GENERIC “LOG” EVENT ENTRY

Platform Events

Abstract

Discusses a Proof Of Concept (POC) of generic log events via platform events sent from Salesforce. The design provides a lot of flexibility to change behavior; however, was initially designed and implemented within a week.

bill anderson

Bill.anderson@salesforce.com

Table of Contents

[This page intentionally left blank]

General Comments and Out of Scope

The following items are out of scope for this POC (Proof of Concept):

- **Bulk** – The test cases and output of the payload could cause HEAP and CPU issues. However, the requirement was that the collection of SObject would, at most, be 1. The short window for POC (7 days); attempts to provide this feature were at an accelerated pace.
- **Metadata Cohesiveness** – The metadata used to parse the incoming collection of SObjects ASSUMES the metadata fields correctly correspond to SObject . For example, if the domain is *Contact*, the metadata fields are found in the *Contact* object
- **JSON generator** (creates the payload) – will throw when it cannot find a field in the SObject (though, can be controlled via metadata). This behavior is by design. The generator does not segment out the entries into separate Payloads as it was not part of the design. In addition, HEAP size (>6M) can throw when processing a LARGE number of entries (see Limits section).
- **POC** – is meant to prove the ability to send platform events in a generic way (driven by metadata and abstraction). This is meant to be tested, owned and validated by ACCC should they wish to use in production. Code is AS-IS.
- **Cross References** – Cross references are possible one-depth per Log Entry. Cross references can cause additional HEAP, CPU and DML issues (see Limits section).
- **Metadata Fields –** Additional metadata fields could be utilize to make parsing the collection of SObjects more robust; however, due to time constraints, have been consolidated.
- **Selector –** There were portions that embedded the SOQL in non-selector code (see the *accc_MetadataBuilder*). Needs Refactoring.
- **Performance –** Performance metrics were gathered; but not extensively. However, there are a lot moving parts where having a base line of SObjects would be appropriate. In a cursory mode, chunking the SOBjects into various sizes needs to be perform per SObject and tested empirically (best practice).
- **Additional Frameworks –** Additional framework is used to perform various functions, such as sending platform events, Big Objects, etc. but are not within the scope of this document; though, discussed in other sources.

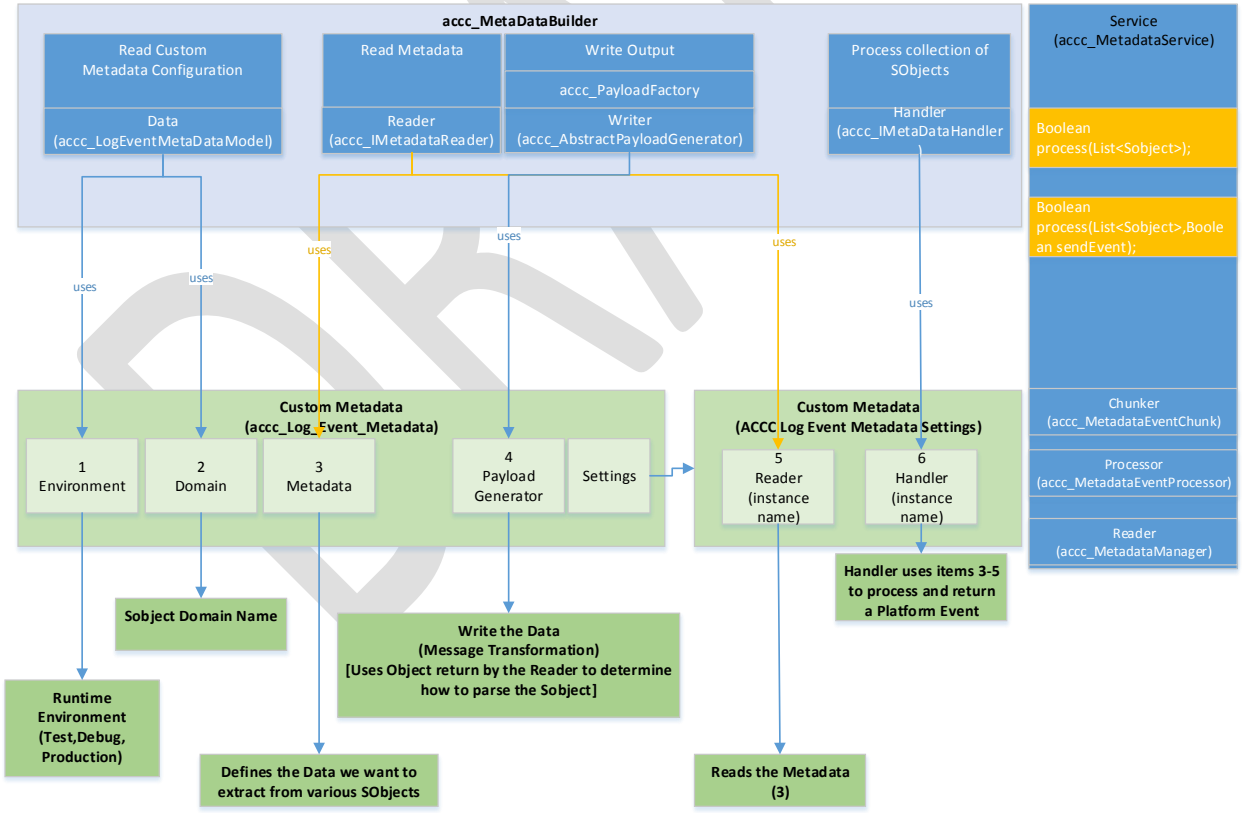
Quick Overview

The design of the Metadata Log Entry is composed of these salient components and briefly outlined below.

Parts	Concrete Class
Read Salesforce Custom Metadata	*accc_LogEventMetadataModel*
Read the Metadata that describes the Data pulled from the SObject list.	*accc_MetadataDefaultReader*
Output (Json) Generator	*accc_PayLoadGeneratorJSON*
Data Handler, processes the collection of SObjects using both the *accc_MetadataDefaultReader* and *accc_ PayLoadGeneratorJSON*	*accc_MetaDataHandler*
Builder, *accc_MetadataBuilder* builds the above parts using the Salesforce Custom Metatdata	*accc_MetadataBuilder*

Parts	Concrete Class
Metadata Manager, *accc_MetaDataManager*, orchestrates all the above parts	*accc_MetaDataManager*
Metadata Service, *accc_MetaDataService*, uses the metadata Manager to bring all the components together, and processes the incoming SObject collection (asynchronously or synchronously via *accc_MetadataLogEventChunked*). It is considered the service/business layer.	*accc_ MetaDataLogEntryService*
Chunking of SObject collections for handling. The underlying function allows a larger number of SObjects to be processed (i.e. creating and sending Log Events). However, it is not without additional limitations as well.	*accc_MetadataLogEventChunked*
The processor, *accc_MetadataEventProcessor*, performs work synchronously. The behavior is controlled by the custom metadata and used by the Chunker and Service. If data is NOT chunked, the Chunker invokes the processor; otherwise, it utilizes a queueable object passing in the processor along with the chunk SObjects,	*accc_MetadataEventProcessor*

The salient components functions are shown the diagram below.



Metadata Components

There are three different metadata components. Two of the components are Salesforce Custom Metadata. The third component is contained in a field within the Salesforce Custom Metadata.

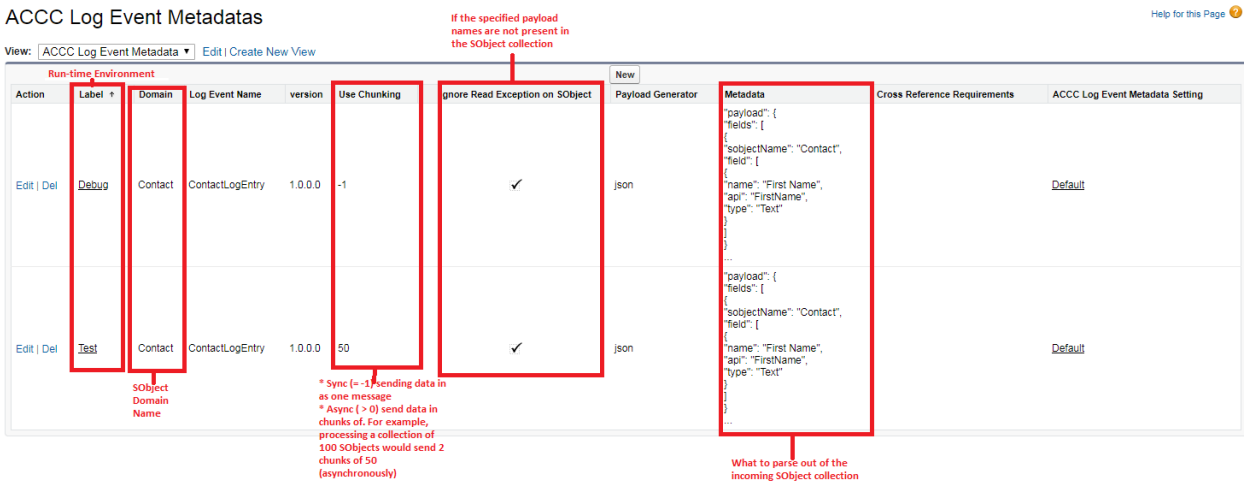
Salesforce Custom Metadata

There are two salient components within Salesforce Custom Metadata are as follows:

1. The first component defines the domain, environment, generator (*json*/*xml*), metadata and a reference to the second component.
2. The second component defines the instances used to process the metadata; namely, read metadata, and the metadata handler.

The third custom metadata component provides the cross references the parent domain (i.e. Contact) require. The third component is not needed if there are cross references.

1. First Component



The fields used are as follows:

- **Label** – is the environment (*test*, *debug*, *production*)
- **Domain** – is the SObject name (will verified SObjects are of the same name)
- **Version** – is the version of the payload
- **Use Chunking** – if -1, run synchronously, otherwise, chunk the SObjects into specified size
- **Ignore Read Exception on SObject** – If the JSON in the metadata refers to a field not present in the SObject , if true, ignore the exception, but record. If false, stop processing immediately and mark the exception and the payload as empty.
- **Payload** generator – either *json* or *xml.* Passed to a Factory to create the appropriate generator
- **Metadata** – how to parse an SObject
- **Cross Reference Requirements** – Allow one-level deep to pull in data from an associated object(s).
- **Setting** – pointer to the second part (Second Component)

2. Second Component

The second part of the custom metadata defines the concrete implementations. Currently, there is only one entry, *Default*. Default provides the basic handling functionality to deal with JSON.

ACCC Log Event Metadata Settings

View: Common Metadata Settings ▾ Edit | Create New View

New

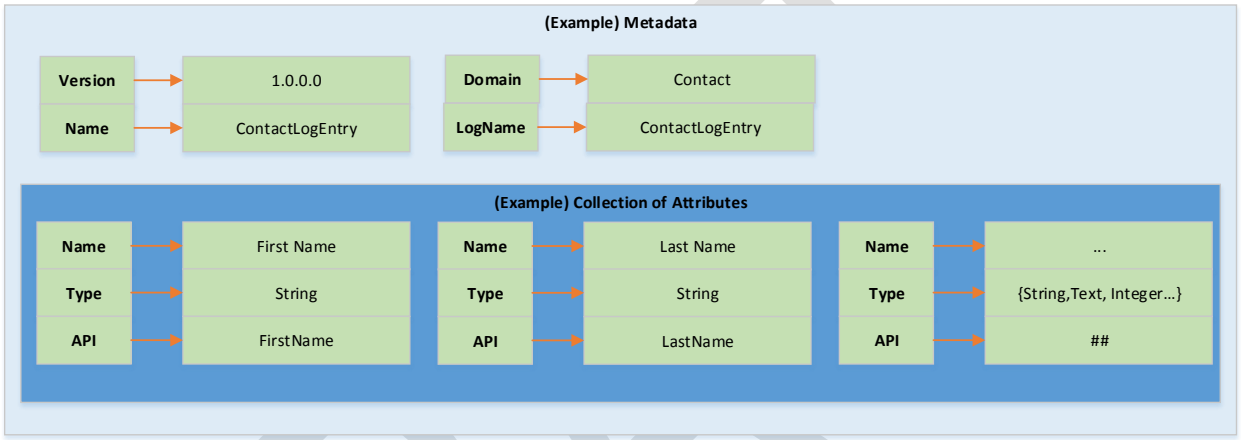
Action	ACCC Log Event Metadata Setting Name ↑	Metadata Runtime Handler	reader
Edit | Del	Default	accc_MetadataHandler	accc_MetaDataDefaultReader

The fields used are as follows:

- **Handler** – glues the reader and the metadata together
- **Reader** – how to parse the metadata from Component One.

Metadata for Parsing

The metadata is as follows:



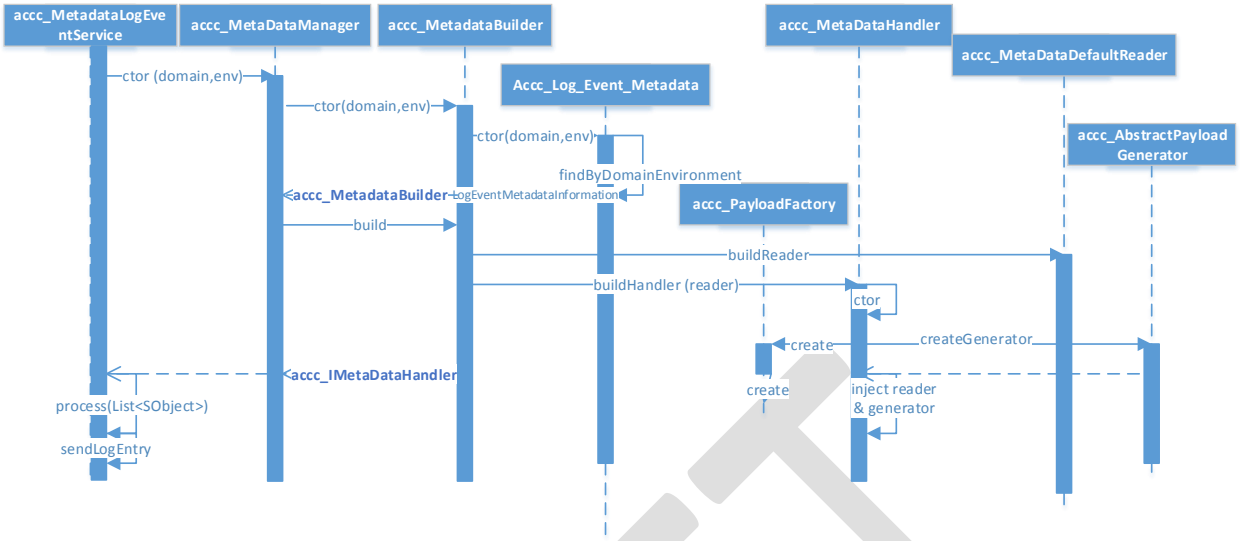
Name	**ASCII, cannot be empty**
Type	**String,Text,Integer,Double,Long,Boolean, Date, DateTime,Id**
API	**API Name to access the Sobject field**

Log Name	**String name of Sobject (cannot be empty or null) + *LogEntry***
Domain	**String name of Sobject (cannot be empty or null)**

The metadata is very sparse. It does not attempt to add fields that have no defined (or future) use at this time. More on the JSON metadata can be found here.

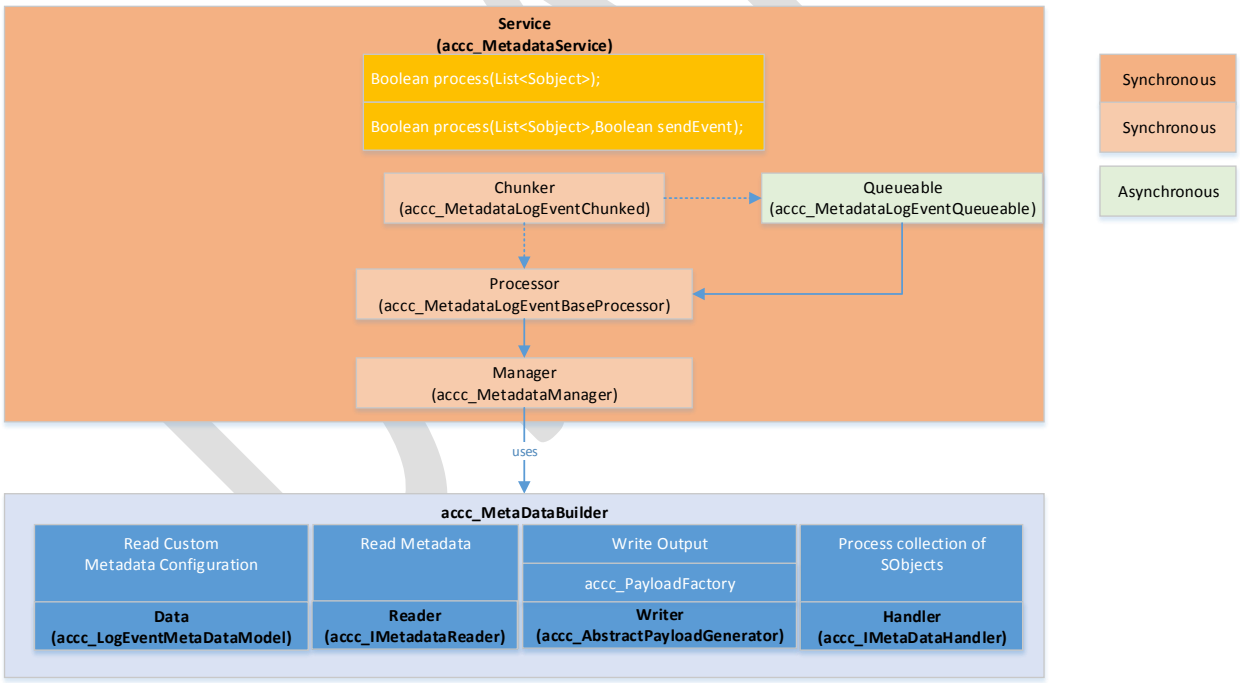
Sequence Diagram

High level sequence diagram is shown below.



Static Class Diagram

The static diagram is broken into the components as shown below. The main entry point is via *accc_MetadataService* (though, it does not have to be).

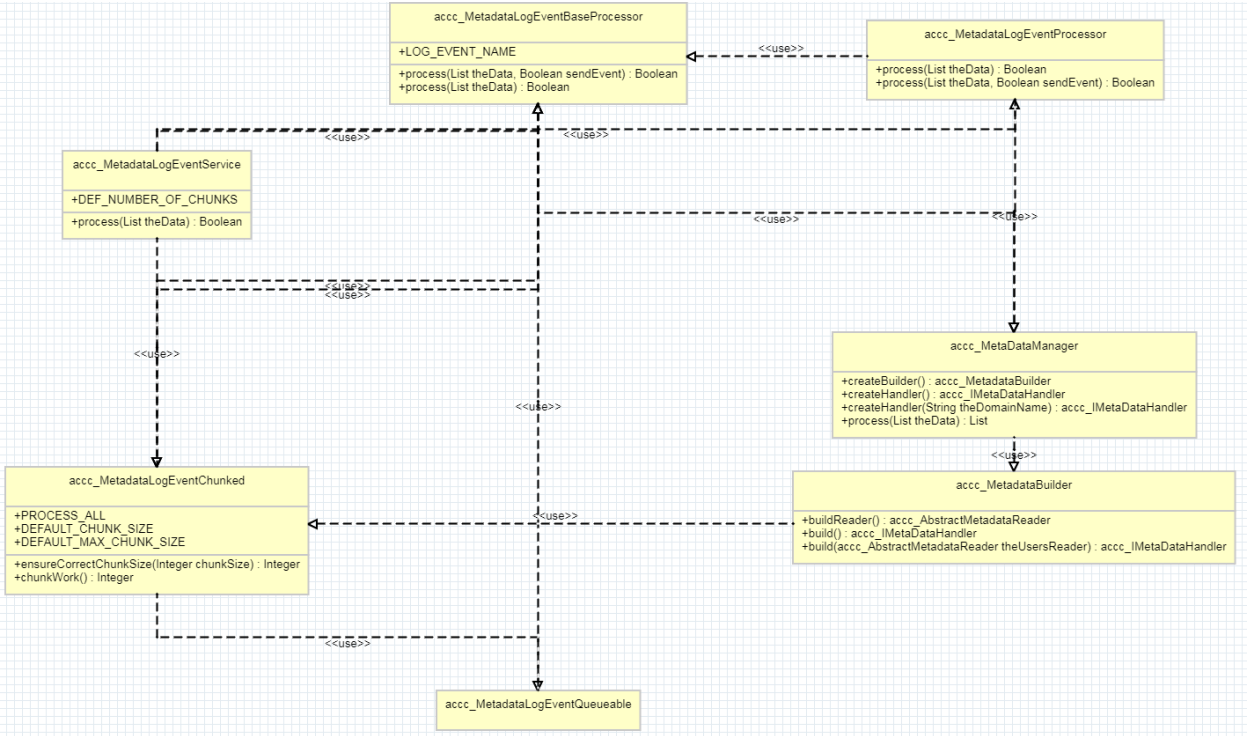


accc_MetadataLogEventService

The Log Event service provides a simple wrapper around the processing of the data and sending the platform events. The class, **accc_MetadataLogEventService** provides the ability to send a platform event; namely, ***ACCC_Log_Entry__e*** *via* the *Chunker***, accc_MetadataLogEventChunked**. The *Chunker*

breaks the collection of SObjects into manageable chunks which provides a simple approach to handling bulk of SObjects. Breaking the collection into smaller number of SObjects will make the generated JSON smaller (as the more SObjects, the larger the JSON payload will be). It should be noted data will only be chunked if the chunk-size (set in the custom metadata type) to a non-negative value (i.e. > 0).

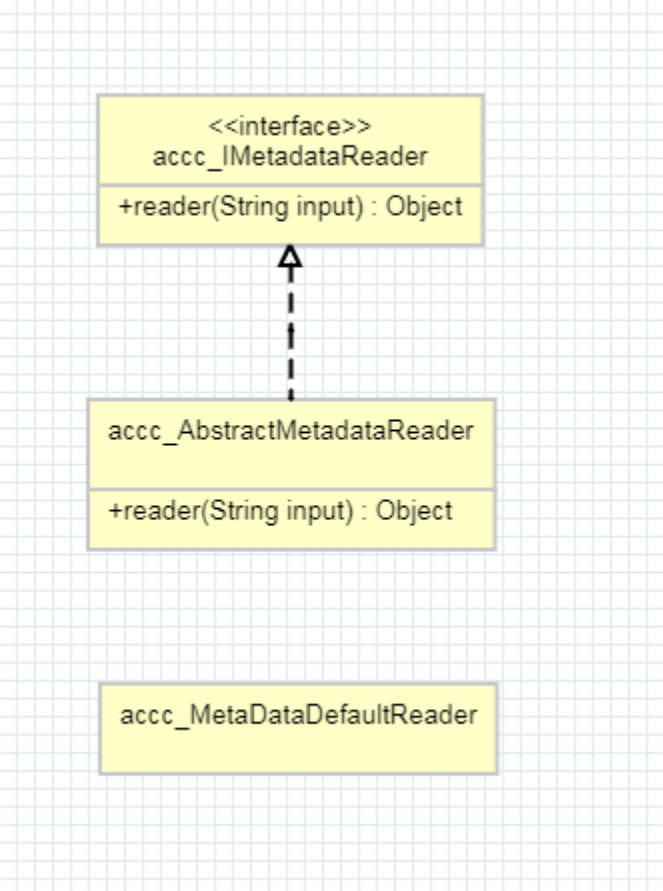
The service class uses a Log Entry Processor, **accc_MetadataLogEventProcessor**, to perform the processing via the **accc_MetaDataManager ;** which orchestrates the creation and flow of components.



Accc_MetadataDefaultReader

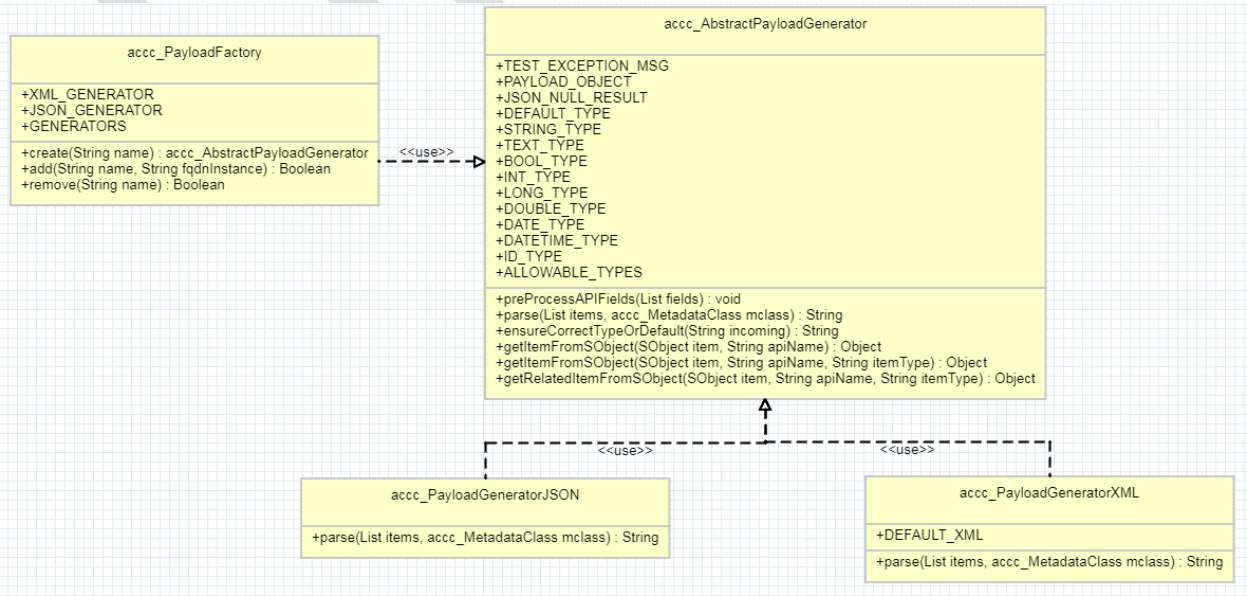
The reader processes the metadata. The metadata defines the fields (api-names, type, etc.) to be found in an SObject. The JSON used in the metadata is passed to JSON2Apex utility to generate the Apex portion (which parses the metadata). Should the metadata evolve, it would need to be re-validated with the JSON2Apex utility and integrated into the code and tests.

Note, a factory was considered for the creation of the reader but time did not allow.



accc_AbstractPayloadGenerator and accc_PayloadFactory

The factory is used to create the appropriate payload. At this time there are only two generators, *accc_PayloadGeneratorJSON* and *accc_PayloadGeneratorXML*; though, only the former is functioning.



accc_PayloadGeneratorJSON

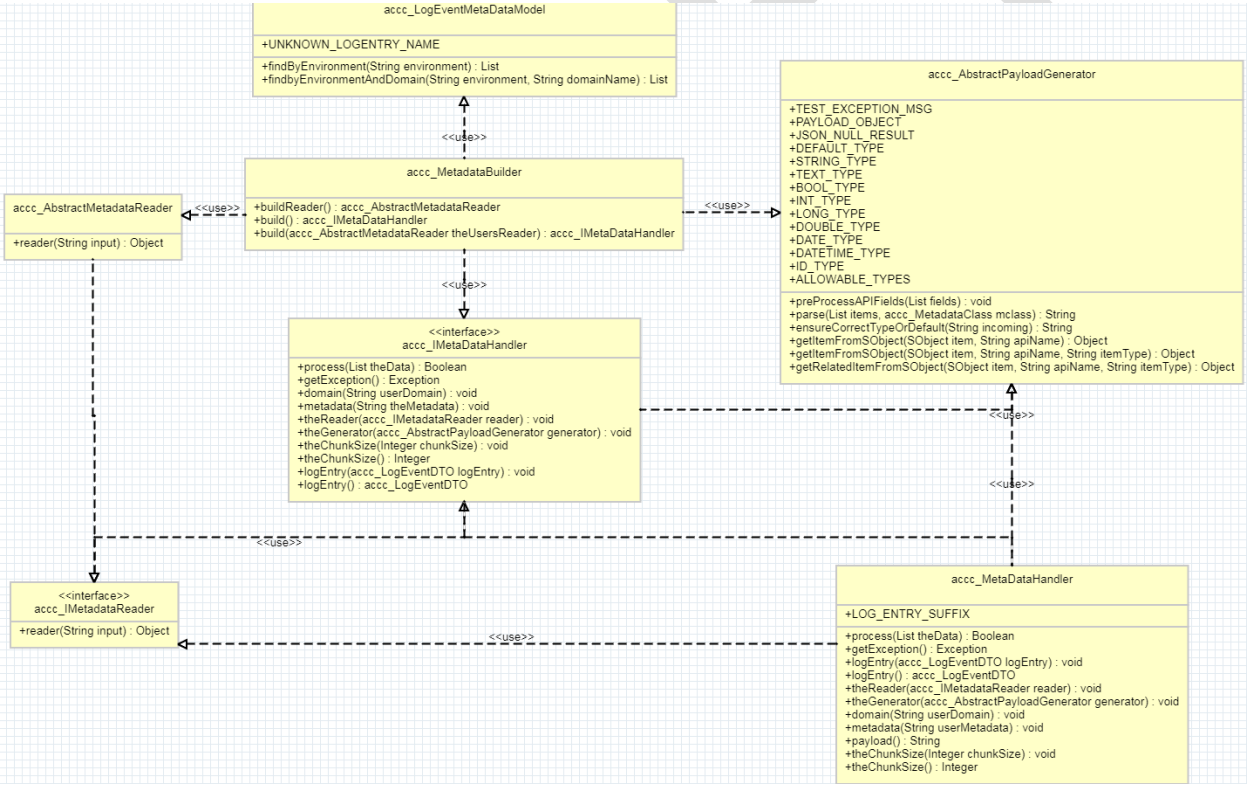
This is the current payload generator implemented. This is where many limit exceptions (see Limit section) can occur. Why? JSON is being created on the fly. Depending on the needed JSON to be injected (as defined by the metadata field in the custom metadata) the JSON can quickly cause a HEAP limit (>6M bytes in synchronous mode) and/or CPU limit. It is further exasperated when cross references come into play (as they add to the payload as well as making SOQL queries for the cross references).

accc_MetadataBuilder

The builder creates all the needed components to read metadata, process collection of SObjects and generate the payload for the event. Unfortunately, the builder is not as abstract as one would like; but time was a factor.

The main output of the Builder is the metadata handler**, *accc_IMetadataHandler***. The handler's salient functionality :

- Read Metadata
- Process SObject Collection
- Generate Payload Output



Business Function : Generic Log Entry Service (Apex)

A log entry service (i.e., for a contact) using the testing runtime, is done so as follows:

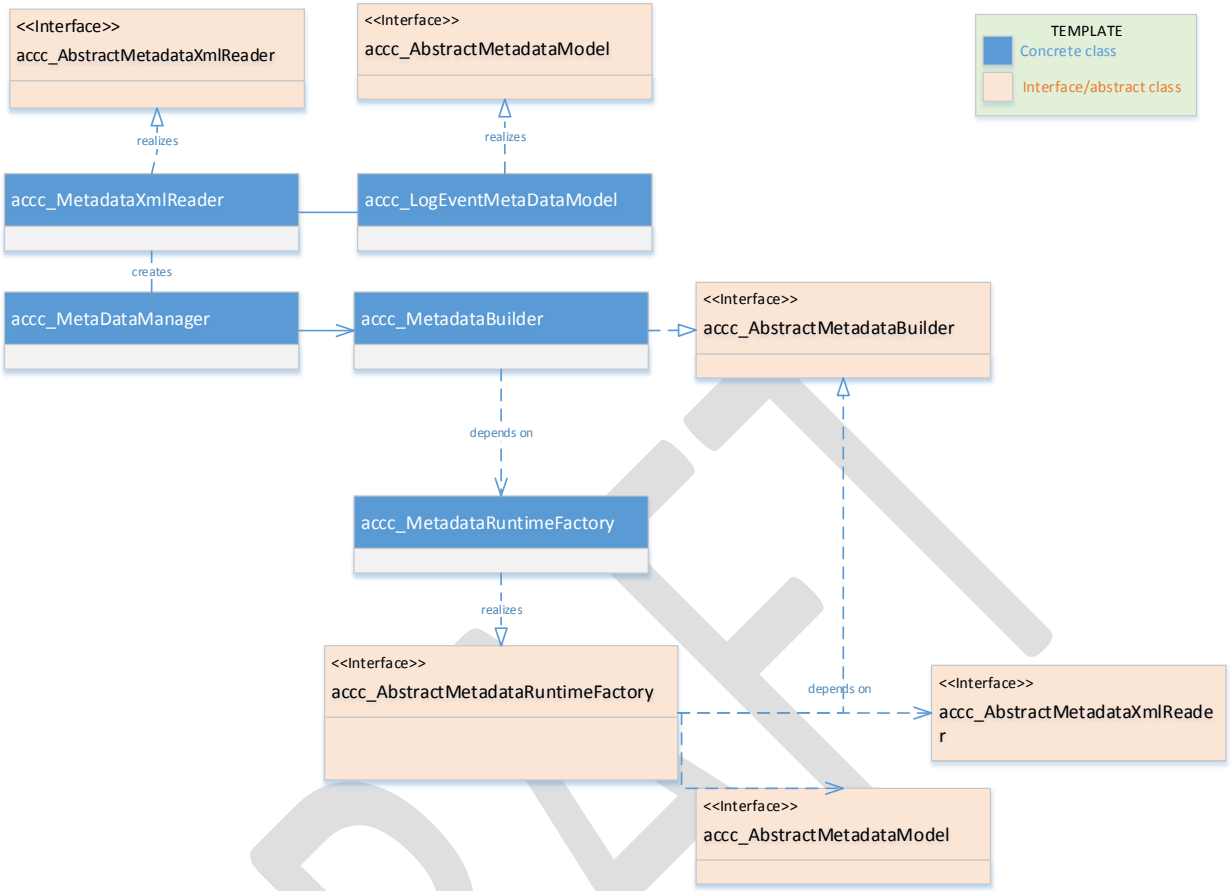
```
List<SObject> cts = [select FirstName,LastName,Email,Phone from contact limit 5];
// we are processing 'contact' using the debug runtime.
accc_MetadataLogEventService service = new
accc_MetadataLogEventService('contact',accc_ApexConstants.DEBUG_CATEGORY);
// publish a log-entry event (returns true, if successful)
boolean status=service.process(cts);
```

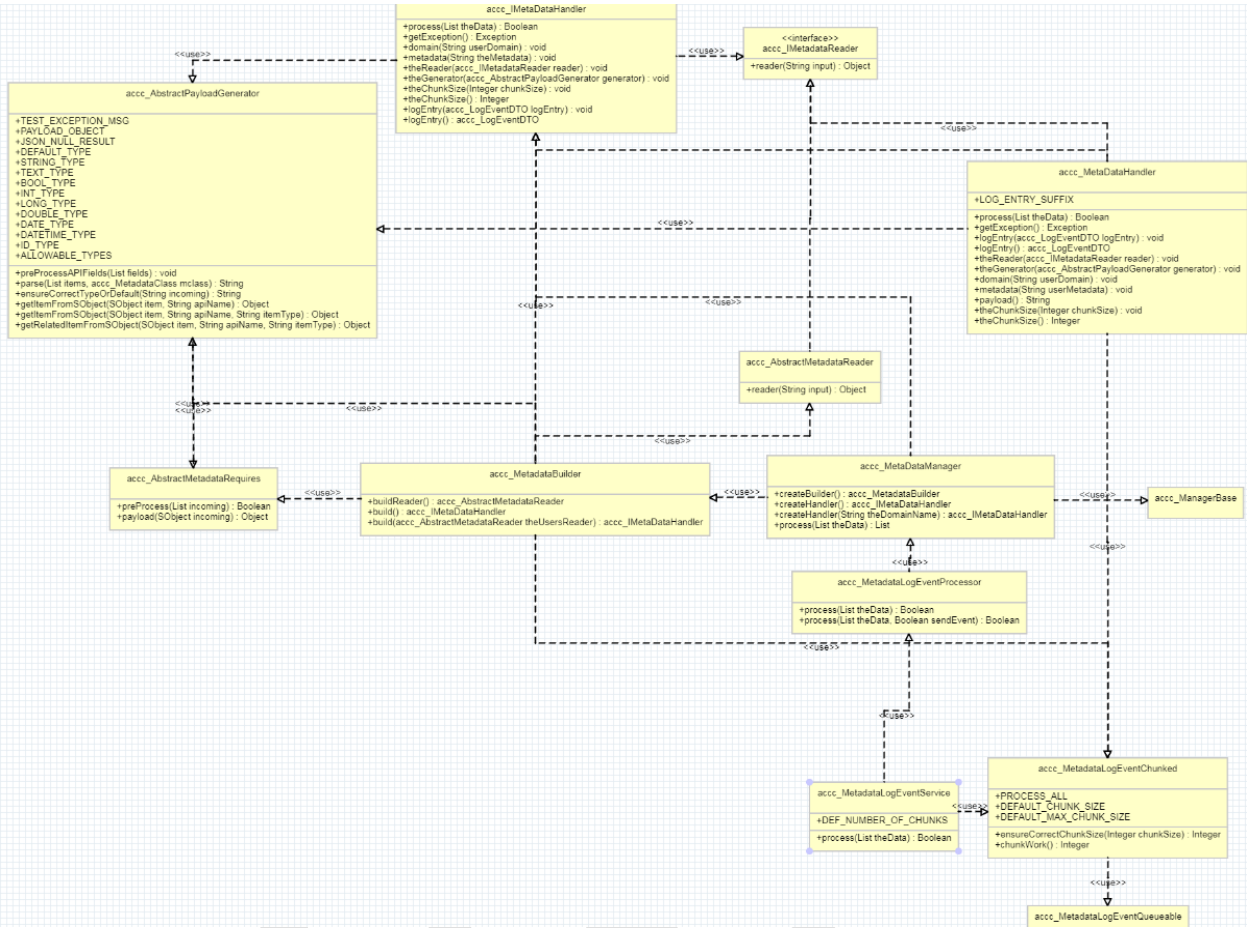
In order to execute the service, there is a requirement to have an entry in the Salesforce custom metadata that matches both the domain (i.e. *contact*) and environment (i.e. *test*).

When you are no longer testing, the proper format is to exclude the environment as the service will determine the appropriate environment as shown below:

```
List<SObject> cts = [select FirstName,LastName,Email, Phone from contact limit 5];
// we are processing 'contact' using the appropriate runtime (test, debug, or production)
accc_MetadataLogEventService service = new accc_MetadataLogEventService('contact');
// process and publish a log-entry event (returns true, if successful)
boolean status=service.process(cts);
```

Static Diagram for Salient components





Caveats - Gotchas

Here are some items to be looked at:

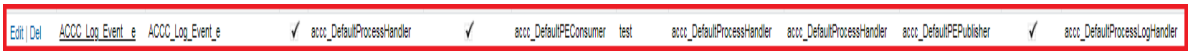
Gotcha	Resolution
If there is no entry in the Salesforce Custom Metadata for the SObject (i.e. Contact) processing will not happen	Add appropriate entry in ACCC Log Event Metadata. Use the *Contact* as a model. Note, if running from Dev Console, the environment lookup will be *debug*. For a unit test, the environment is *test.* Otherwise*, production.*
If you are testing and there is NO test entry in the Salesforce Custom Metadata for the SObject (i.e. Contact) processing will not happen	Add appropriate entry in ACCC Log Event Metadata. Use the *Contact* as a model. Note, if running from Dev Console, the environment lookup will be *debug*. For a unit test, the environment is *test.* Otherwise*, production.*
Metadata has to have the correct field names for the corresponding SObject . For example, in your metadata you have a name equal to *myFirstName,* for the Contact SObject , it will throw an exception. Any field not found in the	Ensure the metadata field name corresponds to the correct API names in the SObject (i.e. *FirstName* for Contact).

Gotcha	Resolution
incoming SObject WILL THROW an exception (and stop processing); this can be controlled by the metadata	
Any change in the metadata has to be validated in JSON2Apex. Otherwise, the processing will fail.	Validate with JSON2Apex.
You should TEST ALL aspects of the metadata components when making any changes; especially, to metadata.	It is NOT recommended you make changes in Production if you have NOT tested in a Sandbox. *Caveat Per-Emptor*.
The SObject collection MUST have the appropriate fields/columns when processing in the metadata.	The application uses a *get(name)* function on the SObject . If the name is NOT found an exception is thrown. At the time of this writing, exceptions will cause processing to stop on the SObject collection.
Sending a Platform Event, namely, ACCC_Log_Event__e, is the default. There must be an entry in the Platform Event Metadata if you wish to control behavior. For example, without a proper entry, the event WILL NOT write to a Big Object (see store handler and is active).	See here.
Unit Tests depend on the JSON metadata and associate SOQL statement. Should you change the JSON metadata or SOQL to pull in different columns/fields not found, Unit Tests will Fail	Keep your Unit Tests updated and checked every time you MAKE a CHANGE
Limitations with Platform Event have a maximum text size of 131072. Within ACCC_Log_Event__e, the Payload__c uses the maximum size; however, the Payload__c is splited into multiple platform events. The Total Number will reflect the multiple platform events and the Transaction Id will be the SAME for both platform events; though, the Message Id will be unique and the Sequence number reflects a monotonic increasing sequence.	• In synchronous mode pass in a smaller number of SObject collections to the service • In an asynchronous mode use a smaller chunk size.

How to have a Platform Event write to a Big Object

1) Go to ACCC Platform Event Binding (custom metadata)

2) Add the event name (i.e. ACCC_Log_Event__e)



Ensure the store handler is set to , *accc_DefaultProcessStoreHandler*. The default handler which utilizes the underlying logging platform event information is done via *accc_ApexUtilities.log*.

This uses the underlying Apex Cross-Cutting-Concerns custom metadata to determine the sink. Refer to the Wiki for more detail on this topic.

Enhancements

There is plenty of room for enhancements (as this was written in a short time). Providing more flow control and dependency injection may be an area for enhancement. The builder is not as abstract as one may like; however, the components the design provide dependency injections to help vary behavior.

Additional items

- Refactor pulling the metadata relationship via a selector
- Better abstraction regarding the metadata builder
- Use of a metadata relationship to pull in the cross-references
- Refactoring aspects of the code into base classes
- Performance metrics with stress testing to get an idea of volume
- Log Entry does not account for more than one entry. Larger volumes could have multiple Log Entries of the same domain to provide greater flexibility.
- Instead of JSON metadata consider providing a list of fields to extract in a separate custom metadata relationship.
- HEAP/CPU/SOQL Limits are possible due to the JSON being generated (> 6M bytes in synchronous mode). There is the ability to chunk records and process (~## records / chunk). However, the metadata handler (*accc_MetadataDataHandler*) does not throttle payload (JSON) size. Needs refactoring to accommodate Bulk.

The design, implementation and documentation were done in a short time. Refactoring will be required!

Metadata Screens

There are four log event custom metadata objects. The reason there are four different custom metadata objects is to allow one to use the functionality of the custom metadata UI screen. Technically, it could be done in fewer custom metadata; however, the thought was to allow for a UI experience over a copy-paste experience.

The four different custom metadata objects are listed in the order one should follow when creating new log event entries. Please note, there are 2 custom metadata objects (screens) not needed if there are **NO cross reference objects**.

ACCC Log Event Metadata [1]

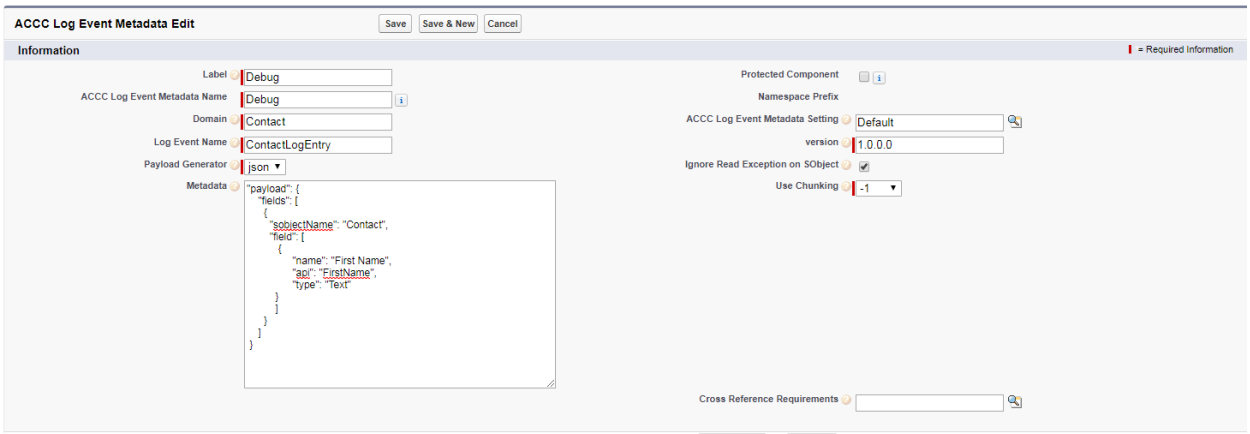
This custom metadata is the *start* of the Generic Log Event Metadata. If there are no cross references, the only other custom metadata entry you will need is the *ACCC Log Event Metadata Setting.* However, the core functionality will use the '*Default*' entry.

The other fields, excluding the cross reference custom setting are as follows:

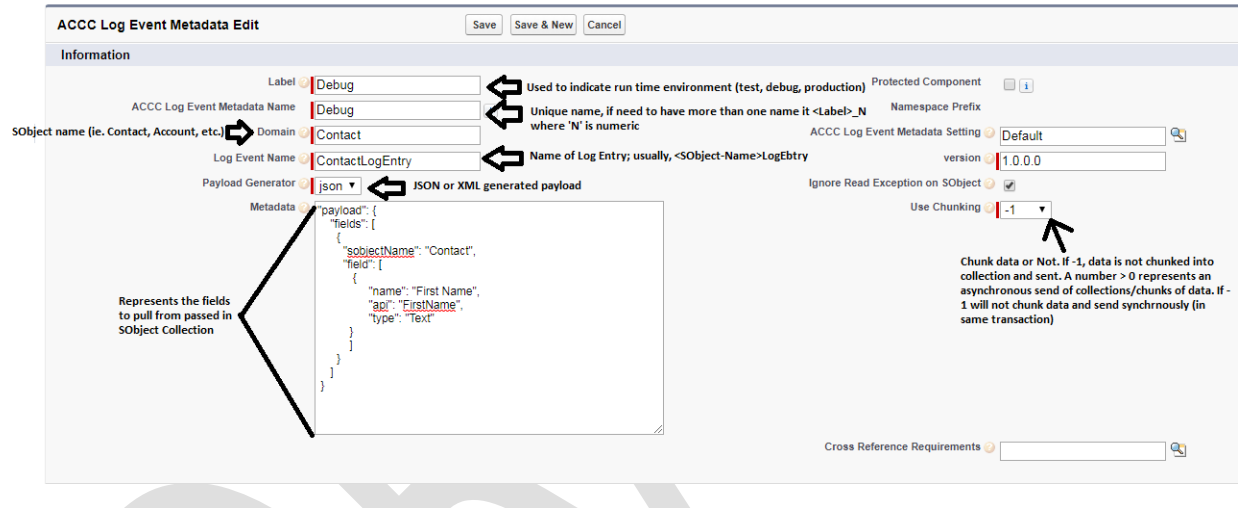
Name	Comment
Label	Should be either *Debug*, *Test* or *Production*. This field represents the runtime environment
Name	This represents the unique identifier. My recommendation is to follow the following:

Name	Comment
	<Label>_N where N is a monotonic number (i.e., 1,2,3, …). For example, *Debug*_1, *Debug*_2, etc.
Payload Generator	This field indicates which generator the Factory will use to generate the output. This value is either *json* or *xml*. The ACCC's default is *json*.
Domain	Is the SObject Name. It is important you spell the name correctly as it is checked for validity. For example, if you use a Domain name of *'cyntact'* it would cause an exception as it is not a valid domain (SObject Name)
Log Event Name	Is the name used in the Payload Generator. The recommendation is to use the Domain name as the prefix and '**LogEntry'** as the suffix. For example, Contact**LogEntry.**
Version	Represents the Payload version. It is injected into the generated payload
Ignore Read Exception on SObject	This checkbox indicates whether to ignore a read/get exception on a field from an SObject . The system reads all SObject with the *get(**api-name**) method*. If it is not in the SObject and the checkbox is unchecked, an exception occurs and processing stops. The Payload generated will be empty. Otherwise, it stores the exception in the Payload exception field and continues.
Use Chunking	A negative value (-1) indicates to process within the same transaction (synchronous). Any other value, tells the processor to break the SObject into specific chunks and process those chunks asynchronously. For example, if there are 100 SObjects passed into the process with a Chunk Size of 50, the SObjects are broken into 2 chunks of 50 and processed asynchronous. The reason for this is simple (see Limits section for more details): 1. Use asynchronous limits (i.e. HEAP doubles, CPU increases, etc.) 2. Increase volume However, there are other caveats that come into play; namely, the number of queueable allowed (currently 50).
ACCC Log Event Metadata Settings	Indicates the Reader and Runtime Handler. The de facto is ***Default***. This is not likely to change anytime soon. However, it provides additional ways to change the behavior of the Log Event process.
Cross Reference Requirements	If the SObject Domain brings in other SObject components or addition processing, select the appropriate cross reference component. See cross reference component section.

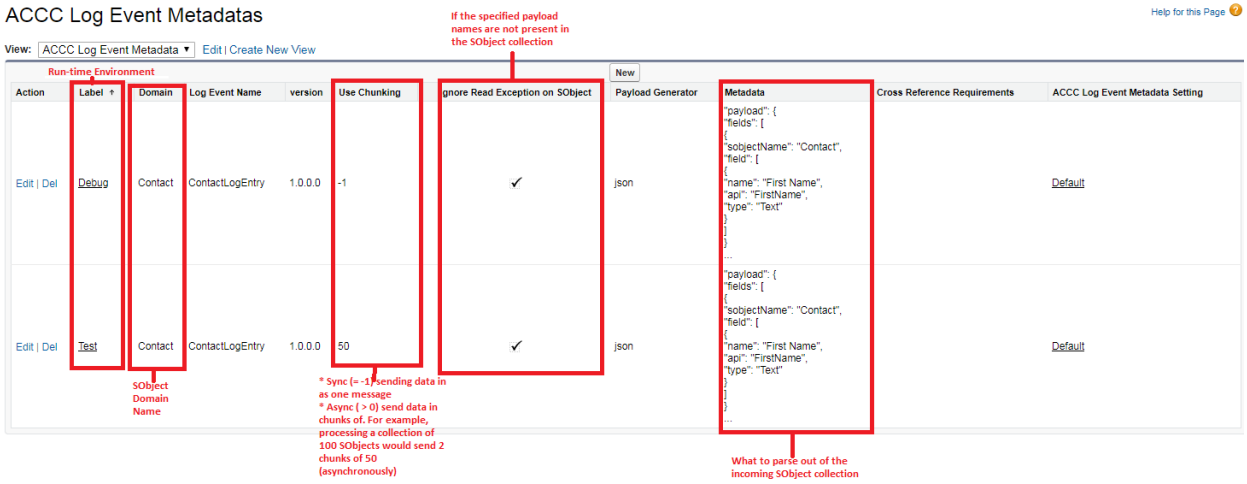
ACCC Log Event Metadata



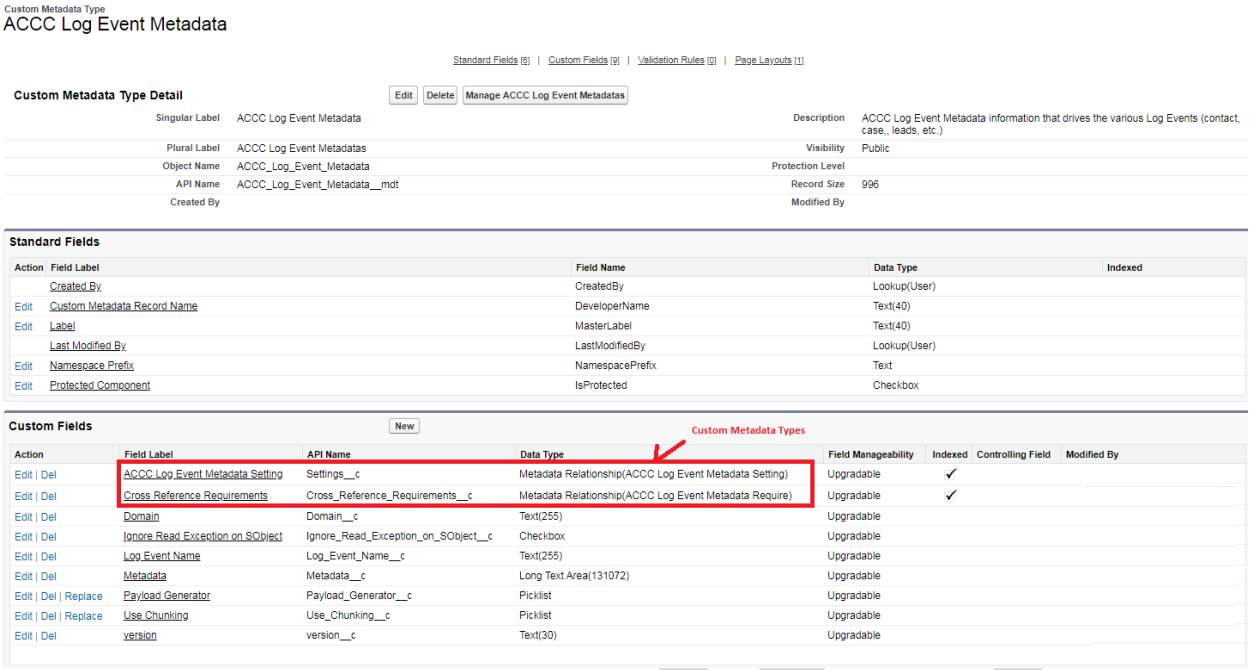
ACCC Log Event Metadata



ACCC Log Event Metadatas



The schema definition for **ACCC Log Event Metadata**



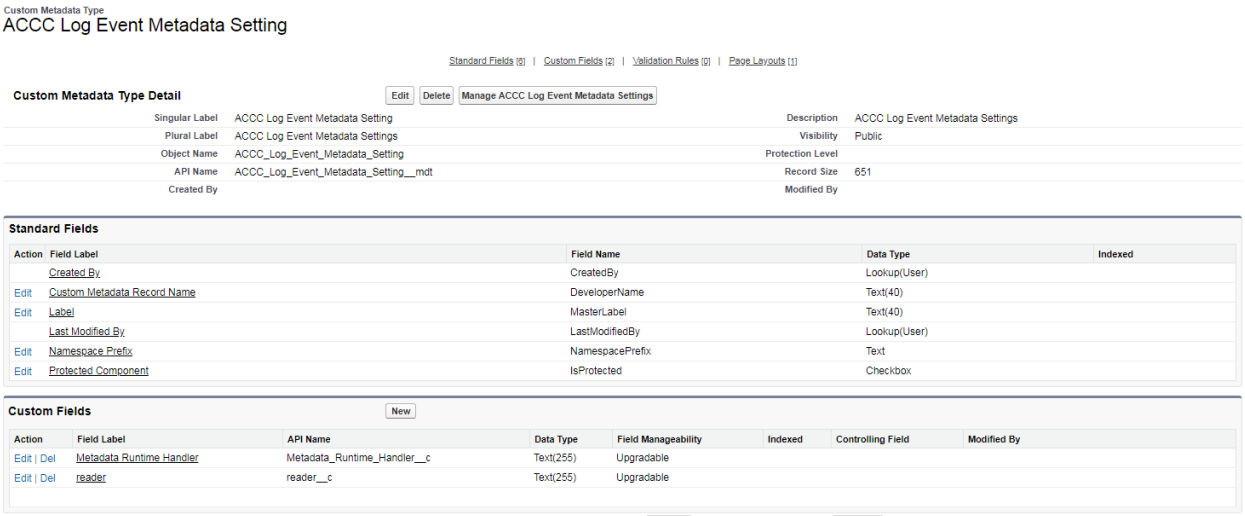
ACCC Log Event Metadata Setting [2]

ACCC Log Event Metadata Setting

There is currently a ***Default*** defined. There is only one entry and should be used until a change in processing or reading of metadata is needed. The above fields are as follows:

Name	Comment
Name	Unique Name
Label	Will be the same as the Name
Metadata Runtime Handler	Provides the components to process the metadata. The default is *accc_MetadataHandler*. This value is not likely to change. However, you have the ability to define a variation of the processing (if needed).
Reader	Reads the (JSON) Metadata. The default value is *accc_MetadataDefaultReader*. This value is not likely to change. However, you have the ability to define a variation of the processing (if needed).

The schema definition for **ACCC Log Event Metadata Setting**



Custom Metadata Type
ACCC Log Event Metadata Setting

Standard Fields [8] | Custom Fields [2] | Validation Rules [0] | Page Layouts [1]

Custom Metadata Type Detail Edit | Delete | Manage ACCC Log Event Metadata Settings

Singular Label	ACCC Log Event Metadata Setting	Description	ACCC Log Event Metadata Settings
Plural Label	ACCC Log Event Metadata Settings	Visibility	Public
Object Name	ACCC_Log_Event_Metadata_Setting	Protection Level	
API Name	ACCC_Log_Event_Metadata_Setting__mdt	Record Size	651
Created By		Modified By	

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	

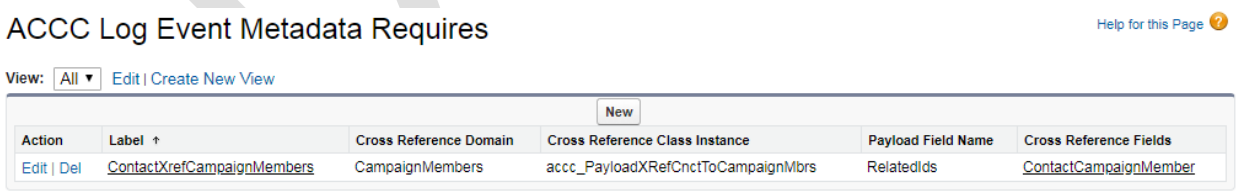
Custom Fields New

Action	Field Label	API Name	Data Type	Field Manageability	Indexed	Controlling Field	Modified By
Edit | Del	Metadata Runtime Handler	Metadata_Runtime_Handler__c	Text(255)	Upgradable			
Edit | Del	reader	reader__c	Text(255)	Upgradable			

ACCC Log Event Metadata Requires [3]

Requires metadata corresponds with the cross reference settings. At this time only one cross reference is defined (due to time constraints). The fields defined are as follows:

Name	Comment
Name	Unique Name (Identifier)
Label	Mimics the Name field
Cross Reference Domain	The SObject parent will reference
Cross Reference Class Instance	This represents the Apex class instance that performs the cross reference work. The example below shows ***accc_PayloadXRefCnctToCampaignMbrs***. The format followed is: ***accc_PayloadXRef****<DomainName>***To***<CrossReferenceDomainName>** *(abbreviate to accommodate for name length)*
Payload Field Name	The name of the payload output. In JSON, this represents an object. If XML, it would represent an element.
Cross reference Fields	Is a lookup of the cross reference fields custom metadata. This lookup will contains the field name, etc. See the cross reference field section.



ACCC Log Event Metadata Requires

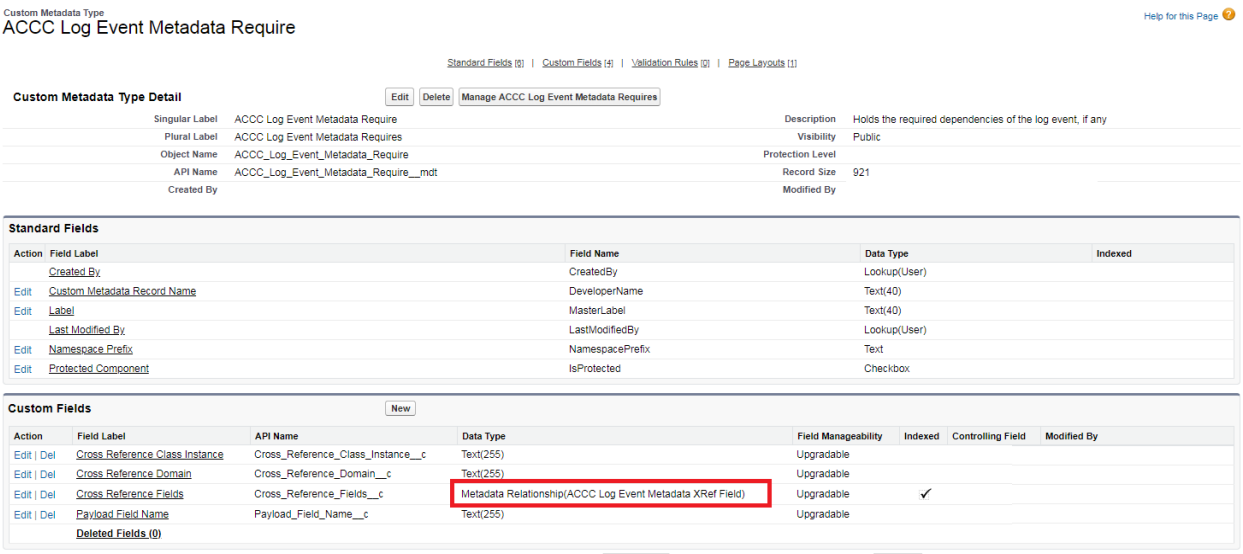
Help for this Page

View: All ▾ Edit | Create New View

New

Action	Label ↑	Cross Reference Domain	Cross Reference Class Instance	Payload Field Name	Cross Reference Fields
Edit | Del	ContactXrefCampaignMembers	CampaignMembers	accc_PayloadXRefCnctToCampaignMbrs	RelatedIds	ContactCampaignMember

The schema definition for **ACCC Log Event Metadata Requires**



Custom Metadata Type
ACCC Log Event Metadata Require

Help for this Page

Standard Fields [6] | Custom Fields [4] | Validation Rules [0] | Page Layouts [1]

Custom Metadata Type Detail Edit | Delete | Manage ACCC Log Event Metadata Requires

Singular Label	ACCC Log Event Metadata Require	Description	Holds the required dependencies of the log event, if any
Plural Label	ACCC Log Event Metadata Requires	Visibility	Public
Object Name	ACCC_Log_Event_Metadata_Require	Protection Level	
API Name	ACCC_Log_Event_Metadata_Require__mdt	Record Size	921
Created By		Modified By	

Standard Fields

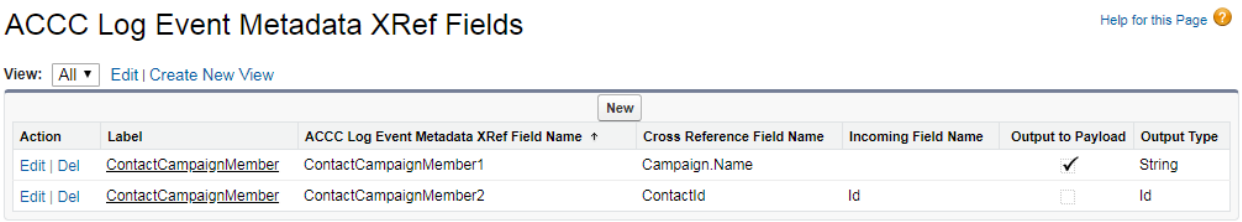
Action	Field Label	Field Name	Data Type	Indexed
	Created By	CreatedBy	Lookup(User)	
Edit	Custom Metadata Record Name	DeveloperName	Text(40)	
Edit	Label	MasterLabel	Text(40)	
	Last Modified By	LastModifiedBy	Lookup(User)	
Edit	Namespace Prefix	NamespacePrefix	Text	
Edit	Protected Component	IsProtected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

Custom Fields New

Action	Field Label	API Name	Data Type	Field Manageability	Indexed	Controlling Field	Modified By
Edit | Del	Cross Reference Class Instance	Cross_Reference_Class_Instance__c	Text(255)	Upgradable			
Edit | Del	Cross Reference Domain	Cross_Reference_Domain__c	Text(255)	Upgradable			
Edit | Del	Cross Reference Fields	Cross_Reference_Fields__c	Metadata Relationship(ACCC Log Event Metadata XRef Field)	Upgradable	✓		
Edit | Del	Payload Field Name	Payload_Field_Name__c	Text(255)	Upgradable			

Deleted Fields (0)

ACCC Log Event Metadata XRef Field [4]



ACCC Log Event Metadata XRef Fields

Help for this Page

View: All ▾ Edit | Create New View

New

Action	Label	ACCC Log Event Metadata XRef Field Name ↑	Cross Reference Field Name	Incoming Field Name	Output to Payload	Output Type
Edit | Del	ContactCampaignMember	ContactCampaignMember1	Campaign.Name		✓	String
Edit | Del	ContactCampaignMember	ContactCampaignMember2	ContactId	Id		Id

Note on cross/related reference objects, the field name is ***Campaign.Name***. This implementation is set up to allow cross/related objects to be accessed (i.e. *Account.Name* on a Contact query). For example, the following SOQL,

SELECT CAMPAIGN.NAME, CONTACTID FROM CAMPAIGNMEMBER

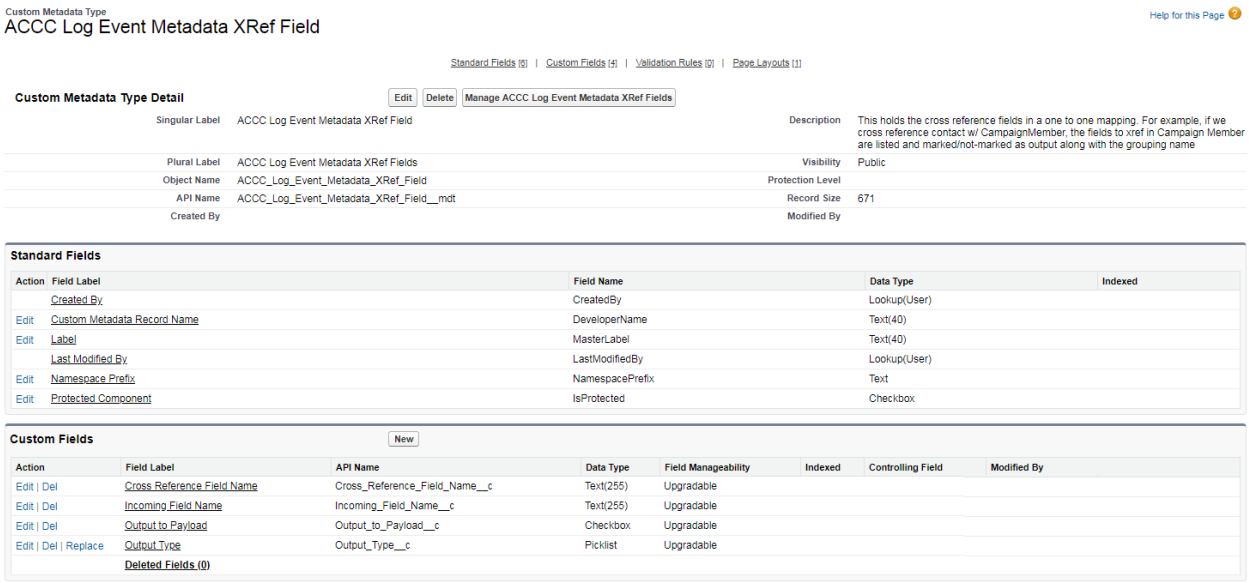
allows the related object, *Campaign*, to be accessible from the *CampaignMember*. However, the above campaign name is NOT accessible with normal dot-notation, i.e. *<object>.get('Campaign.Name').* As such, the implementation, grabs the name,

<SObject >.getSObject('Campaign').get('Name')

Please note, the above is just an example of how one gets a related object field value.

The main point is that when accessing related objects, the system can traverse/navigate the cross reference field name to obtain the underlying value.

The schema definition for **ACCC Log Event Metadata XRef Field,**



Custom Metadata Type
ACCC Log Event Metadata XRef Field

Help for this Page

Standard Fields [6] | Custom Fields [4] | Validation Rules [0] | Page Layouts [1]

Custom Metadata Type Detail Edit | Delete | Manage ACCC Log Event Metadata XRef Fields

Singular Label	ACCC Log Event Metadata XRef Field	Description	This holds the cross reference fields in a one to one mapping. For example, if we cross reference contact w/ CampaignMember, the fields to xref in Campaign Member are listed and marked/not-marked as output along with the grouping name
Plural Label	ACCC Log Event Metadata XRef Fields	Visibility	Public
Object Name	ACCC_Log_Event_Metadata_XRef_Field	Protection Level	
API Name	ACCC_Log_Event_Metadata_XRef_Field__mdt	Record Size	671
Created By		Modified By	

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	Cross Reference Field Name	Cross_Reference_Field_Name__c	Text(255)	Upgradable			
Edit | Del	Incoming Field Name	Incoming_Field_Name__c	Text(255)	Upgradable			
Edit | Del	Output to Payload	Output_to_Payload__c	Checkbox	Upgradable			
Edit | Del | Replace	Output Type	Output_Type__c	Picklist	Upgradable			
	Deleted Fields (0)						

Cross References

The system currently supports one-level deep on cross references. For example, the Contact cross references Campaign Members to determine the Campaigns the contact is a member. There are some caveats which one needs to be aware of creating new cross references:

1. Cross references cause additional HEAP Size ,CPU processing and SOQL Queries
2. Cross references **should** be Bulkified
3. Cross references must inherit from the abstract class, *accc_AbstractMetadataRequires*
    a. It is important to understand the purpose of the two methods (see next section):
        i. preProcess(List<SObject> incoming)
        ii. payload(SObject item)

Abstract class for cross reference

The abstract class, *accc_AbstractMetadataRequires*, defines two methods for the child class to *override*.

First**, *preprocess(List<SObject >),*** is invoked by the (JSON) generator upon entry to perform the necessary Bulkification. Otherwise, handling a large volume of data (SObjects) will cause a SOQL Query Limitation. Second, once the *preprocess* method is called, the (JSON) generator calls the *payload* method to generate the appropriate (JSON) Payload per SObject entity for the cross reference.

If the pre-processing of the SObject collection is not done, within synchronous call, it should be noted it could be because one can perform up to 100 SOQL queries (the current limitation). The only cross reference implementation is *accc_PayloadXRefCnctToCampaignMbrs*. What this class does is performs one SOQL query looking for ALL Contact Ids within the Campaign Members and indexes the results by Contact Id. The *payload* method will lookup the incoming Contact Id and return the Campaigns the Contact is a member.

Limits

Limits can be encountered (as they **CANNOT** be caught). The most prominent exceptions are outlined below. For more information on limits see this link.

HEAP Size

The Payload generator currently generates a JSON package. The 6M bytes limitation (sync mode) will be exceeded assuming more than 50 *SObjects* and the number of fields pushed into the payload. In addition, cross references will add to this volume. Thus, the default number of *SObjects* to process should be determined empirically. This can be configured in the custom metadata (see *Use Chunking*). It should be noted that using an asynchronous mode (i.e. *Use Chunking* != -1), the HEAP size doubles to 12M bytes and the CPU time increases.

CPU Size

As with the HEAP limitations CPU can come into play as well. However, you are more than likely to run into HEAP or SOQL Limitations. (See Heap section). It should be noted that using an asynchronous mode (i.e. *Use Chunking* != -1), the CPU size climbs to 60 seconds (instead of 10 seconds).

Asynchronous Mode

Using a Chunk size > -1, allows for the operation to behave asynchronously. Once this limit is reached a limit violation will occur (and you **CANNOT** catch these limit exceptions).

DML Statements

The SOQL queries become prevalent with cross references. You are not likely to run into DML limitations should there be no cross references (and assuming there are no other queries prior to invocation). It is difficult to determine the number of cross-references as this is still being evolved.

General Limits Comments

Overall, SOQL query limit as well as other limits can come into play. The payload generator will parse the metadata along with the SObjects. If there are **NO** cross references (Xref) you should not run into **SOQL query**[1] limitations (*just DML, CPU or HEAP* Limits). However, should there be cross references it **CAN** cause SQOL requests which can cause *SOQL limit exceptions*.

There is a caveat to the previous statement; namely, it is undetermined (and not yet tested) how the processing will be called. If there are already a large number of DML statements, CPU, HEAP ,etc. prior to calling the processing of the SObject collection then you will need to adjust the chunking (and, probably, a smaller chunk size). Much can be determined with **proper testing**.

At this time, there is no throttle to avoid HEAP issues for Bulk (i.e. 6M max. heap in synchronous mode). There are some techniques that could be used to partition the processing; however, it is up to ACCC to work through. For example, when processing SObjects, check for the payload size and create multiple sequences. The Payload (in the event [***ACCC_Log_Event__e***]) has *'Sequence* and *'TotalNumber'* fields to accommodate this aspect.

The maximum number of SObjects to process will vary (more or less depending on the data element size). In addition, to handle larger chunks of data, one should consider using

[1] Depends on how efficient ones SOQL(s)

accc_MetadataLogEventChunked class. Which will chunk the data if the chunk size (from the custom metadata *'Use Chunking*' is set to a positive value, i.e. not -1) and process asynchronously . This is handled automatically in *accc_MetadataLogEventService*.

Sample Runs

The following data provides a indication of performance though the environment is an isolated Developer Sandbox running the EMP_CONNECTOR (external cometD client) listening for the Log Event. In addition, the cross reference data only amounted to an additional 40 bytes per SObject entry (if any).

The test runs were done with the Contact metadata which (given the fields) average approximately ***300*** bytes per reference. For example, the test metadata schema pulls 10 fields from the Contact. These fields plus extraneous characters, consume approximately 300 bytes. Thus, 20 Contacts would use approximately 6000 bytes within the JSON payload. Given 6M bytes of HEAP (synchronous mode), you could consume 20,000 Contacts. However, the CPU or additional storage consumption would cause a limit exception.

A *synchronous environment* is one in which the processing of the SObject into a Log Event is run within the same transaction (context) as the invoker. An *Asynchronous environment* is one in which the processing of the SObject into a Log Event is run within the **separate transactions** (context) of the invoker.

Number of SObject s	Process time (seconds)	Chunk Size	External Client Gets first Log Event	Comments
200	6.93	30	~2 seconds	
200	6.79	40	~2 seconds	
200	6.16	50	~2 seconds	
200	6.22	100	~2 seconds	
200	6.5	150	~2 seconds	
200	5.66	200	~2 seconds	
200	6.2	250	~2 seconds	
200	6.61	300	~2 seconds	
200	5.5	-1	~6 seconds	
200	6.29		~4 Seconds	**AVERAGE [All]**
200	5.5	Sync	~6 Seconds	**AVERAGE [Sync]**
200	6.38	Async	~2 Seconds	**AVERAGE [Async]**

The above data reflects the following observations:

- Synchronous is faster processing (in Salesforce) but longer delay until the first Log Event received by the external client
- Asynchronous is ~16% slower but faster (300%) in sending the first Log Event to the external client.
- Increasing the Chunk size did not substantially improve the Salesforce processing speed

What is not accounted for in the data above is the increased probability of a Salesforce Limits exception running in a synchronous environment.

When exploring larger collection of SObjects (i.e., 500+) with no cross references (note times will vary on a multi-tenant environment),

Number of SObject s	Process time (seconds)	Chunk Size	External Client Gets first Log Event	Comments
500	~13.86	Async	~10 Seconds	**AVERAGE [Async]**
500	~11.13	Sync	~12 Seconds	**AVERAGE [Sync]**
1000	~21.2	Sync	~20 Seconds	**AVERAGE [Sync]**
1000	~28.3	Async	~15 Seconds	**AVERAGE [Async]**
2000	No Exception			**Sync**
2000	No Exception			**Async (chunk size == 400)**
2500	No Exception (Though, *Payload__c* is split due to JSON MAX size into multiple Platform Events			**Async (chunk size == 1000)**
Maximum Amount is TBD				
>2500	At some point (ACCC to determine threshold as there are too many possible permutations). Note: Working with LARGE datasets has limitations. This is a known fact within Salesforce when trying to perform LARGE volumes of datasets when triggers are NOT disabled.			**Async (chunk size == TBD) ← ACCC TBD**

The above data reflects the following observations:

- Whether synchronous or asynchronous the Salesforce processing time and receiving times are directly proportional the number of SObjects.
- Larger volumes are obtainable via asynchronous mode (i.e. Chunking). However, it will require experimentation/testing with the appropriate chunk-size. The most common issue will be CPU and HEAP exceptions along with possible truncation of the *Payload__c* (JSON) due to the maximum string size allowed (131,072)

How to Determine Optimal Size and Mode

First determine the maximum synchronous size you can handle. Once you know this, if you decide to chunk the data, start with the maximum synchronous chunk size and work backwards on the chunks as needed. It could happen that you will determine the maximum number of SObjects the system can handle asynchronously.

Where is most of the work (CPU)?

Most of the work is performed in the payload generator (***accc_PayloadGeneratorJSON***). The processing iterates over the collection of SObjects (could be *Contact*, *Account*, *Case*, *Lead*, etc.) extracting the fields list (as defined in the custom metadata). The JSON payload is primarily made up of an array of fields (<name>, <value>) pairs :

```
[
 {
   "First Name":"JI",
   "Last Name" : "SMITH"
              :
 },
     :
 {
   "First Name":"Tom",
   "Last Name" : "Jones"
              :
 }
]
```

The above JSON is generated through a loop of the SObject collection in ***accc_PayloadGeneratorJSON:***

```
System.JSONGenerator.writeStartObject()
  + accc_AbstractPayloadGenerator.getItemFromSObject(SObject, String, String)
  + System.JSONGenerator.writeStringField(String, String)
  + accc_AbstractPayloadGenerator.writeNameValue(System.JSONGenerator, SObject, String, String, String)
  + accc_AbstractPayloadGenerator.getItemFromSObject(SObject, String, String)
  + System.JSONGenerator.writeStringField(String, String)
     :
                    :
  +|accc_PayloadGeneratorJSON.processXRefEntry(SObject)
System.JSONGenerator.writeEndObject()
```

If you want to look for improvements consider how to improve the above sequential operations (loop).

Overall

Given the information above, the timing gains of using synchronous over asynchronous are not substantial. However, there is a greater degree of avoidance of Salesforce Limits when using asynchronous processing. And finally, these timings were not perform within a trigger transaction (but within a Developer Console); thus, any previous HEAP, CPU, DML, etc. limits would impact the provided timings.

Recommendation

The recommendation would be to use asynchronous processing; however, the optimal chunk size has to be determined empirically through testing as the content (JSON) will vary.

Appendix: Platform Events

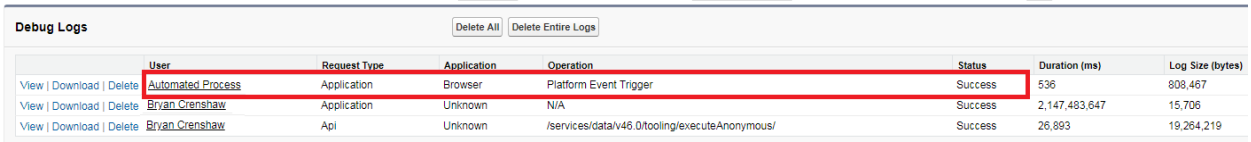
This document does not go into the details of debugging Platform Events on Salesforce Platform as there are plethora of Salesforce pages for this (see here).

However, a common fault many developers skip about getting the logs from a Platform Event is Automated Process Entity. Ensure you set within the Debug Log, register for **Automated Process** logs (see below).

User Trace Flags — New | Add Current User

Action	Name	LogType	Requested By	Start Date	Expiration Date	Debug Level Name
Delete | Edit | Filters	Cloud Connector, Marketing	USER_DEBUG	Jason Scott Sharpe	1/12/2017 12:26 PM	1/12/2017 1:26 PM	integration
Delete | Edit | Filters	Crenshaw, Bryan	USER_DEBUG	Bryan Crenshaw	8/2/2018 11:45 AM	8/2/2018 11:50 AM	Debug
Delete | Edit | Filters	Donnell, Jeremy	DEVELOPER_LOG	Jeremy Donnell	2/12/2019 2:12 PM	2/12/2019 2:17 PM	SFDC_DevConsole
Delete | Edit | Filters	Integration, Flosum	DEVELOPER_LOG	Flosum Integration	11/10/2017 1:51 PM	11/10/2017 2:16 PM	SFDC_DevConsole
Delete | Edit | Filters	Lavis, Ronald	USER_DEBUG	Jason Scott Sharpe	6/10/2018 5:00 PM	6/10/2018 9:00 PM	integration
Delete | Edit | Filters	Leugs, Fisher	USER_DEBUG	Hima Bindu	9/25/2018 10:50 AM	9/25/2018 12:50 PM	integration
Delete | Edit | Filters	Mangar, Kamla	USER_DEBUG	Jason Scott Sharpe	4/19/2016 9:45 AM	4/20/2016 9:15 AM	Workflow
Delete | Edit | Filters	Mello, Joe	USER_DEBUG	Jeremy Donnell	5/24/2016 2:07 PM	5/25/2016 2:07 PM	Debug
Delete | Edit | Filters	Morris, Cassandra	USER_DEBUG	Jason Scott Sharpe	6/11/2018 11:05 AM	6/11/2018 1:35 PM	integration
Delete | Edit | Filters	Padamati, Anusha	DEVELOPER_LOG	Anusha Padamati	9/10/2019 11:51 AM	9/10/2019 12:32 PM	SFDC_DevConsole
Delete | Edit | Filters	Platform Integration User	USER_DEBUG	Bill Anderson	8/11/2019 3:51 PM	8/12/2019 3:51 PM	SFDC_DevConsole
Delete | Edit | Filters	Powell, John	USER_DEBUG	Jason Scott Sharpe	6/10/2018 3:59 PM	6/10/2018 5:59 PM	integration
Delete | Edit | Filters	Process, Automated	USER_DEBUG	Jeremy Donnell	10/3/2019 8:51 AM	10/4/2019 8:51 AM	refine
Delete | Edit | Filters	Sharpe, Jason Scott	DEVELOPER_LOG	Jason Scott Sharpe	5/13/2019 10:54 AM	5/13/2019 11:35 AM	SFDC_DevConsole

Once you set the start/end date for **Automated Process,** once a Platform Event is published you will see the following:



Debug Logs — Delete All | Delete Entire Logs

	User	Request Type	Application	Operation	Status	Duration (ms)	Log Size (bytes)
View | Download | Delete	Automated Process	Application	Browser	Platform Event Trigger	Success	536	808,467
View | Download | Delete	Bryan Crenshaw	Application	Unknown	N/A	Success	2,147,483,647	15,706
View | Download | Delete	Bryan Crenshaw	Api	Unknown	/services/data/v46.0/tooling/executeAnonymous/	Success	26,893	19,264,219

Appendix : Salesforce Platform Events

Salesforce largest text field size in a Platform Event is a Long Text (131,072). As a result, the ***Accc_Log_Event__e*** contains a (Long Text) *Payload__c* field. If the bytes size of the (JSON) Payload exceeds this amount, the system splits the JSON/XML into appropriate chunks that the external system has to stitch back together.

The Platform Event contains Total Number (*Total_Number__c*) and Sequence Number (*Sequence_Number__c*) along with a distinct Message Id. The external client needs to look at the Total Number (*Total_Number__c*) to determine if the payload is complete (*Total_Number__c* equals 1). Otherwise, use the Sequence number of the Platform Events (the *Transaction_Id__c* is the same for all related Platform Events) and append the Payload together.

Appendix: JSON

The core JSON metadata is shown below (abbreviated for readability). It may contain cross reference field(s). Cross reference fields may or may not be present. The cross reference field name will vary as it can be any name ACCC chooses (as defined in the custom metadata). The current JSON metadata generates approximately 300 bytes per SObject entry (*only have Contact to go by*). Except for CPU limits one could process well over 10K items and not encounter HEAP limits.

Pre-Amble

A pre-amble is added to the JSON payload which is pulled for the custom metadata so that you only need the payload information.

```
  "logName": "ContactLogEvent",
  "version": "1.0.0.0",
  "domain": "Contact",
```

Payload

A payload describes the expected information (*name*, *api*, *type*) to pull from the SObject list.

```
   "payload": {
    "fields": [
     {
       "sobjectName": "Contact",
       "field": [
         {
             "name": "First Name",
             "api": "FirstName",
             "type": "Text"
       },
              :
       {
             "name": "Last Name",
             "api": "LastName",
             "type": "Text"
       }
      ]
     }
    ]
}
```

Final JSON

The final JSON passed to the JSON2Apex parser is shown below.

```
{
  "logName": "ContactLogEvent",
  "version": "1.0.0.0",
  "domain": "Contact",
   "payload": {
    "fields": [
     {
       "sobjectName": "Contact",
      "field": [
        {
            "name": "First Name",
            "api": "FirstName",
            "type": "Text"
       },
                :
       {
            "name": "Last Name",
            "api": "LastName",
            "type": "Text"
       }
      ]
     }
   ]
  }
}
```

For example, Contact may have a cross reference field, *RelatedCampaigns*, (see accc*_PayloadXRefCnctToCampaignMbrs).* The sample below is an example JSON cross reference data generated.

```
{
  "logName": "ContactLogEvent",
  "version": "1.0.0.0",
  "domain": "Contact",
   "payload": {
    "fields": [
     {
       "sobjectName": "Contact",
      "field": [
        {
            "name": "First Name",
            "api": "FirstName",
            "type": "Text"
       },
                :
       {
            "name": "Last Name",
            "api": "LastName",
            "type": "Text"
       },
       "RelatedCampaigns":["7013B000000PIdqQAG","7013B000000PIdlQAG"]
       }
      ]
     }
   ]
  }
}
```