

Platform Event Wrapper

What is it?

This unmanaged package wraps the 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 mish-mash of incongruous parts. It provides a consistent and manageable control of publishing and subscribing to platform events; both standard and high-volume events.

The value lies in being consistent, reliable, reusable and flexible.

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
evt_IEventHandler	Defines the behavior for a Publisher or Consumer
evt_IPlatformEventModel	Is a container for handling publish or subscription service
evt_PlatformEvtBuilder	Builds the model based on custom metadata, if defined, or uses defaults
evt_IProcessEventHandlers	Container of handlers (log, success, error, alert)
evt_PlatformEventAttrs	Attributes used to manage logging (high-volume or standard), alerts, retries, validations, etc. of the publish/consume process
evt_PlatformEvtMdtDataModel	Provides a wrapper around the custom-metadata (DAO, Data Access Object), <i>evt_Platform_Event_Binding__mdt</i>

A static class diagram brings this into more clarity.

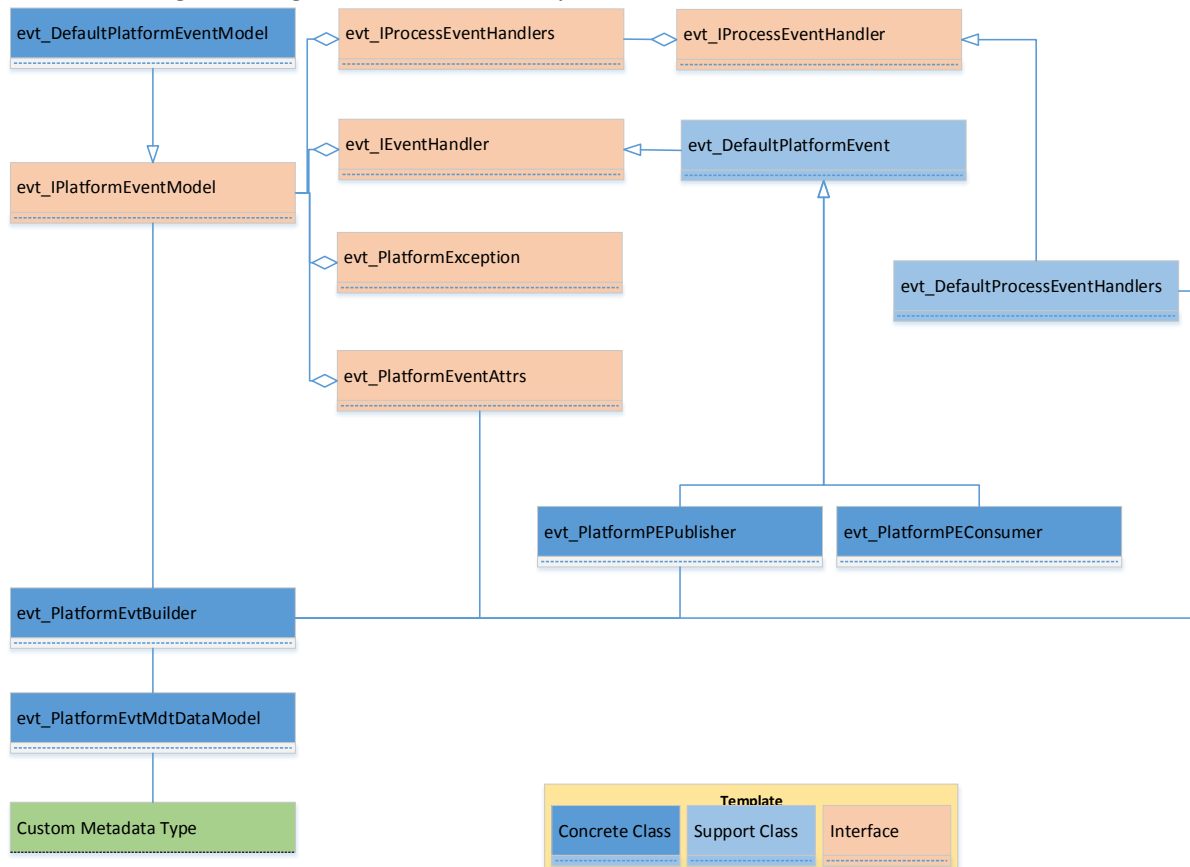


Figure 1 Static Class Diagram

Platform Event Model

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

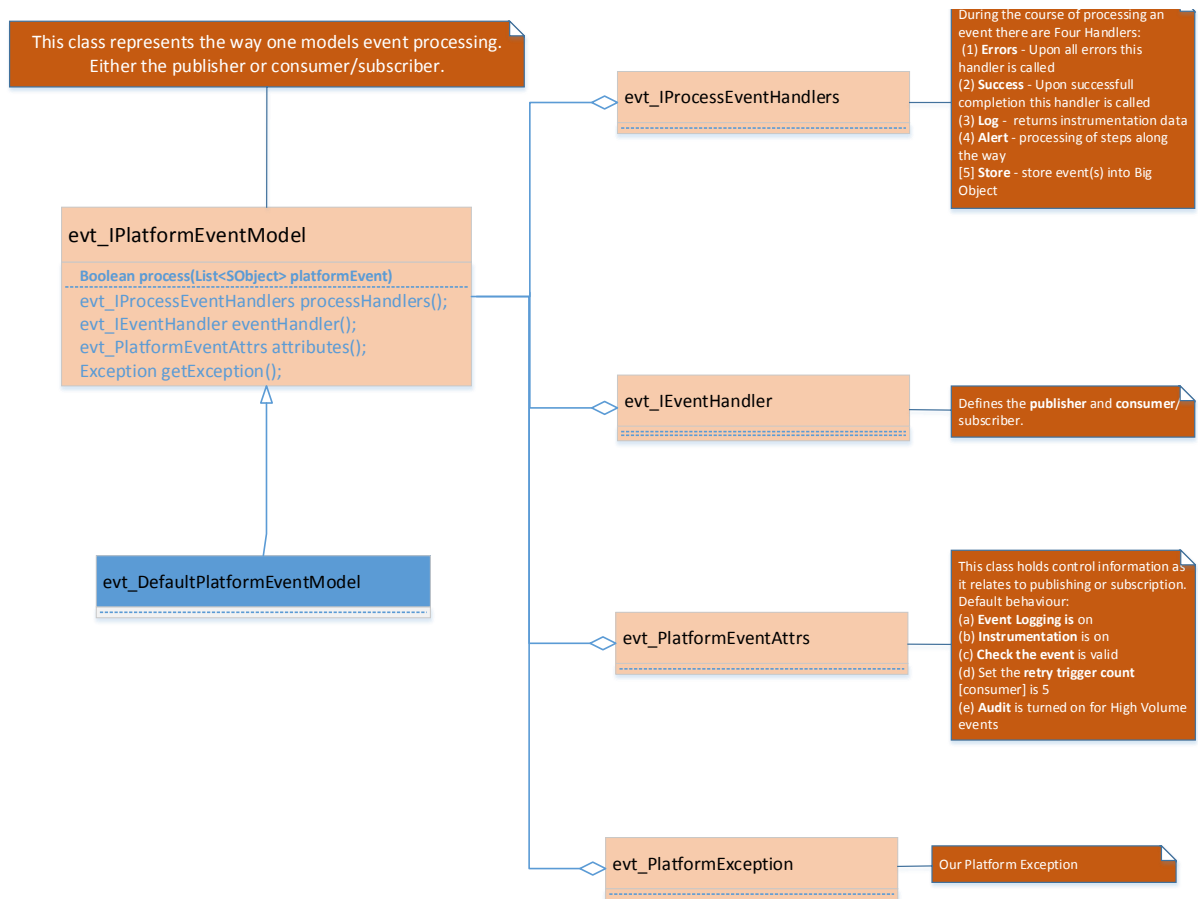


Figure 2 Platform Event Model

Code Snippet / Example for Publishing

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

```

// [1] create default attributes -- optional
evt_PlatformEventAttrs attributes = new evt_PlatformEventAttrs();
// [2] create platform event builder, platform event name and the runtime environment ('test','debug'prod')
evt_PlatformEvtBuilder builder = new evt_PlatformEvtBuilder('pe_test__e','test');
// [3] create the default publisher
evt_IEventHandler publisher = builder.buildPublisher();
// [4] create event model
evt_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	<code>evt_PlatformEventAttrs attributes = new evt_PlatformEventAttrs();</code>	Create the default attributes (optional). See Platform Attributes
2	<code>evt_PlatformEvtBuilder builder = new evt_PlatformEvtBuilder('pe_test__e','test');</code>	Create a platform event builder. The event name, ' <i>pe_test__e</i> ', 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	<code>evt_IEventHandler publisher = builder.buildPublisher();</code>	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	<code>evt_IPlatformEventModel model = builder.build(publisher);</code>	Now we can build the model. The model holds the five handlers, the event handler (publisher) and attributes that control the behavior.
5	<code>List<pe_test__e> pe=new List<pe_test__e> {new pe_test__e ()};</code>	Create the collection of events to publish
6	<code>model.process(pe);</code>	Publish the event, returning true, if successful; otherwise false.

Code Snippet / Example for Consuming

The sample code would be consumed in a Trigger class or one invoked from an after insert platform event trigger.

```
// [1] create default attributes -- optional
evt_PlatformEventAttrs attributes = new evt_PlatformEventAttrs();
// [2] create platform event builder, platform event name and the runtime environment ('test','debug'prod')
evt_PlatformEvtBuilder builder = new evt_PlatformEvtBuilder('pe_test__e','test');
// [3] create the default consumer
evt_IEventHandler consumer = builder.buildConsumer();
// [4] create event model
evt_IPlatformEventModel model = builder.build(consumer); // or builder.build(consumer,attributes);
// [5] process/consume the event (returns true, if processed successfully)
System.debug('++++ result =' + model.process(Trigger.new));
```

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

Figure 4 Consume an event

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

Code	Comment
1 evt_PlatformEventAttrs attributes = new evt_PlatformEventAttrs ();	Create the default attributes (optional). See Platform Attributes
2 evt_PlatformEvtBuilder builder = new evt_PlatformEvtBuilder ('pe_test__e','test');	Create a platform event builder. The event name, ' <i>pe_test__e</i> ', 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 evt_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 evt_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.

Special notes about the consumer

Consumers are 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 **evt_DefaultPEConsumer**, as this class follows a canonical form. There are two 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
, evt_IProcessEventHandlers handlers
, evt_PlatformEventAttrs attributes
, List<evt_DefaultPlatformEvent.PlatformEvtResultPOAC> errResult
, List<evt_DefaultPlatformEvent.PlatformEvtResultPOAC> theLogData)
{
    // record process of save result
    super.addInstrumentation(super.willInstrument,
        new evt_DefaultPlatformEvent.PlatformEvtResultPOAC('Consumer:
consumePlatformEvent result=true')
        , theLogData);

    return true;
} // end of consumePlatformEvent
```

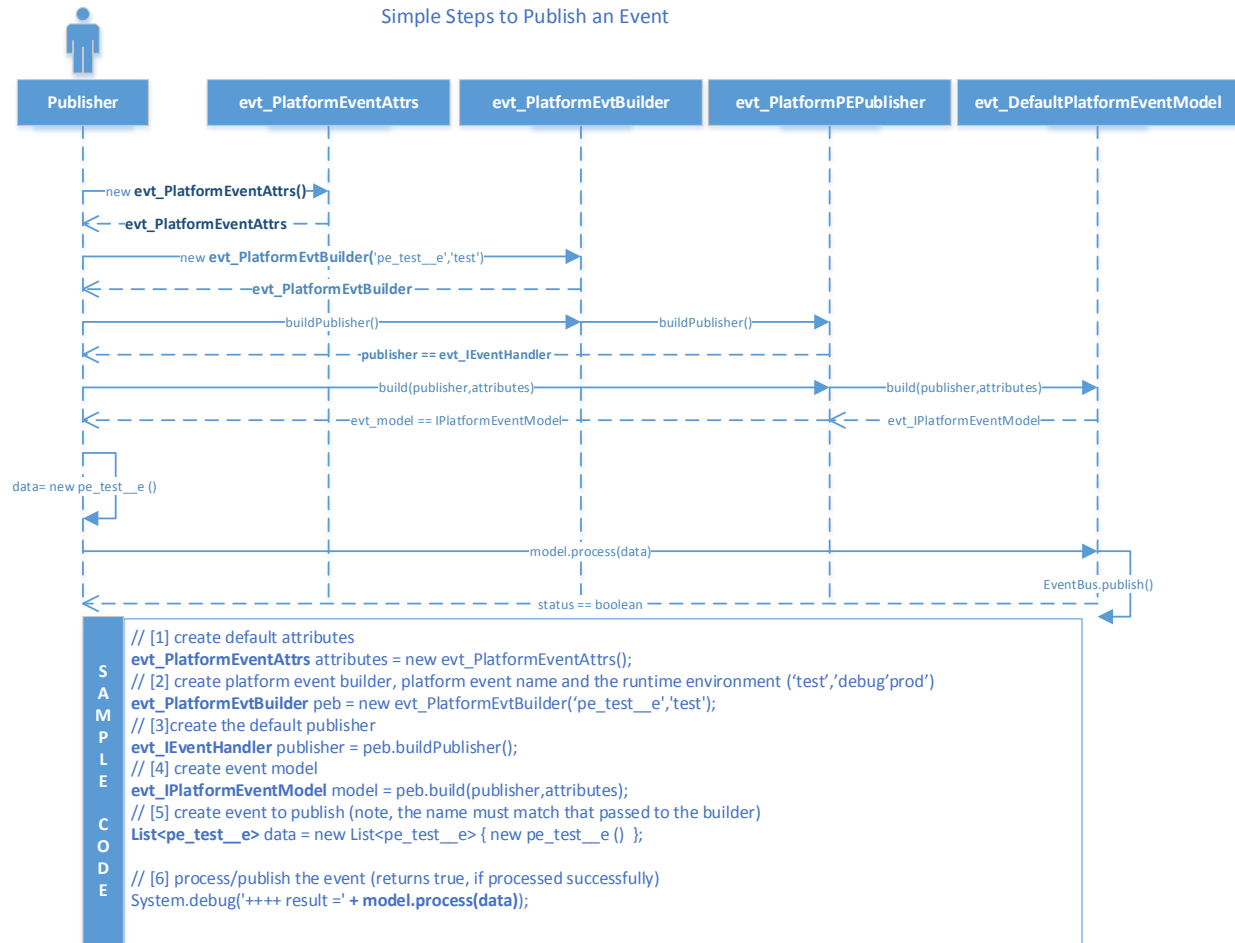
Figure 5 Default consumePlatformEvent

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*

Appendix: Publish Sequence Diagram



Platform Attributes

Platform attributes class, *evt_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, *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

Name	Value	Comment
		subscribers. Within a trigger there may be an occurrence (due to latency) that had not occurred. Thus, an <i>EventBus.RetryException</i> 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 (start-time, 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

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
};

```

Figure 6 Default Attributes

Custom Metadata

The Platform Event Wrapper uses a custom metadata, *evt_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.

Label	API Name	Type	Comment
Environment	Environment__c	Picklist (test,debug,production)	Which environment this event will run in
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, <i>evt_DefaultPEPublisher</i> , 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

Standard Fields (6) | Custom Fields (9) | Validation Rules (0) | Page Layouts (1)

Custom Metadata Type Detail		Edit Delete Manage Platform Event Bindings	
Singular Label	Platform Event Binding	Description	Bindings for Platform Events. These bindings define the following: (1) Consumer -- Consumes/Subscribes handler (2) Publisher-- Publisher handler (3) Error Handler, (4) Success Handler, (5) Log Handler (6) Alert Handler Note, these handlers DO NOT need to be defined and will default to the known Default classes
Plural Label	Platform Event Bindings	Visibility	Public
Object Name	evt_Platform_Event_Binding	Record Size	1,700
API Name	evt_Platform_Event_Binding__mdt		

Platform Event Bindings

New									
Platform Event Binding Name	Active	Environment	High Volume	Pulisher	Consumer	Alert Handler	Error Handler	Log Handler	Success Handler
pe_test_e	✓	test	<input type="checkbox"/>	evt_DefaultPEPublisher	evt_DefaultPEConsumer	evt_DefaultProcessHandler	evt_DefaultProcessHandler	evt_DefaultProcessHandler	evt_DefaultProcessHandler
recordCDC_e	✓	test	✓	evt_DefaultPEPublisher	evt_DefaultPEConsumer	evt_DefaultProcessHandler	evt_DefaultProcessHandler	evt_DefaultProcessHandler	evt_DefaultProcessHandler

evt Platform Event Binding

Save

Save & New

Cancel

Label

pe_test_e

Protected Component

☐

i

evt Platform Event Binding Name

pe_test_e

i

Namespace Prefix

Allow Consumer Retry

☒

Serialize Event

☒

Active

☒

Environment

test

Publisher

evt_DefaultPEPublisher

Consumer

evt_DefaultPEConsumer

Success Handler

evt_DefaultProcessLogHand

Log Handler

evt_DefaultProcessHandler

Alert Handler

evt_DefaultProcessHandler

Error Handler

evt_DefaultProcessHandler

Store Handler

evt_DefaultProcessStoreHan

Storage – Big Object

Either the consumer or the publisher can control whether the Platform Event(s) will be stored. However, note 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 been introduced.
- Storage is performed asynchronously (so as to improve performance and avoid governor limits).

The Big Object, **evt_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
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.

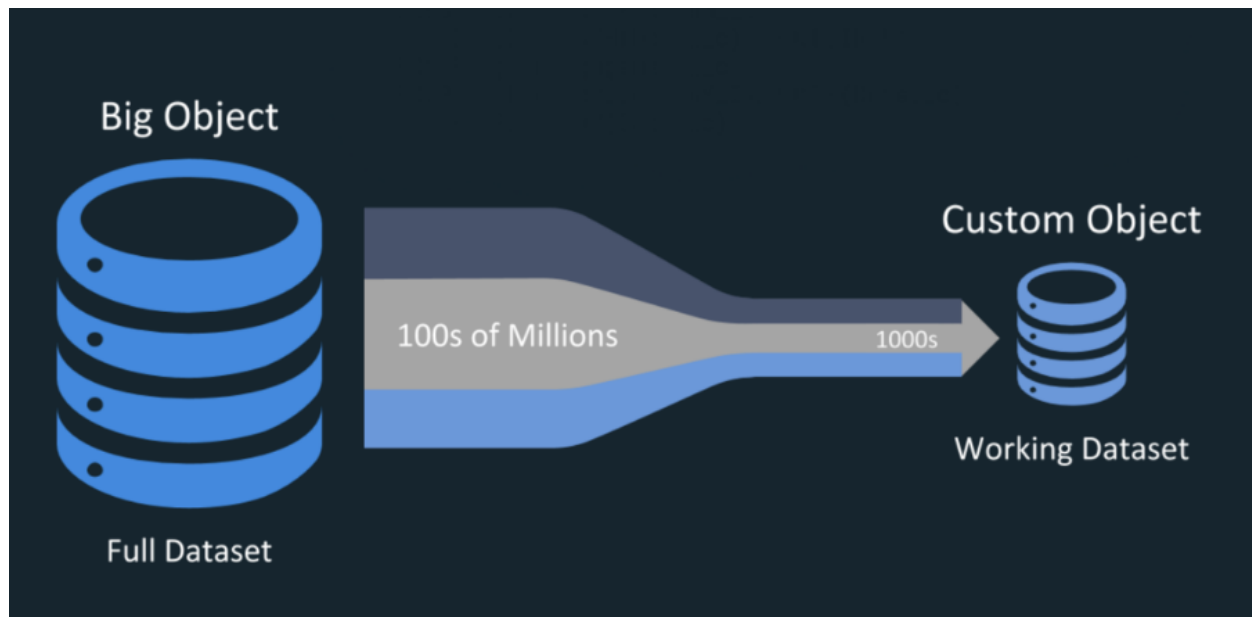


Figure 7 Big Object to Custom Object

The Big Object, **evt_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 **evt_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 **evt_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 (ie. If it was serialized)
 - It is expected to return a user-defined object.