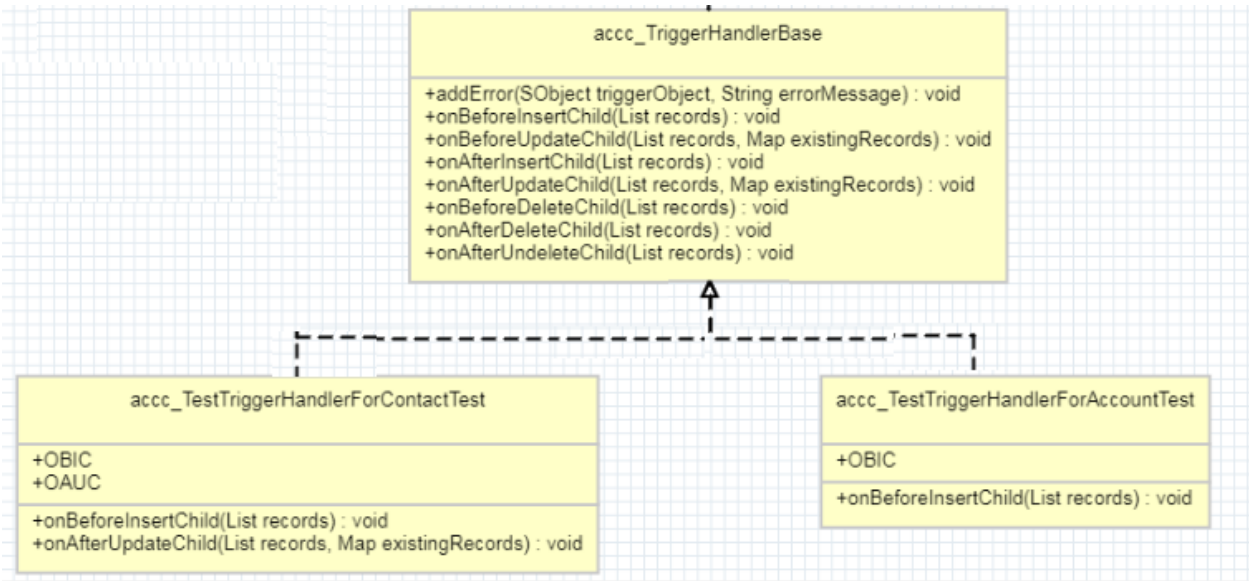


Figure 1 Static Class Diagram

For example, the two classes, ***accc_TestTriggerHandlerForContactTest*** and ***accc_TestTriggerHandlerForAccountTest*** below inherit from **accc_TriggerHandlerBase.** All developers writing trigger handlers must inherit from **accc_TriggerHandlerBase[1].**



[1] These classes are just used for testing but provide a prototypical form of trigger handler

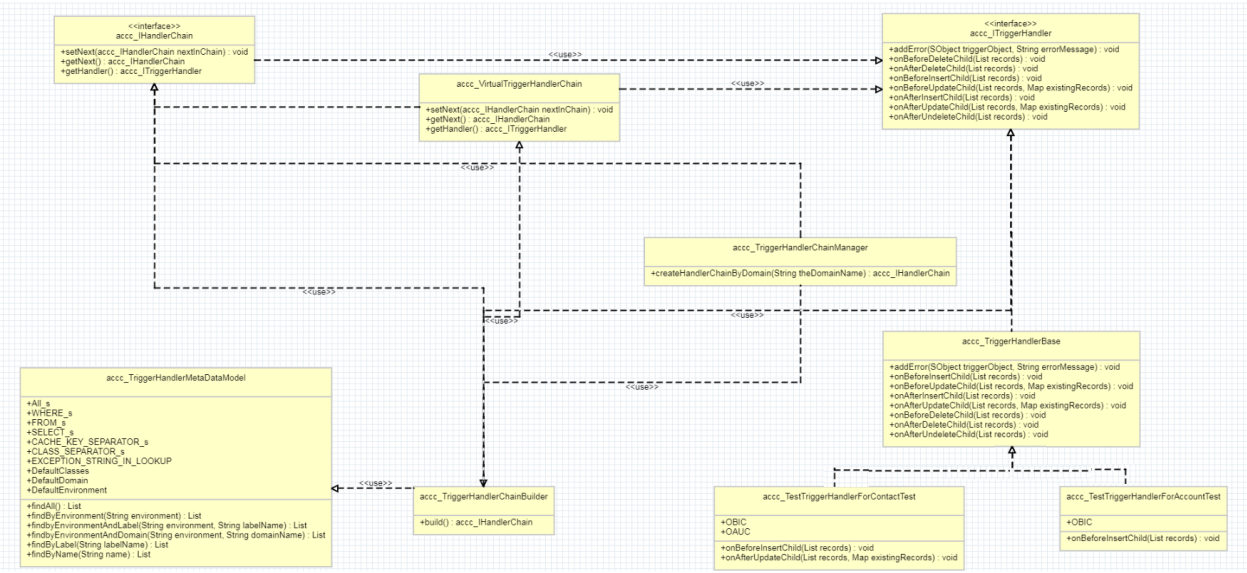


Figure 2 Overall UML Static Diagram of the Trigger Injection mechanism

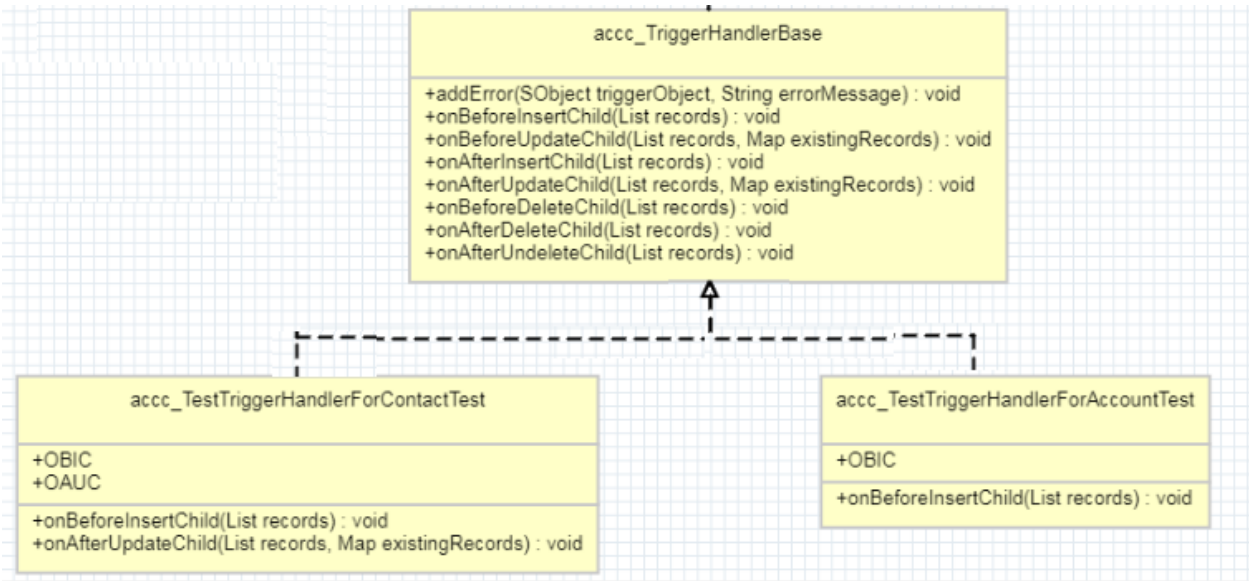
Design Patterns

Chain of Responsibility is used to manage the trigger handlers that are controlled by the base Domain (ie. *wf_DomainBase*). This provides a sequential processing of the trigger handlers from the base class. There is no need to modify the child domain classes (i.e. *wf_Accounts, wf_Contacts*, etc.) unless the child **DOES NOT** want to participate in the configured trigger handling. These handlers are injected from custom metadata; configured at runtime and processed in the order listed. The **Builder,** *accc_TriggerHandlerChainBuilder***,** is used by the **Mediator**, *accc_TriggerChainManager*, to build the components associated with the Trigger handling mechanism. The Builder will pull information from the custom metadata model, *Trigger_Handler_Binding__mdt*. In addition, the trigger handlers may throw an exception which will be caught within *wf_Accounts.* There is flag that will either allow the trigger handler process to continue or to abort the rest of the handlers

Responsibilities of Developers

The developers/architects will be responsible for writing the concrete class, I.e. *<Prefix>_TriggerHandlerAccountChain, <Prefix>_TriggerHandlerContactChain,* etc. which should inherit (extends) from *accc_TriggerHandlerBase.* As a suggestion, the concrete classes could have a three letter prefix, which provides enough uniqueness for new handlers. For example, new trigger handlers for the contact domain, could be, **mdm1_ TriggerHandlerContactChain** and **mdm2_ TriggerHandlerContactChain.** All the developer would need to do is write their logic for their class by overriding only the appropriate trigger event method. The numbers in the prefix indicate the ordering; thus, *mdm1_* would be executed before *mdm2_*.

The above classes would ALL inherit from *accc_TriggerHandlerBase*. The example below shows how two test classes were used to validate the trigger handler behavior. For example, in the diagram below the two classes inherit from *accc_TriggerHandlerBase*. The class *accc_TestTriggerHandlerForAccountTest*, overrides the ***onBeforeInsert*** method.



Custom Metadata Updates

The custom metadata type, *accc_Trigger_Handler_MetaData_Model__mdt*, shows the information required to inject new trigger handlers into user's core.

Custom Metadata Type
Trigger Handler Binding

Help for this Page

Standard Fields [6] | Custom Fields [7] | Validation Rules [0] | Page Layouts [1]

Custom Metadata Type Detail Edit Delete Manage Trigger Handler Bindings

Singular Label	Trigger Handler Binding	Description	Trigger Handler Binding
Plural Label	Trigger Handler Bindings	Visibility	Public
Object Name	Trigger_Handler_Bindings	Protection Level	
API Name	Trigger_Handler_Bindings__mdt	Record Size	464

Standard Fields

Action	Field Label	Field Name	Data Type	Indexed
	Created By	CreatedBy	Lookup(User)	
Edit	Custom Metadata Record Name	DeveloperName	Text(40)	
Edit	Label	MasterLabel	Text(40)	
	Last Modified By	LastModifiedBy	Lookup(User)	
Edit	Namespace Prefix	NamespacePrefix	Text	
Edit	Protected Component	IsProtected	Checkbox	

Custom Fields New

Action	Field Label	API Name	Data Type	Field Manageability	Indexed	Controlling Field	Modified By
Edit | Del	Active	Active__c	Checkbox	Upgradable			Bill Anderson, 8/30/2020, 10:24 AM
Edit | Del	Continue If Exception	Continue_If_Exception__c	Checkbox	Upgradable			Bill Anderson, 8/30/2020, 10:25 AM
Edit | Del	Domain	Domain__c	Metadata Relationship(Entity Definition)	Upgradable			Bill Anderson, 8/30/2020, 10:21 AM
Edit | Del | Replace	Environment	Environment__c	Picklist	Upgradable			Bill Anderson, 8/30/2020, 10:24 AM
Edit | Del	Order	Order__c	Number(18, 0)	Upgradable			Bill Anderson, 8/30/2020, 11:08 AM
Edit | Del	Performance Metrics	Performance_Metrics__c	Checkbox	Upgradable			Bill Anderson, 8/30/2020, 10:25 AM
Edit | Del	Trigger Handler Class	Trigger_Handler_Class__c	Text Area(255)	Upgradable			Bill Anderson, 8/30/2020, 10:26 AM

Figure 3 Custom Metadata for Trigger Handling Injection

There are already two entries, for Account and Contact, which are used for testing; as noted by the Environment. These two entries must be left in for testing and as new trigger handlers are introduced you should follow the same format. It should be noted that the framework checks for duplicate handlers.

Trigger Handler Bindings

New

Order	Label	Namespace Prefix	Trigger Handler Binding Name	Domain	Environment	Trigger Handler Class	Active	Continue If Exception	Performance Metrics
2	Leave For Testing Account Handler		Leave_For_Testing_Account_Handler	Account	Test	accc_TestTriggerHandlerForAccountTwo	✓	✓	✓
1	Leave For Testing Account One Handler		Leave_For_Testing_Account_One_Handler	Account	Test	accc_TestTriggerHandlerForAccount	✓	✓	✓
1	Leave For Testing Contact Handler		Leave_For_Testing_Contact_Handler	Contact	Test	accc_TestTriggerHandlerForContact	✓	✓	✓

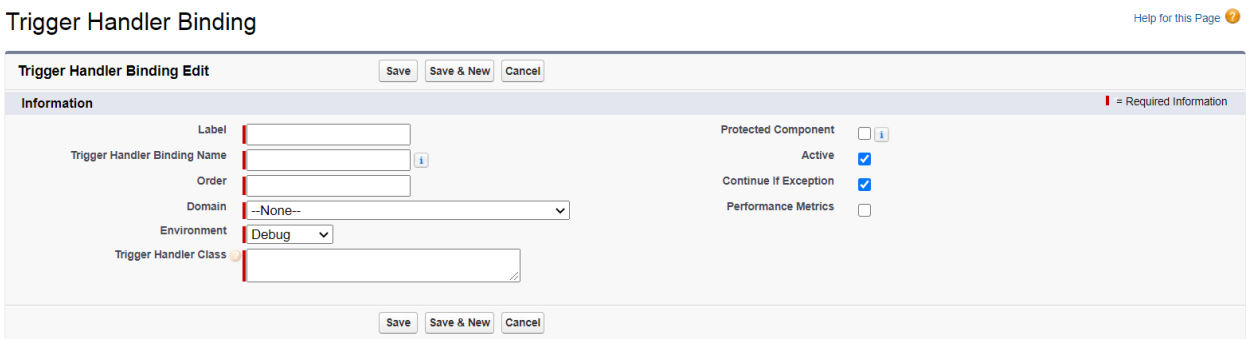
Figure 4 Used for Testing the Trigger Handler framework

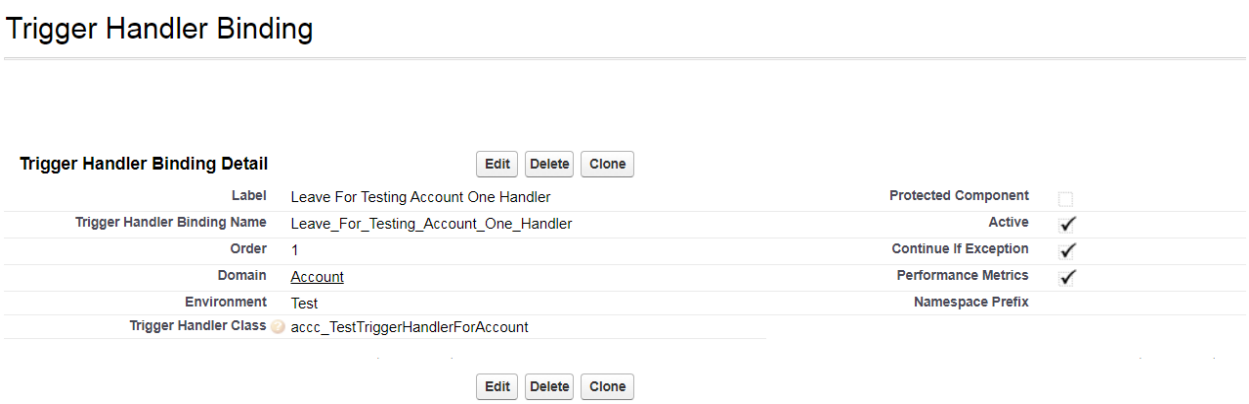
Creating a new entry

Below represents the information required to have a trigger handler injected into the core.

- **Label and Name** are unique names across the custom metadata model for the trigger handlers.
- **Environment** represents where to use the trigger handler (Test, Production, Debug).
- **Domain** represents the Sobject Name (Standard or Custom); i.e. Account, Affiliation__c, Contact, etc.
- **Trigger Class** represents the handler to be invoked based on the trigger event. User needs to ensure they inherit from ***accc_TriggerHandlerBase*** *and override the appropriate methods (trigger events).*
- **Order** indicates precedence (i.e. the sequence of handler invocation)
- **Active** indicates if the trigger handlers are invokable

- **Continue If Exception** continue to the next handler in the event of an exception.
- **Performance Metrics** gathers performance metrics on each handler



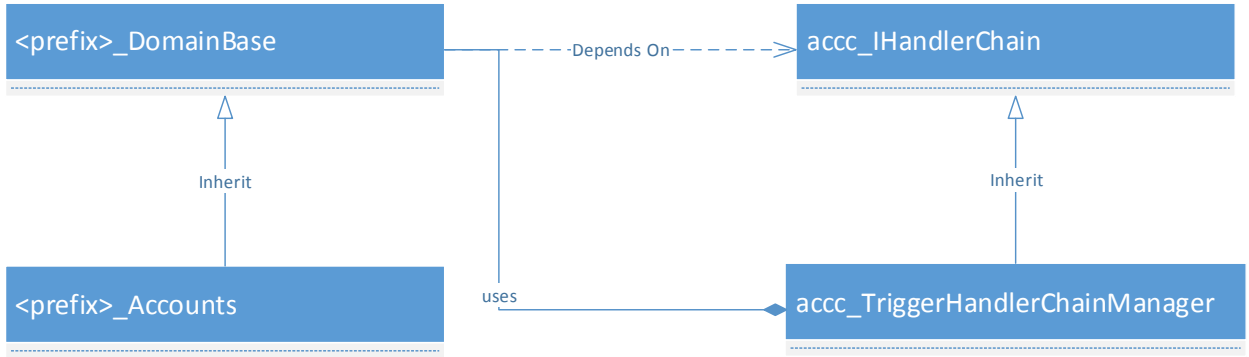


Testing Trigger Handlers – Developers

The custom metadata provides developers the ability to test and debug their handlers without injecting into the production environment. In production, ONLY trigger handlers marked in the **environment** as *production* and marked as **active** will be injected into the core for execution.

Integration into Current Architecture

The current architecture can be updated without any modifications needed in the Domain classes (*wf_Accounts*, etc. [*except for removal of previous handlers*]) as the driver for invoking the trigger handler classes will be done in the base class, ***wf_DomainBase*.**



The above diagram has the base class, ***wf_DomainBase***, which invokes the appropriate handlers per trigger event. For example, the before insert event, would be invoked from the base class. Instead of calling down to the child's handler (which it can), we can now pull the Trigger Handlers from the custom metadata and call (in order) all associated events for that domain (note exception handling removed for readability.

```
/**
 * Override this to perform child processing, if needed; however,
 * this will not execute the chain of handlers
 **/
public virtual void onBeforeInsertChild() {
        if (System.Test.isRunningTest() ) {
                this.log('+++++++++++++++onBeforeInsertChild - Entered / Exit ');
        }
        // get the chain of trigger handlers
        accc_IHandlerChain chain = this.theTriggerChain();

        // iterate over the trigger handler chain
        this.triggerHandlerInvocation(chain,
                        accc_TriggerHelperClass.TriggerState.BeforeInsert,
                        null,
                        this.theTriggerManager().continueIfException,
                        this.theTriggerManager().performanceMetrics
                        );
}// end of onBeforeInsertChild
```

Figure 5 wf_DomainBase's onBeforeInsert

The above will be replaced in ALL the trigger events (before/after) within the base domain class. After the change you can add/remove trigger handler classes in the custom metadata as the builder will pull from there. This gives the administrator the ability to control what gets brought into the run-time.

Salient Classes

The salient classes in the design have the following responsibilities.

Class	Responsibility
accc_ITriggerHandlerChain	Defines trigger events
accc_IHandlerChain	Defines get/set of the chain and the getHandler
accc_VirtualTriggerHandlerChain	Links the Trigger Handler and Chaining together
accc_TriggerHandlerChainBuilder	Builds the respective handlers
accc_TriggerHandlerChainManager	Orchestrates building and invoking
accc_TriggerHandlerMetaDataModel	Models the custom metadata in Salesforce (ViewModel)
accc_TriggerHandlerBase	Defines the base class developers inherit (and override)
<prefix>_Accounts	User defined classes that inherit **accc_TriggerHandlerBase**

How to add new Trigger Handlers

All new trigger handlers must perform at minimum three steps.

Step 1: Create a class that inherits from **accc_TriggerHandlerBase,**

Step 2: Override the only trigger methods *as needed* (two were overridden),

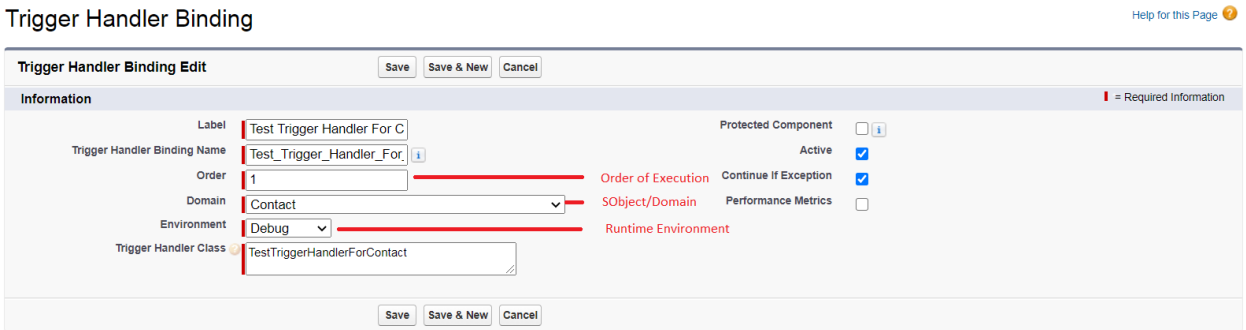
```
// STEP 1 ----------------------------------------------------------------------→ inherit
public with sharing class TestTriggerHandlerForContact extends accc_TriggerHandlerBase {

        ////////////////////////////////////////////////////////////////////
        //
        // We are ONLY overriding the one method for testing. You override
        // any trigger event you are interested ( and NO MORE)
        //
        ////////////////////////////////////////////////////////////////////

        /** STEP 2 : Override method
         * @description On Before Insert - We override this to perform  processing
         * @param records the current records associated with the event
         **/
        public override void onBeforeInsertChild(List<SObject> records) {
                if ( Test.isRunningTest()) {
                        accc_ApexUtilities.log('TestTriggerHandlerForContact on Before Insert');
                }
        } // end of onBeforeInsertChild
        /** STEP 2 : Override method
         * @description On After Update - Override this to perform  processing
         * @param records the current records associated with the event
         * @param existingRecords the old records associated with the event
         **/
        public override void onAfterUpdateChild(List<SObject> records, Map<Id, SObject> existingRecords) {
                if ( Test.isRunningTest()) {
                        accc_ApexUtilities.log('TestTriggerHandlerForContact on After Update);
                }
        }// end of onAfterUpdateChild

} // end of accc_TestTriggerHandlerForContact
```

Step 3: Add class name, **TestTriggerHandlerForContact**, to custom metadata, **accc_Trigger_Handler_Binding__mdt**



Trigger Handler Base Class

Developers' trigger handler will inherit from **accc_TriggerHandlerBase**. All the methods are shown in the class below.

Part 1 - accc_TriggerHandlerBase

```
public virtual with sharing class accc_TriggerHandlerBase implements accc_ITrigg

////////////////////////////////////////////////////////////////////////////////
/// Data Members
////////////////////////////////////////////////////////////////////////////////
@TestVisible
private Object m_parameters = null;
////////////////////////////////////////////////////////////////////////////////
/// Ctors
////////////////////////////////////////////////////////////////////////////////
/**
 * @description default ctor
 *
 */
public accc_TriggerHandlerBase() {
  this(null);
} // end of ctor
/**
 * @description ctor
 *
 * @param parameters information for the children
 */
public accc_TriggerHandlerBase(Object parameters) {
        this.theParameters = parameters;
} // end of ctor
////////////////////////////////////////////////////////////////////
/// Properties
////////////////////////////////////////////////////////////////////
/**
 * The paramters for this handler
 */
@TestVisible
protected Object theParameters {
        get { return this.m_parameters;}
        set { this.m_parameters = value;}
} // end of theParameters

////////////////////////////////////////////////////////////////////
/// Public Methods
////////////////////////////////////////////////////////////////////
/**
 * @description add an error on the sobject. When used on Trigger.new
 * in before insert and before update triggers, and on Trigger.old in
 * before delete triggers, the error message is displayed in the application
 * interface.
 *
 * @param triggerObject the salesforce object (from the trigger invocation) to se
 *      the error message
 * @param errorMessage error message
 */
public virtual void addError(SObject triggerObject, String errorMessage) {
  if ( triggerObject != null
      && !string.isBlank(errorMessage)
      && (!System.isBatch() || !System.isFuture() || System.isScheduled()) ) {
        //try {
        triggerObject.addError(errorMessage);
        //} catch (Exception) {
        // TBD -- ensure it is on a trigger sobject
        //}
  }
} // end of addError
/**
```

Part 2 - accc_TriggerHandlerBase

```
/**
 * @description on Before Update - Override this to perform  processing
 * @param records the current records associated with the event
 * @param existingRecords the old records associated with the event
 **/
public virtual void onBeforeUpdateChild(List<SObject> records,
                        Map<Id, SObject> existingRecords) {
  if ( Test.isRunningTest()) {
      accc_ApexUtilities.log('+++++++++++++++On Before Update' );
  }
}// end of onBeforeUpdateChild
/**
 * @description On After Insert - Override this to perform  processing
 * @param records the current records associated with the event
 **/
public virtual void onAfterInsertChild(List<SObject> records) {
  if ( Test.isRunningTest()) {
      accc_ApexUtilities.log('+++++++++++++++On After Insert' );
  }
}// end of onAfterInsertChild
/**
 * @description On After Update - Override this to perform  processing
 * @param records the current records associated with the event
 * @param existingRecords the old records associated with the event
 **/
public virtual void onAfterUpdateChild(List<SObject> records,
        Map<Id, SObject> existingRecords) {
  if ( Test.isRunningTest()) {
        accc_ApexUtilities.log('+++++++++++++++On After Update' );
  }
}// end of onAfterUpdateChild
/**
 * @description On Before Delete - Override this to perform  processing
 * @param records the current records associated with the event
 *
 **/
public virtual void onBeforeDeleteChild(List<SObject> records) {
  if ( Test.isRunningTest()) {
        accc_ApexUtilities.log('+++++++++++++++On Before Delete' );
  }
}// end of onBeforeDeleteChild
/**
 * @description On After Delete - Override this to perform  processing
 * @param records the current records associated with the event
 *
 **/
public virtual void onAfterDeleteChild(List<SObject> records) {
  if ( Test.isRunningTest()) {
        accc_ApexUtilities.log('+++++++++++++++On After Delete' );
  }
}// end of onAfterDeleteChild
/**
 * @description On After Undelete - Override this to perform  processing
 * @param records the current records associated with the event
 **/
public virtual void onAfterUndeleteChild(List<SObject> records) {
  if ( Test.isRunningTest()) {
        accc_ApexUtilities.log('+++++++++++++++On After UnDelete' );
  }
 } // end of onAfterUndeleteChild


} // end of accc_TriggerHandlerBase
```

Summary

This example uses the interfaces and base classes as defined in ACCC. You are not bound to these interfaces. Instead, you can have the ***<prefix>_BaseDomain*** call your specific interface.

Appendix: Simple Flow