Platform **E**vent **W**rapper

About the Author

I have been writing software since High School. I have contributed to Open Source since 1990's in UNIX and Windows environment (though, under my company name). I have been contributing to Salesforce Apex since 2017 in an attempt to provide consistent, reliable, reusable and flexible Apex Code.

What is it?

This unmanaged package wraps *publish* and *subscribe* mechanism in Salesforce. It provides the ability change behavior at runtime, via Dependency Injection, as well as changing behavior via extensions.

What is the value?

Without some framework, or extensible tools, platform events are piecemealed, forgotten, or a mishmash of incongruous parts. It provides a consistent and manageable control of publishing and consuming platform events. The consumer of the framework does not worry about the management of Platform Events, rather the business logic related to the event.

The value lies in being consistent, reliable, reusable, flexible and agnostic to the underlying pinning of Platform Events .

How does it works?

Defining a set of common interfaces and custom metadata within the Platform Event framework allows one to change/augment aspects at different levels/granularity. First, let's define some of the salient components and their functions before we walk through a code-snippet.

Salient Components

The Platform Event Wrapper provides six basic components:

Component	Function
accc_IEventHandler	Defines the behavior for a Publisher or Consumer
accc_IPlatformEventModel	Is a container for handling publish or subscription service
accc_PlatformEvtBuilder	Builds the model based on custom metadata, if defined, or uses defaults
accc_IProcessEventHandlers	Container of handlers (log, success, error, alert)
accc_PlatformEventAttrs	Attributes used to manage logging (high-volume or standard), alerts, retries, validations, etc. of the publish/consume process
accc_PlatformEvtMdtDataModel	Provides a wrapper around the custom-metadata (DAO, Data Access Object), ***accc_Platform_Event_Binding__mdt***

A static class diagram brings this into more clarity.

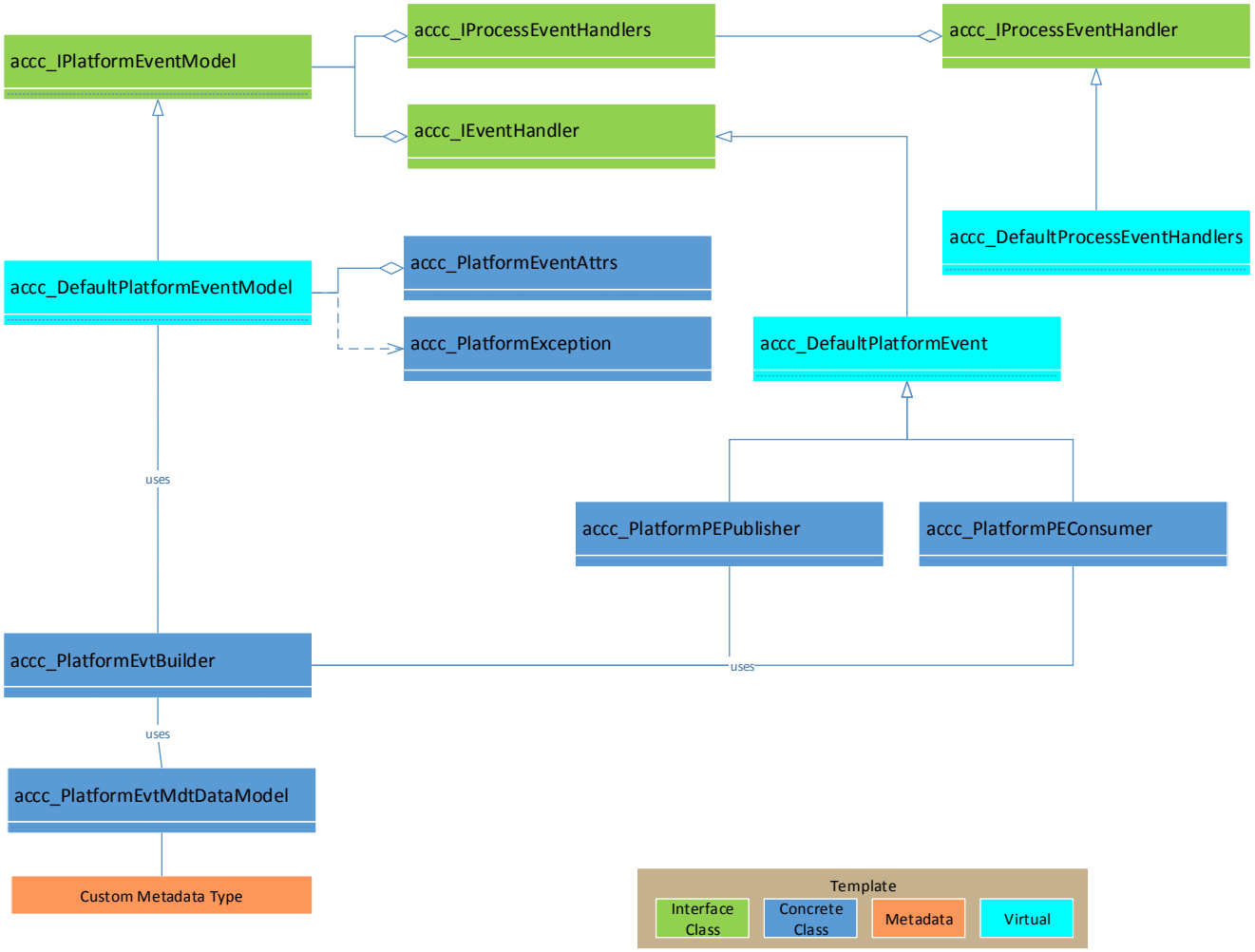


Figure 1 Static Class Diagram

Platform Event Model

The Platform Event Model, *accc_IPlatformEventModel*, defines the behavior for processing platform events for publish or consumption.

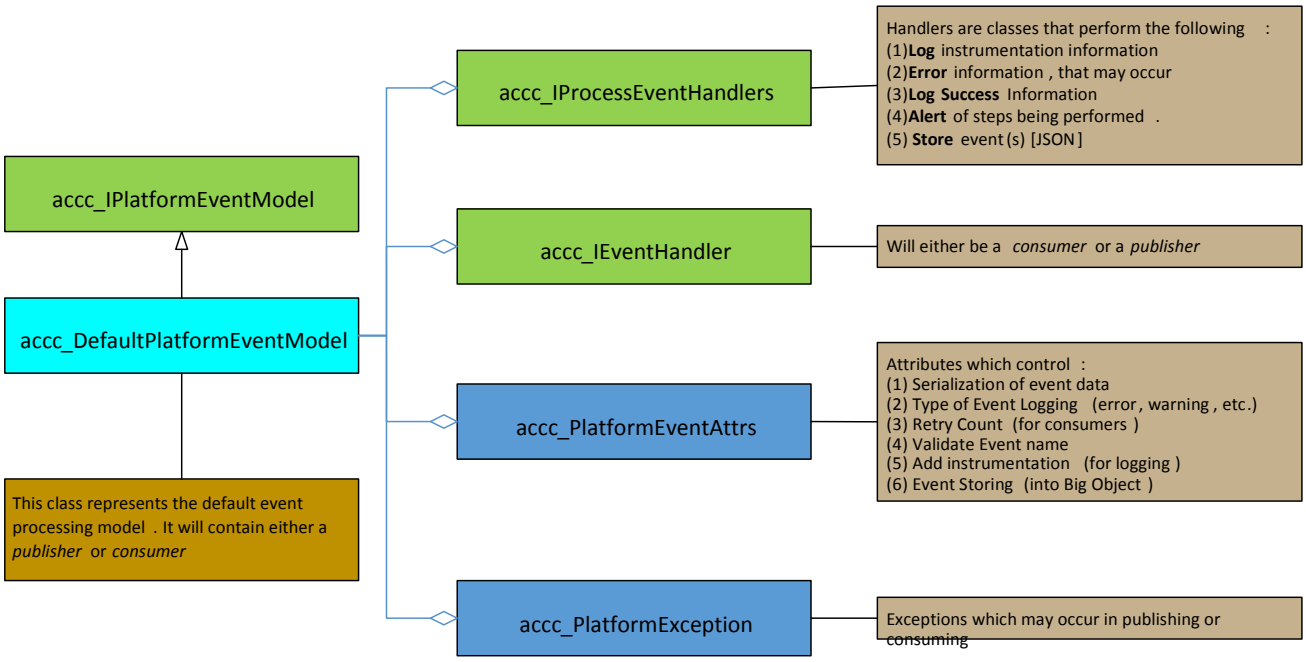


Figure 2 Platform Event Model

Code Snippet / Example for **Publishing**

This code snippet, publishes an event, *pe_test__e*.

```
// [1] create default attributes -- optional accc_PlatformEventAttrs attributes
= new accc_PlatformEventAttrs();
// [2] create platform event builder, platform event name and the runtime environment (‘test’,’debug’prod’)
accc_PlatformEvtBuilder builder = new accc_PlatformEvtBuilder(‘pe_test__e','test');
// [3] create the default publisher  accc_IEventHandler
publisher = builder.buildPublisher();
// [4] create event model accc_IPlatformEventModel model = builder.build(publisher); // or
builder.build(publisher,attributes);
// [5] create event to publish (note, the name must match that which was passed to the builder)
List<pe_test__e> data = new List<pe_test__e> { new pe_test__e () };
// [6] process/publish the event (returns true, if processed successfully)
System.debug('++++ result =' + model.process(data));
```

Figure 3 Publish an event

Steps 1 – 6 are described as follows (Step 1, is optional),

	Code	Comment
1	**accc_PlatformEventAttrs** attributes = new accc_PlatformEventAttrs();	Create the default attributes (Optional). See Platform Attributes

2	**accc_PlatformEvtBuilder** builder = new accc_PlatformEvtBuilder('pe_test__e','test');	Create a platform event builder. The event name, '*pe_test__e',* along with the environment, 'test', is looked up in the custom metadata. The custom metadata may contain, five handlers. Handlers are classes that perform the following: (1) **Log** instrumentation information (2) **Error** information, that may occur (3) **Log Success** Information (4) **Alert** of steps being performed. (5) **Store** event(s) [JSON]
3	**accc_IEventHandler** publisher = builder.buildPublisher();	From step 2, the builder knows if there are any special handlers (other than the default) defined in the custom metadata. Publisher follow the process log, check, alert and publish.
4	**accc_IPlatformEventModel** model = builder.build(publisher);	Now we can build the model. The model holds the five handlers, the event handler (publisher) and attributes that control the behavior.
5	**List<pe_test__e>** pe=new List<pe_test__e> {new pe_test__e ()};	Create the collection of events to publish
6	**model.process(pe)**);	Publish the event, returning true, if successful; otherwise false. If false, check the **model.getException().**

Code Snippet / Example for **Consuming**

The sample code would be consumed in a Trigger Handler class or one invoked from an after insert platform event trigger. This code snippet, consumes an event, *pe_test__e*.

```
// [1] create default attributes -- optional accc_PlatformEventAttrs attributes
= new accc_PlatformEventAttrs();
// [2] create platform event builder, platform event name and the runtime environment ('test','debug'prod')
accc_PlatformEvtBuilder builder = new accc_PlatformEvtBuilder('pe_test__e','test');
// [3] create the default consumer accc_IEventHandler consumer =
builder.buildConsumer();
// [4] create event model accc_IPlatformEventModel model = builder.build(consumer); // or
builder.build(consumer,attributes);
// [5] process/consume the event (returns true, if processed successfully)
If ( model.process(Trigger.New) == false ) {
    System.debug('++++ result =' + model.getException());
} else {
    System.debug('++++ Success');
}
```

Figure 4 Consume an event

Steps 1 – 5 are described as follows (Step 1, is optional),

	Code	Comment
1	**accc_PlatformEventAttrs** attributes = new accc_PlatformEventAttrs();	Create the default attributes (optional). See Platform Attributes
2	**accc_PlatformEvtBuilder** builder = new accc_PlatformEvtBuilder('pe_test__e','test');	Create a platform event builder. The event name, *'pe_test__e',* along with the environment, 'test', is looked up in the custom metadata. The custom metadata may contain, five handlers. Handlers are classes that perform the following: (1) **Log** instrumentation information (2) **Error** information, that may occur (3) **Log Success** Information (4) **Alert** of steps being performed. (5) **Store** event(s) [JSON]
3	**accc_IEventHandler** consumer = builder.buildConsumer();	From step 2, the builder knows if there are any special handlers (other than the default) defined in the custom metadata. Consumers follow the process log, check, alert and publish.
4	**accc_IPlatformEventModel** model = builder.build(consumer);	Now, we can build the model. The model holds the five handlers, the event handler (publisher) and attributes that control the behavior.
5	**model.process(Trigger.new)**);	Consume the event, returning true, if successful; otherwise false. If false, check the **model.getException().**

Special notes about the consumer

Consumers have the ability to retry. According to Salesforce documentation:

> *Get another chance to process event notifications. Retrying a trigger is helpful when a transient error occurs or when waiting for a condition to change. Retry a trigger if the error or condition is external to the event records and is likely to go away later.*
>
> *An example of a transient condition: A trigger adds a related record to a master record if a field on the master record equals a certain value. It is possible that in a subsequent try, the field value changes and the trigger can perform the operation.*

This is best accomplished by inheriting from the ***accc_DefaultPEConsumer,*** as this class follows a canonical form. There is one methods to override.

```
/**
*       @description child decide how to consume; child should throw EventBus.RetryableException if the
handler  needs to be called again.
*       @param collectionOfEvents the collection of events
*       @param handlers platform event handlers
*       @param attributes platform event attributes
*       @param errResult errors that occur
*       @return true if processed
 */
protected virtual boolean consumePlatformEvent(List<SObject> collectionOfEvents
                    , accc_IProcessEventHandlers handlers
                    , accc_PlatformEventAttrs attributes
                      , List<accc_DefaultPlatformEvent.PlatformEvtResultPOAC> errResult
                      , List<accc_DefaultPlatformEvent.PlatformEvtResultPOAC> theLogData)
{

    // NOTE, Children SHOULD OVERRIDE this method and perform consumption of the event
    // TBD -- Make Abstract

    // if we are here , we handled the events;
    if ( accc_ApexConstants.UNIT_TEST_RUNNING) {
       // record process of save result -- LEAVE this in for testing
       super.addInstrumentation(super.willInstrument,
                        new accc_DefaultPlatformEvent.PlatformEvtResultPOAC('Consumer: consumePlatformEvent
result=true , collection size=' + collectionOfEvents.size())
                        , theLogData);
    }
    return true;

} // end of consumePlatformEvent
```

Figure 5 Default consumePlatformEvent

Custom Metadata

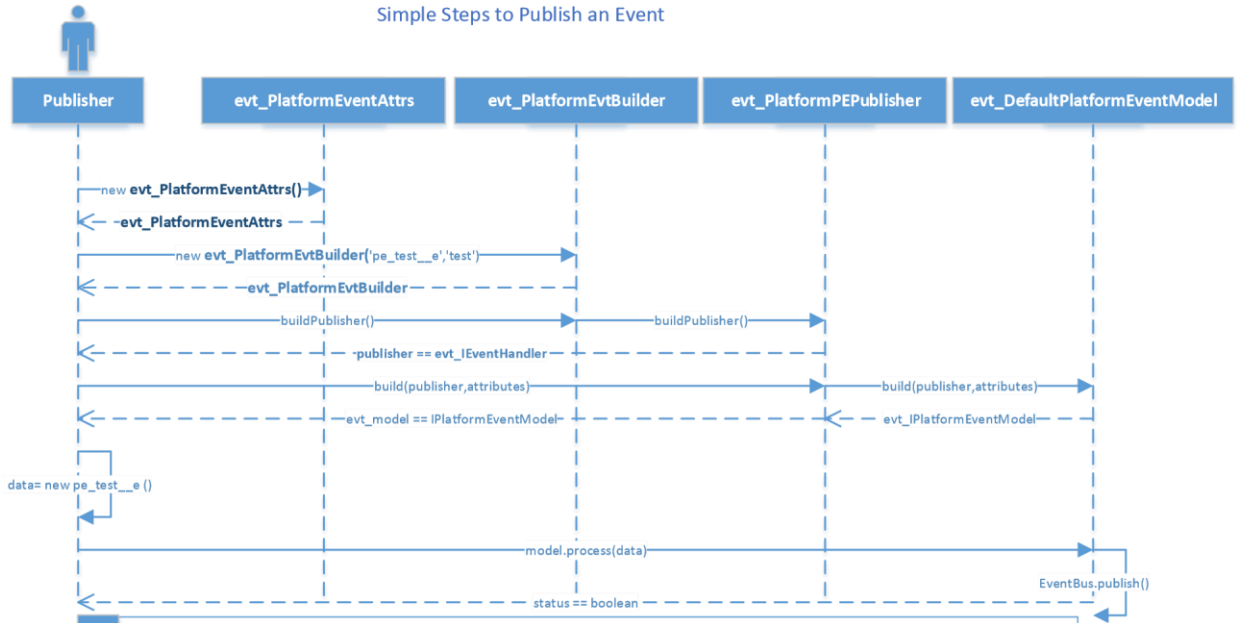
In the custom metadata type below (**Apex Code Configurations),** three environments (based on the '*Label*') are defined with three different runtime environments.

1. **[DEBUG]** *a Sandbox (not running in a Unit Test)*
2. **[PROD]** *Not a sandbox and not a Test procedure*
3. **[TEST]** *i.e. Test.IsRunning*

Updates

Currently, will be adding Summer '19 features **AsyncOperationEvent**; however, this will **tie this package to version 46**. Finally, updates include more data written into the BigObject (i.e. *replayids* and *operation ids*) for recovery on a queued high-volume event.

Appendix: Publish Sequence Diagram (replace accc_ w/ evt_ prefix)




```
SAMPLE CODE
// [1] create default attributes
evt_PlatformEventAttrs attributes = new evt_PlatformEventAttrs();
// [2] create platform event builder, platform event name and the runtime environment ('test','debug'prod')
evt_PlatformEvtBuilder peb = new evt_PlatformEvtBuilder('pe_test__e','test');
// [3]create the default publisher
evt_IEventHandler publisher = peb.buildPublisher();
// [4] create event model
evt_IPlatformEventModel model = peb.build(publisher,attributes);
// [5] create event to publish (note, the name must match that passed to the builder)
List<pe_test__e> data = new List<pe_test__e> { new pe_test__e () };

// [6] process/publish the event (returns true, if processed successfully)
System.debug('++++ result =' + model.process(data));
```

Platform Attributes

Platform attributes class, *accc_PlatformEventAttrs*, helps control functionality of the process of publishing and subscription. The table below breaks down the attributes and purpose. The names are defined in the class, *accc_PlatformEventAttrs.cls,* and shown bellows.

Name	Value	Comment
SERIALIZE_EVENTS_s	Boolean	If true, converts the incoming List of events into JSON. The JSON is passed into the log handler for processing.

EVENT_LOGGING_s	enum EventLogging {
	ALL,
	ON_ERROR,
	ON_SUCCESS,
	ON_LOG
	}
		What information to log

RETRY_COUNT_s	**Integer**, the value between 1 and 9 (inclusive), default is 5	Number of retries; the default is 5. Retries occur ONLY for subscribers. Within a trigger there may be an occurrence (due to latency) that had not occurred. Thus, an *EventBus.RetryException* can be thrown. The consumer will handle up-to the retry-count.
CHECK_EVENT_NAME_s	Boolean, default is true	Checks to determine if the event name passed in is correct.
ADD_INSTRUMENTATION_s	Boolean, default is true	Gather instrumentation (starttime, end-time). The information is passed along to the log handler
EVENT_STORING_s	Boolean, default is true	Stores event(s) from consumer or publisher into a Big Object
EVENT_PROCESSING_BATCH_SIZE_s	Specify the number of events to consume (default is 200)	The Default Consumer supports the ability to define the number of Events to process in a given trigger. Events successfully consumed use the **setResumeCheckpoint**() to

Name	Value	Comment
		ensure subsequent calls into the consumer trigger apply to the next (unprocessed) event

Users can change the behavior with the use of a Map<String,Object>. In fact, the defaults are defined below.

```
Map<String, Object> attributes = new Map<String, Object> {
                    AUDIT_EVENTS_s => AuditEvents.HIGH_VOLUME
                    , EVENT_LOGGING_s => EventLogging.ALL
                    , RETRY_COUNT_s => DEFAULT_RETRY_COUNT
                    , CHECK_EVENT_NAME_s => true
                    , ADD_INSTRUMENTATION_s => true
                    , EVENT_STORING_s => true
                    , EVENT_PROCESSING_BATCH_SIZE_s => 200

};
```

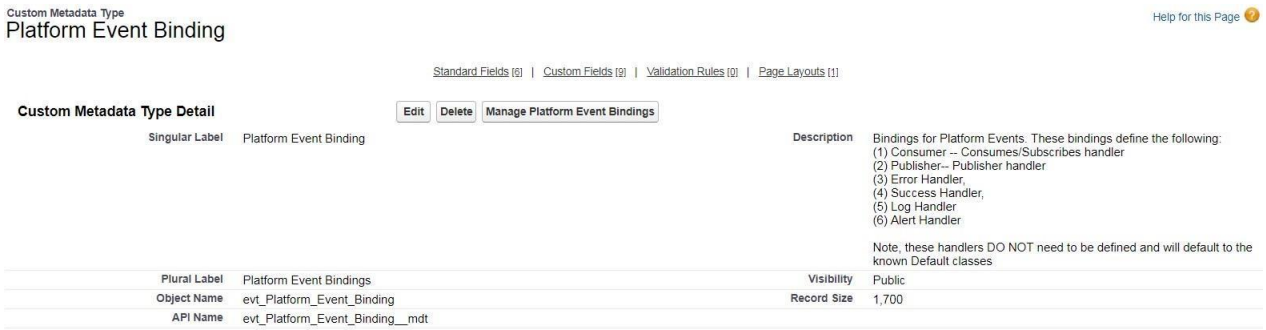
Figure 6 Default Attributes

Custom Metadata

The Platform Event Wrapper uses a custom metadata, *accc_Platform_Event_Binding__mdt*, to lookup the event information. The information contains the following fields:

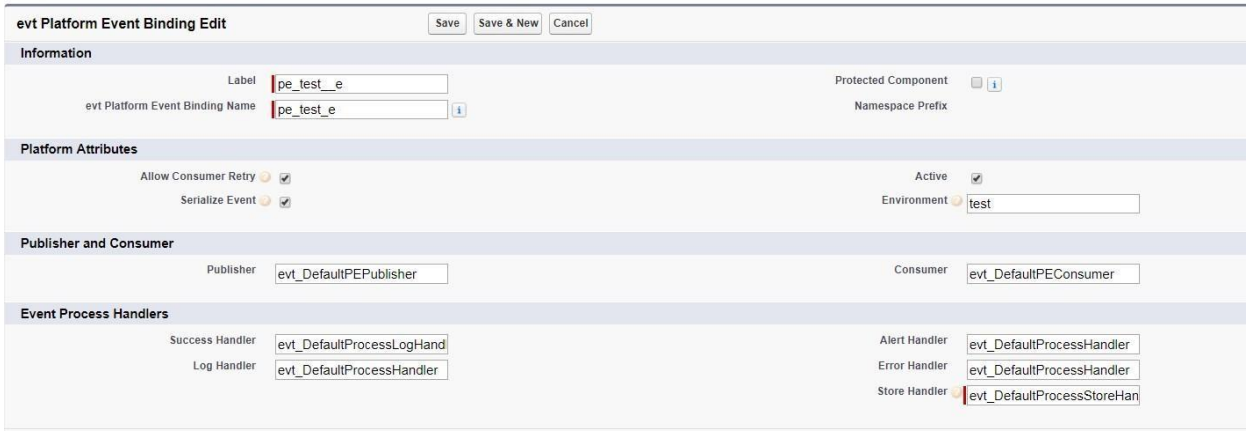
Label	API Name	Type	Comment
Active	Active__c	Checkbox	Allow execution. If inactive the event is not handled (published or consumed)
Alert Handler	Alert_Handler__c	Text(255)	The alert handler will be called at various steps performed by the consumer or the publisher
Consumer	Consumer__c	String	The consumer is how one decides to consume an event. There are hooks to override behavior, as needed.
Environment	Runtime_Environment__c	Picklist (*test*,*debug*, *production*)	Which environment this event will run in

Label	API Name	Type	Comment
Error Handler	Error_Handler__c	String	The error handler will be called at various steps of errors/exception that occur in the consumer or the publisher
High Volume	High_Volume__c	Boolean	Is this a high-volume? If true, then it is common to save the incoming event in JSON and passed to the log handler
Log Handler	Log_Handler__c	String	The log handler will be called in the consumer or the publisher with instrumentation data
Publisher	Publisher__c	String	The publisher to invoke. The default publisher, *accc_DefaultPEPublisher*, provides a consistent process for publishing. There are hooks to override behavior, if needed.
Success Handler	Success_Handler__c	String	The success handler will be called if no errors or exceptions that occur in the consumer or the publisher
Store Handler	Store_Handler__c	String	Store handler will be called by consumer or publisher to store event(s) into a Big Object



Platform Event Bindings

evt Platform Event Binding



Storage – Big Object

Either the consumer or the publisher can control whether the Platform Event(s) will be stored. However, there are some caveats to this approach.

- The event MUST BE marked to be serialized. Otherwise, the data will not be stored.
- The largest field available is the ***Long Text*** (131,072). As such, the data will attempt to store the record(s) but will be marked as truncated if it does not fit. At the time of this writing, *compression* has not be introduced.
- Storage is performed *asynchronously* (so as to improve performance and avoid governor limits, if possible).

The Big Object, **accc_Org_Events__b**, contains the following fields:

Label	API Name	Type	Comment
Event Day	Event_Date_Day__c	Number (2)	The day - Calculated from the date/time

Label	API Name	Type	Comment
Event Month	Event_Date_Month__c	Number (2)	The month - Calculated from the date/time
Event Year	Event_Date_Year__c	Number (4)	The year - Calculated from the date/time
Event Date	Event_Date__c	DateTime	Calculated from instantiation
Event JSON	Event_Json__c	Long Text (131072)	The serialized event
Event Name	Event_Name__c	Text(254)	Event Name (ends in __e)
Event Truncated	Event_Truncated__c	Boolean	True, if the serialized event(s) does not fit in the **Event_Json__c** field.
Number of Events	Number_Of_Events__c	Number	Number of serialized events held in the **Event_Json__c** field.

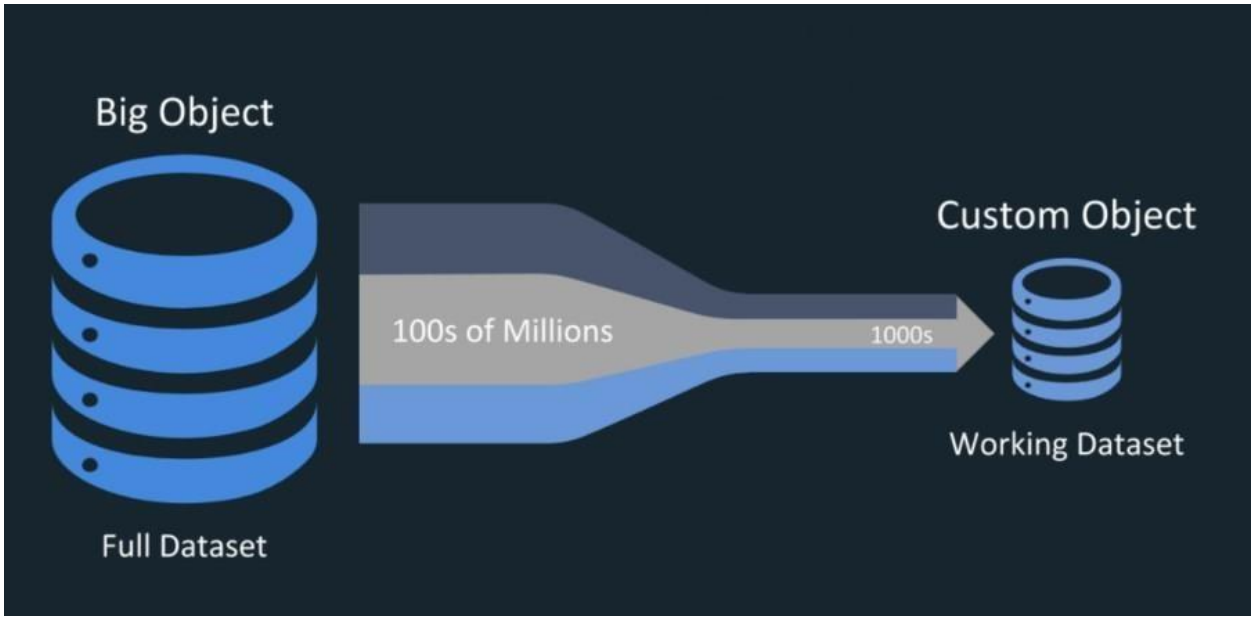


Figure 7 Big Object to Custom Object

The Big Object, **accc_Org_Events__b**, facilitates billions of records. However, in order to extract portions of the data (events) it would be advisable to create a secondary custom object as the working dataset. See this Salesforce help for more information).

Storage of Events

Storage of Platform Events is handled by the **accc_DefaultProcessStoreHandler** class. Because custom metadata is used to handle changes in the behavior of handlers, one can easily define a new handler. The only constraint is:

- Class MUST inherit from **accc_DefaultProcessHandler,**
    - Inherited class overrides *childProcess* method. ○ Note, this method will be passed in the List of Data (which will contain the serialize event),
    - ○ The attributes used for this events (i.e. If it was serialized) ○ It is expected to return a user-defined object.

Platform Events Consumer Model

The primary consumer Model this framework follows (and recommends) is the use of **setResumeCheckpoint** because of the reasons stated in the comparison chart below. However, it supports both models.

The following table is found at this link and helps *determine which method is most suitable for resuming a platform event trigger*.

`setResumeCheckpoint()` Method	`EventBus.RetryableException`
Trigger execution continues after `setResumeCheckpoint()`.	Trigger execution halts after the `EventBus.RetryableException` is thrown.
DML operations performed are committed.	DML operations performed before the exception is thrown are rolled back and not committed.
When the trigger fires again, only the event messages after the one with the specified replay ID are resent, in addition to any new event messages.	When the trigger fires again, all event messages from the previous batch are resent in the new batch, in addition to any new event messages.
These `TriggerContext` properties don't apply and aren't populated: `retries` and `lastError`.	These `TriggerContext` properties are populated: `retries` and `lastError`.

The Consumer model (via attributes) allow you to control the number of platform events consumed within a trigger. The default is 200 and the maximum is 2000. Depending on the amount of work and efficiency of the trigger handler you may need to vary the amount of work consumed (test empirically, in multiple environments; especially, bulk). Work the framework does regarding Store and Log is performed via Queueable Apex as to avoid limits/overhead