Event Governance At Scale

A Practical Guide To Effective Message Design

Timothy S. Hilgenberg

Version 0.9.9, October, 2022

Table of Contents

Colophon

This book is a very early draft of this manuscript

This book is being written in AsciiDoc using Visual Studio Code. I’m just learning this tool, so I apologize in advance for any bad formatting issues with early versions of the book.

Special thanks to the CloudEvent working group for their review and comments for this book. I would especially like Doug Davis for his support and feedback.

In the examples, HilcoTech is a mythical company.

The author’s experience has been in working with large Enterprise Application, such as Human Resources (HR), Benefits Administration, Payroll, Customer Relationship Management and ERP. These applications tend to have major independent business objects with deeply nested data graphs, along with a high degree of business processes that manage the data objects. This book should helps architects and event designers coming from these domains. This is very different than the domains that produce events from real-time environments like cars or sensors.

All comments welcome

Early Working Draft Version 0.9.8 — October 2022

Preface

This book describes a practical approach for defining standards for asynchronous messaging for a large complex company who is committed to implementing an Event Driven Architecture. The goals of the book are:

- Provide a solid framework for the definition of messages types to assist designers in creating high quality messages: Events and Commands
- Provide a specification for these event types that can provide standardization within the complex scale of large organizations
- Provide a event governance process that can be used to ensure message design consistency and quality design.

It is a language agnostic interface description for asynchronous message processing. The specification is also intended to be message platform agnostic and does not specify how events could be delivered through various industry standard protocols (e.g. HTTP, AMQP, MQTT, SMTP), open-source protocols (e.g. Kafka, NATS), or platform/vendor specific protocols. CloudEvents.io discusses these aspects of event processing in much greater detail. Although there are other serialization formats (binary, XML), the specification focuses on using JSON as it’s primary format. It should be an easy exercise to convert these specification into the other serialization formats. The goal within an organization should is theseo have all messages published by any internal producer conform to these standards. The specification also provides consumer of the messages with the detailed information they need to properly understand, consume and process these messages. It is also agnostic to the "cloud", the focus being on quality event design, which should be able to run in any commuting environment.

CloudEvents.io Compatibility

In general, this specification has the same philosophy as the CloudEvents.io specification. It can be considered a supplement to the CloudEvents specification, extending it to support a large enterprise organization’s Event Driven Architectures. As stated above, it does not address the various protocols to deliver and process the message, but extending the semantics to support scale. (i.e number and complexity of event definitions within an organization) Most of these attributes would be considered *extensions* in the CloudEvents specification and placed in that category. *(Ed. This statement needs to be validated by the group)*

Also, the CloudEvents specification focuses on Events, where as this specification also includes Commands and Audit type messages.

See Appendix for more details on the mapping of the CloudEvents core attributes to this specification.

What’s in the Book?

Chapter 1

```
Events are everywhere, yet event publishers tend to describe events differently
  - CloudEvents.io Project <<cloudEvents>>
```

Why is Event Driven Architecture critical to tomorrow’s applications?

The old, but tired and true, intra-company interaction model of overnight batch files is being replaced by newer, faster approaches. The flow of data was characterized by full file populations, byte stream multi record format and daily or longer delivery schedules. Another way to look at this approach is the model is (1) high latency, (2) uses old binary formats and (3) sends a large amount of data that might not have changed. However, they were operationally efficient, guaranteed delivery (no transaction were lost) and event based. (These were actions that already occurred.) More modern applications are requiring lower latency (sometimes referred to as "near-real time"), modern formats like JSON and XML and changes only. But, they still want the guaranteed delivery and operationally efficiency of batch offline processing.

To meet these new requirements, organizations are turning to "API" for large group or batch processing. They have had good success creating web-based user API that are short lived and managed by the user themselves. The user is able to manage any of the issues or errors presented by the application. REST APIs are a well established approach for data integration of smaller units of work. They are simple to consume by clients and easy to build and publish. There is lots of tooling and technology support in this space and a large cohort of practitioners who know how to leverage "API" technology. They can delivery relatively large payloads with a very low latency and are an effective mechanism to deliver changes in state of business objects. This why they are so effective in e-commerce application. However, given that they are HTTP based and because of that,they are not 100% guaranteed. When used to simulate batch processing, they can cause resource contention and operational issues for web based infrastructure.

Despite these concerns, there is a desire to use HTTP REST API as a substitute for batch processing. The real time request-reply processing, use of JSON as a format and ease of development are attractive in creating simulated batch solutions. Also, instead of a full population file, the API can deliver data that has been changed. However, this places a large burden on the consumer of the API to mimic the operational complexities of batch processing. This included cursor management, checkpoint/restart and ACID qualities of data management (dirty reads, etc) It also requires the providers of the service to have much larger web based compute capacities (i.e. servers and threads) for longer running requests because threads are bing held for longer periods of time. This is also open to more network issues and ability to get the entire workload processed.

Where do events and messaging come into play?

As the service ecosystem continues to grow and more and more applications are taking advantage of them, the need for tighter integration between applications continues to become more and more important. To handle more complex and integrated business processes, companies are looking for changes only, low latency and guaranteed delivery of changes or occurrences from systems of record. They are looking for events in real-time. There are two models for moving events between organizations.

(1) Consumers poll for events
(2) Systems of record publish Events

In the first model, the event publisher provides an API that allows a consumer to request a collection of events based on query parameters. Essentially, simulating an SQL Query. The consumer then can make REST API calls at pre-determined intervals to retrieve the events. This is an imperfect model because polling is an inefficient manner of integration. The consumer can either poll to frequently or not frequently enough causing inefficient resource utilization or data inconsistency. Data can be missed or the same data provided multiple times. Client need to be idempotent when processing the data. It can also can excessive resource utilization on the system of record.

The second model is a publish-subscribe paradigm. The system of record publishes an event to a middleware resource (i.e. queue, log) without any knowledge of who will consumer the message. The latency provided by the middleware is near instantaneous with out the need for polling. There are event middleware technologies like Kafka, Rabbit MQ and IBM MQ, that can provide these qualities of service in a production environment. However, at the moment these technologies do not work well between external organization.

Despite the issues around polling, a "polling" approach is good enough today to provide the reduced latency and guaranteed delivery of events from a system of record.

Event Driven Architectures can be used in data integration models where real-time, changes only data with guaranteed delivery needs to be exchanges between two organizations

In addition to new data exchange model, events are also being used more and more in User Experience applications. Today, the primary interaction model for web based architectures to communicate with other platforms is REST API via HTTP. This interaction paradigm is called **Synchronous Request-Reply**, where the client is *waiting* for a response from the service. This is driving the overall user experience, where the user presses a button or link and then waits for a response. As more mobile applications are calling multiple backend services in a single request, this is causing longer wait times for the user as it takes time for all the requests to respond. Ux Designers feel strongly that an synchronous response is the best user interaction model.

However, there are many application interaction models in practice today that do not fit this model. People may not think of it in this way, but email and text are really disconnected (asynchronous) interaction models. The person submits a communication message and then can perform other actions, while they wait for a notification that a response is available for review (e.g. waiting for an email response). If the asynchronous request is fast enough, the user does not know the difference. So there is precedence for user interface design using an asynchronous model.

So, why are interface designers not using a disconnected paradigm like email or text? These types of interactions are very hard to design for. Plus there is not great tooling support to implement this approach There are some situations, where the user expects an immediate response. It is hard in an asynchronous event driven model to simulate a synchronous response to the user. It requires a very low latency interaction approach, which was hard to do with older technologies. It was just easier to design a synchronous model when all the interactions were within the company computing ecosystem and there wasn't available technologies other than HTTP to provide an appropriate user response.

> *Event Driven Architectures are key to providing a rich satisfying user experience as applications become more complicated and more and mores services are being provided by outside providers*

A synchronous model also has a major impact on the computing resources required to support the application. Instead of having to provision servers for maximum requests, having one can have fewer workers in the background continuously managing the work. An event driven architecture following an asynchronous approach is always a more efficient use of computing resources than a asynchronous approach.

Asynchronous technologies have been around for a very log time. The IBM mainframe environment had MQ middleware that allowed inter-machine communication and intra-platform integration (primarily distributed platform). There are even more modern technologies, like Kafka and RabbitMQ. In prior application designs and even today, these technologies were used for inter-platform communication and more balanced resource allocation and management. There were mostly used for underlying application interaction, but not used in an interface design.

> *Event Driven Architectures are key to a more efficient use of computer resources. This is even more important in a cloud based environment*

where the pricing is based on actual detail usage.

One of the keys to designing modern event-driven applications is establishment of a set of event design standards and governance processes within the organization, along using asynchronous messaging technologies (IBM MQ, Kafka, RabbitMQ)
The purpose of this document is to provide a set of event specifications that can work at the scale of a large organization.

What is different in today's environment?

- Sophisticated application are leveraging more domain services that are being provided by parties who are outside of the control of the organization and the central application. This leads to more interactions and increases user experience wait time as the services are executed in a sequential manner.
- There are more modern messaging technologies are available to application architects to lower time time latency. There is also more practical real life experience in Event Driven Architecture for architects and designers to leverage
- User are becoming more comfortable with a disconnected interaction model, where they submit a request, work on a different task and then act on the response they get after the fact. Techniques, like deep linking from the response, to the specific page within the application are becoming more prevalent.
- Real time analytics being processed by AI and Machine Learning are becoming value tools for decision making within an organization. Events are an important technique in making it successful.

As with any other scaled integration strategy, the key to success is standards and event design needs to follow this strategy.

Why are event design standards critical for organizational success?

As an organization's application portfolio continues to grow, so does it's integration requirements between these application. In some cases, it is very difficult to create tightly integrated solutions. In addition, users are expecting their outcomes to move faster and having their entire process completed in a single session. Multiple applications are interested in the same events leading to increased complexity when using point to point approaches. Publish-subscribe approaches with events helps reduce this complexity. Requirements for more integrations, faster deployment, reduced complexity and low latency integration is leading to more and more use of events and event driven architecture.

The design of **events** can come in all shapes and sizes. They can be as low level as a single data element changing(changing one’s email address), to an entire business process completing (completing a transfer in a bank account). With the move to more domain oriented organizational structures, independent autonomous groups will be designing events. This will cause a prolific number of events being created by an organization. Without organizational design standards and guidelines, the lack of consistency will lead to chaos losing all the benefits of an Event Driven Architecture. Having multiple independent team developing their own event standards will lead to more integration code for mapping models and fields between applications, which lead to more cost, longer delivery times, more brittle code and more long-term technical debt. This may lead to additional processing cost, which might affect performance and require additional server purchases. This might be more acute in a cloud centric environment.

Event design standards are critical to the success of an Event Driven Architecture as the strategy is scaled up within the organization. It isn’t enough to commit to using events as a key integration strategy, the organization needs standards and the governance to enforce the standards in the design of events. To support friction-less and over-reaching governance, the organization needs comprehensive design guidelines to support the event designers and give rubrics to the governance groups on how to judge the quality of the design.

Having common event standards are key to creating programming language libraries and tooling. The creation of these artifacts will lead to faster development times, increased quality and improve interoperability across application platforms.

Why do standards and governance matter?

Interoperability

How do standards and governance support interoperability and tooling?

First, definitions.

message specifications	The message specifications are designed to provide a level of design consistency and quality in the design of messages within the organization. The focus here is too provide a starting point and guidance for design as organizations embark on Event Driven Architecture. The goal of the specification is to provide a glossary of terms, suggested structure and organization of the message and a preliminary list of fields names and field data types. Although the specifications suggests a JSON formats, the fields can be expressed in other formats (e.g. XML). This specification does not address the available protocols and language SDK.
message governance	Message governance is the enforcement of the specifications. Specifications without governance will negate the benefits of the specifications. The goal of governance is to insure quality message design, making sure the message meets the domain objectives. It also intended to insure that the message follows the specification. This insures message interoperability. Message governance is not intended to be a heavy handed process.

In order for messages to be interoperable, all application need to follow these specification. This will hopefully avoid any semantic mapping, where the same business object are modeled differently. It should also avoid any field mapping withing the application. This should lead to simpler code and even less code. Standardization also leads to the creation of tooling, which should increase productivity, quality and development time. Tooling can leverage the knowledge already baked into the specification.

In general, standards and tooling, should make the development of code and application interoperability less complex, less brittle, less costly, more agile with higher quality. This should enable speed to market and lower cost of ownership in the long term.

Chapter 2 - Message Definitions

Overview

This chapter provides the definitions of the types of the messages that could be supported within the organization’s messaging ecosystem. The goal of the chapter is to:

- define the types of messages and some of the guiding principles used to identify and name them.
- describe the potential fields definitions and formats for each of the message types.

Message Types

A **message** is domain packet of information (fact or request) that is typically processed in an asynchronous disconnected manner. It can be used to communicate between two parties or can be information shared in a publish/subscribe manner. A message is a general-purpose data structure with no special intent. As a component of the Event Driven Architecture, there are 3 types of asynchronous messages:

Events

An **Event** is a message which informs various interested parties or listeners about something which has happened in the past. It is sent by a producer which doesn’t know and doesn’t care about the consumers of the event. There can be multiple consumers of the event, each having their own interest in the data contained in the event. This type of messages promotes highly decoupled systems using pub/sub architectures.

An Event is an immutable record of a single event at a moment in time.

Commands

Commands trigger some action which should happen in the near future. It’s typically a one-to-one connection between a producer (who sends the command) and a consumer (who takes and executes the command) and, in a few cases, the order of commands is also of utmost importance. Commands are usually performed by actors outside the current system. However, commands can also be rejected, requiring new error handling patterns.

The difference in thinking between an event and a command is an event-X has occurred, rather than command-Y which should be executed as part of a conversation or interaction.

A Command is a request to retrieve some data or perform some action

Audit

An Audit message is an adhoc publishing of a domain business object’s state. There is no true triggering action, either from a business process or straight data change. It would typically be triggered in a batch fashion with a query predicate. As part of normal processing, there will be situations where there are failures in the pipeline, which might lead to data inconsistencies between a systems of record and systems relying on this data. In situations where the business objects are very stable and don’t change often, the audit message is used to get to eventual data consistency between systems. A full business object for a bounded context can be published on a periodic basis and then any consumer caches can be updated. This, can also be used to seed new consumers with domain data.

Event

Something of interest that has happened in the past

An **Event** represents something that has happened, along the data context, at a defined point in time.

Definitions:

- Result of some outcome
- Collins: a happening or occurrence, esp. when important
- CloudEvent Concepts
    - An event includes context and data about an occurrence. Each occurrence is uniquely identified by the data of the event.
    - Events represent facts and therefore do not include a destination, whereas messages convey intent, transporting data from a source to a given destination.

Event names should indicate a past tense action, a past-participle verb, and should be action oriented. Events are both a historical fact and a notification of an occurrence or happening to other interested parties:

- Notification - a call for action, this is considered a stateless event
- A state transfer - pushing data wherever it is needed, known as an Event-Carried State Transfer

Events never produce a response object when published. They could be the result of a business process (i.e. completed enrollment) or command (i.e. Change medical plan election). They can't be rejected, but can be ignored. There is no expectation of any future action by the publisher.

Events can varying in semantic granularity or scope. This can be more pronounced for events that reflect the data element changes in complex business objects They can be as small as a single field change in a business object or a notification that some element(s) changed within the business object. In this case, the event data within the message may or may not document the changes. There are also levels in between. For complex objects that are hierarchical in nature, the event may reflect a changes in a major sub-component of that data object.
For example, a data change for a "person" business object for an email address change could be:

- At the *person* level - Indicating something changes in the entity
- At the specific component: *home contact point* - Indicating a change in a contact point for

the person

- At the specific field level: *email address for home* - Indicating that the person’s home email address had changed in

> ❗ Events are immutable

Applications are like islands. Inter application processing is becoming more and more important. As applications become more specialized, business process will involve multiple services,requiring more integration to provide some specific aspect of the overall process experience. Although it is not common practice, every application should consider publishing events as part of it’s overall design and implementation. Today, it is a bit of an afterthought. Events and Commands should be first class elements of the application. Even if there is no requirement for this in the present for an application, it will be at some point in the future. Every quality application domain engine should provide API for request-reply processing and publish events for any downstream application. A good event design will anticipate what events might be of interest and publish them.

Command

Represents a request to perform an important action task or retrieve data

Definitions

- Webster Definition: *to direct authoritatively*
- Represents an intent to perform some sort of action

The requested action of the command has not yet happened (e.g. Change medical plan election). As opposed to events, where some action has already occurred. The commands should be named with an imperative verb. It has an explicit expectation that something (a state change or side effect) will happen in the future. Commands can be rejected, but it these cases it is important for any consumer to response with some form of message. Typically, this is part of a Ux conversation or program-to-program interaction. However, there can one-way requests or fire-forgot messages with no response.

Commands can be used in situations where an async request/reply is desired. However, this is not to be confused with a synchronous Request/reply like REST API. The message can provide a call back information, which can be used in an asynchronous conversation with another consumer.

Audit

Represents the current state of a business object - Published on specific schedule

Audit messages can be used to synchronous data between systems of records and any consumer who is dependent on that data

More details to come.

Chapter 3 - Event Message Specifications

Overview

The purpose of this chapter is to define an overall design specifications for **Events**. It forms the foundation for a set of design guidelines and standards for the design of an organization’s events. The chapter will provide suggestions for various components of the event - event common header, an event category header and event type context data. For each component, an inventory of suggested fields, their attributes and their definitions will be defined.

The key features of the Event Message Specification are:

Provides a Unique and Global Message Identifier

Each message needs to have a unique global identifier. A tenant of the organization’s overall event architecture is there should be no duplicate messages in the event ecosystem and each message should have a global identifier.

Provides key auditing information

Each message should provide important auditing data - date, time, publisher - when the message was created. Along with the message identifier, this is important for any storage of the message in a data base and for logging, debugging and auditing purposes.

Provides provenance or "chain of processing" for the message, which includes a history of where the logical message has been processed

This answers the questions:

- Where and When did the event happen?
- Who was the original source of the event, along with any publishers who processed and augmented the message? This includes, both systems of record and any publishing platform in the stream.
- Who or What was the caused of the event?

Strives to be independent, stand alone and self-contained

The message should try to contain all the information a consumer might need to process the event. The goal being to avoid complex and time consuming look ups for any consumer requiring additional context to process the message.

Provides simple standard headers and metadata to facilitate routing and filtering of the event as it works it’s way within the event processing network

In any complex environment, there will be a need to route the message to one or more message brokers and consumers. The goal of the standard headers is to provide a set of standard fields to facilitate the creation of libraries and tooling. This will lead to easier creation of message brokers leading to more efficient processing within the networking infrastructure.

Supports schema version control and message validation

Event are not stagnant. They will continually evolve, which means version control should be a first class design element within the message. Events, in general, are immutable, so changing them is basically impossible. This makes version identification extremely important in publishing and consumer processing. The use of *dataSchemas* data elements for event category headers and event type context data allow for the validation of a message. This allows for the easy storage of the specifications in a dictionary or directory in order to validate a message.

Provides the ability to store and find the key of the business object or subject in a system of record associated with the event

In general, every event should be associated with a key independent domain business entity. (For example in HR/Payroll, this would be a *person*). The message should store the primary key, name and type of the business object. This would facilitate the retrieval of the domain object if needed. In addition to the primary business object, there can also be related business objects that were part of the overall context when the event was published. The specifications allows for this information to be part of the event. It too, can be used to retrieve the data about this business entity. If helpful, it can provided correlation for cross event processing.

Provides the ability to submit test or synthetic event for testing

Sometimes during testing, it is helpful not to perform the action the consumer wants to take based on the event it receives. This field allows the consumer to identify this and not take any additional action, like calling a update API or updating a data base.

Contains metadata about the event which helps in stream processing

This information helps to route the request to the proper stream processing nodes within the topology and to facilitate proper processing of the event by a component within the stream topology.

Event Categories

Every domain or industry has their own unique set of events. However, all domains share a set of cross-cutting information that is common across all types of events. These cross cutting concerns share a well-known and consistent pattern. This commonality allows the creation of *Event Categories*. Event Categories provide consistent information in the form of an Event Category Header, that is common to that category of event. Within each category for a given domain, there are specific *Event Types* with their own data schema.

Some examples of *Event Categories* with their own specific event header data are:

Business Consumer/Application User

- Business Process (a.k.a Workflow) State Change Event
- Business Object Data Change Event
- User Experience Action Event
- Generic Goal Event

Runtime

- Platform Processing Event

In the case, where none of the standard event type apply, there is a concept of a *General Category Type*. In this case, the Event Category Header will be absent and the URI would end with */generalEvent*.

Event Design Guidelines

Standard information

When designing an event, one needs to consider and document the following standard metadata information:

- A well known event category, such as a business process state change or data change category.
- The name of the event type.
- The definition of the event target audience.
- An owning application, and by implication, an owning team.
- A schema defining the event body or data payload.

Anatomy of an Event Message

The event message is a JSON document containing one parent JSON object named **event**. The **event** JSON object contains 3 child JSON objects:

eventHeader	Common Event Header - Common data structure across all event messages, independent of event category, event type and event context data
Event Category Header	Common header for a specific event category. The name is based on event category.

eventData	Event message data or key event contextual data at the time of message creation. Since the context of a given event type tends to be unique, this is a free form structure or JSON object (i.e. schema would vary by event) for each event defined within a given event type.

The two event headers (Common and Event Category) contain the metadata about the event: A standard global message header and a event category standard header. The eventData contains the actual data related to the event at the time of the occurrence. This provides the context for the event. The data schema for the eventData is determined by the event designer.

The analogy here is a package distribution center. The message is like a package with the contents inside the package being the **eventData** component and the label being the combination of the two headers. The package label standard structure allows for the packages to move correctly through the distribution center without having to look at the contents inside the package. Event message distribution can act in the same manner, where general message delivery programs can move and direct messages by only looking at the header of the message. If the header follows a standard, then it makes it easier to create programs via tooling to distribute message through the network.

Why is the Event Category Header a first class component?

Most messages within a given category have a single header that can be consistent to consumers interested in that type of message. The purpose of the *Event Category Header* is the author's hypnosis that domains have another level of standardization for their messages. This is another layer of standardization in addition to the common message attributes. This provides the event designer in their domain another layer of consistency and all the benefits of standardization.

The event categories fall into two styles:

(1) Standardization with the domain business objects and processes themselves and,

(2) Event Types that are cross cutting or common against the domains data types. General data changes, state changes in business process and User Experience logging events fall into this category.

Common Message Header

The **Common Message Header** provides the following key features:

Global Message Metadata

The Global Message Metadata contains key information about the message:

- a unique, global message identifier,

- the category of message,
- the type of message
- creation timestamp,
- original publisher and
- history of consumer processors

Any message defined within the ecosystem MUST contain these fields. This information is common to all events and commands.

Event Type MetaData

The Event Type Metadata contains key information about the event type. The event type is a attempt to create more standardization by observing that events can fall into certain categories. Adding this level only increases the ability to take advantage of standards and has the same impact as the Global Metadata. This includes the type of event and the DataSchema of the type to support automated access to the schema definition of the event type and eventData of the event.

Event Context

The event context data is the key fields and their values at the point when the event was published. It includes a context label or tag, along with the action (past tense) that occurred at the time of the event. This supports any routing of the event to other consumers and is key in analytics processing. In general, events are processed against business domain objects. The event context provides the fields for the retrieval of the main subject business entities and any additional related resources involved at the time of publication.

Audit History/Chain of Custody

To support debugging and auditing, the message contains information around who was the original publisher of the message, a history of processors that have touched the message. In addition, it documents the System of Record for the key subject of the message.

Common Header Details

Every message type - event, command or audit - will have a common standard message header. There will only be one format or schema for the common message header and the object is required.

> The name of the JSON object is **eventHeader**.

It contains fields that describe the message at the highest levels and it identifies the source and type of the message. These fields determine the format and names of the fields that follow in the message object. Since this is JSON, routing or filtering (e.g message brokers) can use only the header to determine routing of message or if the consumer is interested in processing the message. This provides a high degree of standardization, which leads to excellent tooling.

Event Category Header

The Event Category Header is a second level header that contains the common elements for all messages of a given event category.

The name of the JSON object is based on the name of the event category.

Each event category will have its own header name and structure. Examples:

- uxEventHeader - for Ux action events
- bpEventHeader - for business process state change events
- boEventHeader - for business object state change events

The *eventCategoryHeaderSchema* field in the header will indicate which event category header is in the message. There will be a structured format/schema for each event category. For an organization, the goal is a small bounded list of event categories. There can be an unlimited number of event definitions (i.e types) within a category. The goal is to have as much standardization in the headers as possible. The variations are meant for the **eventData** JSON object.

Event Message Context

The Event Message Context contains the actual data about the event, when the event was created. This is the context at the time of creation.

The name of the JSON object is **eventData**.

These are fields that are specific to an event type and form the overall definition of the event. The goal is to make the event as self-describing as possible, trying to avoid additional data retrievals to process the message. Since most applications have a large unbounded set of events, the eventData represents the specific fields for a given event. The above headers are intended to be standard, but the eventData is where the specific fields for that event are stored. Each eventData should have it's own schema that can be placed in a schema repository and retrieved by the *bodyDataSchema* field. The schema can then be used for validation and code generation. The eventBodyDataSchema in the Event Type Header will describe the schema for the fields in the eventData.

For example, there are situations where a consumer might be interested in a change within a business object. In this case, the eventData can contain both a before and after image or a list of changes fields with the old and new values. This information can only be observed at the time of the event.

Event JSON Structure

In order to keep the processing of a message simple and easy to produce and consume, the event message has a very flexible structure and is basically an unstructured document. The goal is to have a schema for the header, each event category header and every event data (i.e eventData) itself. The desire is to have a schema dictionary which has a JSON or AVRO schema as it values and it's keyed by some name. The hierarchy is as follows:

- There is only one header schema (key name: eventHeader)
- To determine the <*eventCategoryHeader*> schema definition name, the eventCategoryHeaderSchema field contains the name of the event category
- To determine the eventData schema definition, *eventBodyDataSchema* field determine the name for the eventData schema

The event structure looks as follows:

{"message" :
"eventHeader" : { … },
"*eventCategoryHeader*" : { … },
"eventData" : { … } }

Samples

{"message" : "eventHeader" : { "eventCategoryHeaderSchema": "com.hilcoTech.messages/uxEvent", "eventName" : "PageABC:clicked", … }, "uxEventHeader" : { … }, "eventData" : { … } }

{"message" : "eventHeader" : { "eventCategoryHeaderSchema": "com.hilcoTech.messages/bpEvent", "eventName" : "ContributionRateChange:Completed" … }, "bpEventHeader" : { … }, "eventData" : { … } }

Common Message Header Field Specification

Ed: Need to align these names with the CloudEvent name. Need to consider shorting some of the names (messageId → id) or using some of their names

Table 1. Schema Fields Table

Field Name	Attributes
eventId	String; Required
eventCategoryHeaderSchema	URI (String); Required
eventBodyDataSchema	URI (String); Required
version	String; Required
topic	String ; Optional
eventName	String ; Optional
contextTag	String; Required
action	String; Required
creationTimestamp	Timestamp; Required
businessDomain	String; Optional
correlationId	String; Optional
correlationIdType	String; Optional
subjectIdentifier	String; Required
publisherId	String; Required
publisherApplicationName	String; Required
publisherApplicationInstanceId	String
publishingPlatformsHistory	Object; Array; Optional
- publisherId	String; Required
- publisherApplicationName	String; Required
- publisherApplicationInstanceId	String
- messageId	String; Required; Required
- messageTopic	String; Required
- eventName	String; Required
- messageTimestamp	Timestamp; Required
- sequenceNumber	String
subjectSystemOfRecord	Object; Optional
- systemOfRecordSystemId	String; Required
- systemOfRecordApplicationName	String; Required

Field Name	Attributes
- systemOfRecordApplicationInstanceId	String
- systemOfRecordDatabaseSchema	String
- platformInternalId	String; Required
- platformExternalId	String
correlatedResources	Object; Array; Optional
- correlatedResourceType	String
- correlatedResourceId	String
- correlatedResourceState	String
- correlatedResourceDescription	String
isSyntheticEvent	String

Schema Field Definitions

eventId	Globally Unique Identifier of message. The eventId is expected to be unique from a global perspective, so it is recommended to use some form of a GUID or UUID for this value. It is not recommended that this value have any additional sematic value or meaning beyond uniqueness.

eventCategoryHeaderSchema	eventCategoryHeaderSchema is used to distinguish between the different categories of events, source (internal vs external), and schema versions to avoid collision and help in processing the messages. They also identify the type of Event Category Header contained in the full message. The dataSchema can be used as an external endpoint to provide the schema and other machine-readable information for the event category and the latest major version. Used to provide message definition and validation. Example Values: • com.hilcoTech.messages/events/generalEvent • com.hilcoTech.messages/events/uxEvent • com.hilcoTech.messages/events/businessProcessEvent • com.hilcoTech.messages/events/dataChangeEvent • com.hilcoTech.messages/events/goalEvent • com.hilcoTech.messages/events/platformProcessingEvent
eventBodyDataSchema	Describes the schema for a specific type of event within the category It describes the structure/definition and version of the **eventData** field in the message. This type of information can be placed in a repository and used in the validation of a message. The eventData structure and metadata details are understood based on this name. This field is optional and only be set if there is a structure or schema for the eventData. If there is no eventData, then this field should not be sent.
version	Conveys the version number (major.minor) of the message, and describes the structure of the overall message at hand. Recommendation is to use semantic versions based on breaking changes. Valid values managed by governance • Example: 1.1

topic	Logical name to describe the type of event. Note: this is not the physical topic name (i.e kafka topic) of the messaging system. Sample Valid Values: • BusinessProcess • DomainDataChange • UserExperience • Goal • PlatformProcess
eventName	Provides a standard name of the actual event that occurred in the publishing system. It can be treated as a label/code and used for filtering, routing, general analytics and simple processing of events in the ecosystem. It should be a combination of the business object or process name and action taken on that entity. There are specific naming conventions used to determine the value of the field. It is a field that will require governance approval.
contextTag	Machine readable generic label for the event type. The purpose of the contextTag is to provide a label that encodes some additional context for the event. It is highly structured, follows a specific format and provides valid values to allow programs and applications, like analytics, to easily consume the values. See event category for more details on the values. To reduce the complexity in trying to capture all the levels and details of components that produced the event, the recommendation is to encode all contextual or hierarchical information into a single label or tag. This tag along with the **action** field should reduce the complexity of the event structure and make it easier for the consuming tools to do their work without having to get into the details of the eventData structure. To make it more human readable, there will be an encoding standard in place to make it more human-readable and make it easier to parse the tag if necessary.

action	Represents the actual logical action or happening based on the event type. See event category for more details on the valid values. For events,the action should be described in the past tense and the name should be initial caps. For commands, the action should be present tense with initial cap. The organization should have a bounded set of actions and try to minimize the number.
creationTimestamp	Describes the date and time at which the actual event was generated by the publisher. To be provided by the producer component and should not be derived by message publishing framework(s) or component(s). The timestamp must be in the RFC 3339/ISO 8601 date format standard.
businessDomain	Describes the business domain under which the event/command was generated. Sample Valid Values in HR/Benefits: • Person • Worker • PersonWorker • Health • DefinedContribution • DefinedBenefit • Operations • N/A (for domains that do not match up to an organization service domains.
correlationId	Provide a globally unique identifier (UUID) to tie multiple events to the occurrence. Typically generated within the publishing application. This is used to correlate multiple messages across a logical process. The messageId is unique for the individual message, but the correlationId can be repeated across multiple messages

correlationIdType	Describes the type of correlation identifier. Suggested Values: • SessionId - for participant Ux actions and sessions • BatchId - for batch processing jobs. This is the actual instance id of a job type. • PublisherCorrelationId - for publisher specific correction type (Typically used if the above two does not apply)
subjectIdentifier	Describes the global identity of the business subject being acted upon. The 'subject' is typically a key business domain object. In the HR/Benefits domain, an example would be the person.
publisherId	Identifies the name or id of the publishing company who created the message.
publisherApplicationName	Describes the name of the publisher application platform or service.
publisherApplicationInstanceId	Describes the specific instance of the publisher application or service.

publishingPlatformsHistory	This is the historic details and providence of the message: *the audit trail for the message*. It is an array, describing the internal platforms that have been processing a given logical message from the edge platforms to any internal consumer applications. If the consumed message is being augmented (i.e new information is being added) is is important that the consumer/publisher or program add its own auditing information to the history. It has similar fields to the overall message (see above).

publisherId

Identifies the publishing company entity of the message.

publisherApplicationName

Describes the name of the publisher application platform or service

publisherApplicationInstanceId

Describes the specific instance of the publisher application or service.

messageId

Describes the messageId for the given prior message instance. See above for field details

messageTopic

Describes the messageTopic for the given prior message instance. See above for field details

eventName

Describes the eventName for the given prior message instance. See above for field details

messageTimestamp

Describes the messageTimestamp for the given prior message instance. See above for field details

sequenceNumber

The sequence should be from earliest to latest in

subjectSystemOfRecord

System of Record containing details related to finding the related subject or domain business object.

systemOfRecordSystemId

Identifies the system of record company entity of the message. Sometimes referred to as the partner ID.

systemOfRecordApplicationName

Describes the name of the publisher application platform or service.

systemOfRecordApplicationInstanceId

Describes the specific instance of the system of record containing the person

systemOfRecordDatabaseSchema

Describes the database schema instance of the system of record containing the business object

platformInternalId

Describes the internal identity of the business object within the platform. Only provided if the publishing platform is a source system of record and not a pure publisher application

platformExternalId

Describes the external identity of the business object within the platform. Only provided if the publishing platform is a source system of record and not a pure publisher application

correlatedResources

Describes a list of the related resources also being being accessed during the processing creating the event. These are key *bounded contexts* associated with the primary business entity during processing.

correlatedResourceType

Describes the type of the related resource.

correlatedResourceIdentifier

Identifies the primary key of related resource. This can be the external or internal unique identifier of the resource.

correlatedResourceState

Identifies the state or status of related resource at the time the event occurred.

correlatedResourceDescription

Description of related resource at the time the event occurred.

isSyntheticEvent

Is this a synthetic or fake event? If true, assumes this is an event that should be processed under special circumstance, meaning don’t change state or issue commands. Used for testing/monitoring in production by sending in fake events

Potential Extensions

dataContentType

This will be helpful if the eventData is not JSON. The current best practice is that all eventData payloads, should be JSON. The values would follow HTTP mime types

Chapter 4 - Domain Event Examples - Consumer Business Events

Each domain or industry has their own events with a event category. To help get a better understanding of the general specification, this chapter provides some real life examples. These examples are taken from the author’s experience in developing HR/Benefits Administration system. These examples should also apply to any consumer based application. The following real life consumer related events are discussed:

- Business Process State Changes
- Business Domain Object Data changes
- Consumer User Experience (Ux) Application Interactions
- Consumer General Activities

Add comment about a generic or general or undefined events. Undefined Event: A free form category suitable for events that are entirely custom to the producer.

All of these event would use the Common Message HeaderHeader described in the prior chapter.

Business Process State Change Event

The purpose of this event type is to capture events related to the processing of a business process. These events are sometimes known as workflows or business transactions and are used to manage the changes to business domain objects. Typically, upon the completion of a business process, the process will update a business object. This might also create a Business Object Data Change event.

Events can be generated from two types of business processes.

1) A *long running state machine based* process that has many wait states,with time gaps in between the actions of the business process.

2) A *straight through* process or single task process, where there are no waits in the process and the action is completed in a single unit of work. Typically, it is all or nothing from a commit standpoint.

Business Process examples:

- Contribution Rate Change
- Annual Benefits Enrollment

- Defined Benefit Retirement

State changes includes:

- Started
- Completed
- Validated
- In Error

Business Process State Change Event Type Header Specifications

❗ **JSON Name for *Event Type Header*: bpEventHeader**

Table 2. Schema Fields Table

Field Name	Attributes
businessProcessReferenceId	String; Required
businessProcessId	String; Required
businessProcessDescription	String; Required
businessProcessStatus	String; Required
businessProcessEffectiveDate	Date; Required
businessProcessChangeTimestamp	Timestamp; Required

Schema Field Definitions

businessProcessReferenceId	Describes the primary key or Business Process Reference Identifier of the business process instance as described in platform(s).
businessProcessId	Describes the internal identifier of the business process definition. It is a template that is used to create the actual specific instances of the business process.
businessProcessDescription	Describes the long more formal description of the business process.

businessProcessStatus	Business Process status. Sample Values: • Created • Validated • Invalid • Completed • Canceled
businessProcessEffectiveDate	Effective date of the business process
businessProcessChangeTimestamp	Timestamp of when the business process was changed. The timestamp must be in the ISO 8601 date format standard.

EventName Standards

For the eventName, the standard will be the following fields separated by a colon (":") in camel case.

- *tag* which represents the business process name
- *action* which represents the business process action

Tag Definition

The tag represents the business process name.

Format:

- Free format single alpha or numeric value
- No formal specification is defined

Action Definition

The action represents the types of actions that result from the change in state of the business process during the processing of the consumer's business process.

Action Component Sample Values:

- Started
- Updated
- Completed
- Canceled

Body Definition Considerations

The eventData section is named **eventData**. The **eventData** can be any valid JSON schema. It should contain key information about what action or event triggered the change in state of the process. In some case, a Command with be the triggering event at create this change.

Business Object Data Change Event

The purpose of this event type is to capture the changes to key domain business objects. The event can have both the before and after image or a list of data elements changes, along with the new and old values.

Sample Business Objects include:

- Person
- Employee
- Person 401k Benefits
- Person Medical Benefits
- Person Document

Data actions include:

- Creation
- Updated
- Deletion
- Master Data Management Document Merge/Split

Business Objects Data Change Event Type Header Specifications

JSON Name for *Event Type Header*: boEventHeader

Table 3. Schema Fields Table

Field Name	Attributes
businessObjectResourceType	String; Required
businessObjectIdentifier	String; Required
additionalBusinessObjectResource	Array; Optional
- additionalBusinessObjectResourceType	String; Optional
- additionalBusinessObjectResourceId	Date; Optional

Field Name	Attributes
dataChangeTimestamp	Timestamp; Required

Schema Field Definitions

businessObjectResourceType	Describes the primary domain data object type that was changed. Sample Values: • person • personDefinedContribution • personHealthManagement • personDefinedBenefit • personDefinedBenefitCalculation • personDocument
businessObjectIdentifier	Provides the primary key of the business object that was changed. This information might be a duplicate of what is in the Common Message Header.
additionalBusinessObjectResource	Provides any additional resource type and key to help further identify the component that changed. This is similar to the path (../resource/{id}) in a REST URL **additionalBusinessObjectResourceType** Additional resource type **additionalBusinessObjectResourceId** Additional resource identifier or primary key
dataChangeTimestamp	Timestamp of the data change in the source platform. The timestamp must be in the RFC 3339/ISO 8601 date format standard. See Appendix for details.

EventName Standards

For the eventName, the standard will be the following fields separated by a colon (":") in camel case.

- *tag* which represents the business object name and
- *action* which represents the CRUD operation taken against the business object

Tag Definition

The tag represents the business object name.

Format:

- Free format single alpha numeric value
- No formal specification is defined

Action Definition

The action defines the type of data maintenance (CRUD) action taken on the business object.

- Action Component Sample Values

  dataAction

  Describes the data change or CRUD action performed on business object.- Create, Update, Delete. Also includes an primary key changes and Master Data Management (MDM) document merging.

- Create
- Update
- Delete
- MdmDocumentMerge
- MdmDocumentSplit

Body Definition Considerations

- The eventData section is named **eventData**
  - **eventData** can be any valid JSON schema
- Contains one predefined element **extension**
  - Extension is a private area that can contain its own schema
  - The field is an map/array with:
    - Namespace as a key and,
    - Any valid JSON schema as its value

Data Fields Best Practices by Data Action

Update The recommendation for data fields to report is to provide only the fields that changed providing both old and new values.
Best practice recommendations:

- Personal Identification Information (PII)
    - Fields: Bank/Credit Account Numbers,
    - Provide old/new unchanged from CustomerMaster; no masking required
- Arrays
    - Provide Lowest Level Detail field, include all cascading keys
    - Example: Contact → streetAddress → { AddrID → OldZipcode, newZipcode }
    - Include all the fields at the same level as the changed field in entire array data object
    - For fields in a high level/hierarchy, include all keys and simple primitive types (strings, numbers,etc) at the same hierarchy
- Do not include objects or arrays in the higher levels Do not include non-changing arrays at the same level

Create Provide the entire New entity. The alternate is too only provide foreign keys, which can be used to retrieve data from an API or data base.

Delete Only provide a delete event if the entire document is being deleted, not if one of the source systems deleted a person. In the eventData, provide the primary document key (UniversalId or Mongo _id) and any IdMapping table If the object/person is being delete in a given platform, but the person still exists in another platform, treat as an Update. Only delete when no more IdMappings exist in the document

Master Data Management Platforms/CustomerMaster

Editor: Should this be removed

Merge

- Treat as an MDM Merge Update event with two sections of data, one for survivor and one for deleted
- Both sections
- Survivor _id & Deleted _id
- Id Mapping for both survivor and deleted
- Survivor document section contains the update record for the survivor document (see Update section)
- Deleted document section
- Reason for merge
- The Platform that caused the change to occur
    - System Instance
    - Merge Field Change (old, new)

Split

No new events, just two new event being generated Web service call to deletePersonId service, which cleans up IdMapping and domain sections. Generates a Normal Update event. Web Service call refreshPersonForInternalId service, which causes a refresh through .

Ingest

Generates a Normal Update event

User Experience Action Event

Ed: Ad some content around: In Ux Events, state that UX logs are really events and should be treated as them. In particular trying ot log business action. It also works for debugging and tracing log entries.

The Ux Action events are intended to capture the actual keyboard/mouse events performed by the user - displaying pages, clicking button or links, entering text. These are events related to the behavioral actions taken by the user in the online channels. Channel include web, mobile, IVA/chat and other future user devices like Voice Assistants. These events are not the result of any business process or data change events.

They are used for:

- Behavior actions for data reporting and analytics
- Provide notifications to non-domain processes (document management, campaigns) to drive their underlying processes

Actions may include, but not limited to:

- Button clicks
- Link or action selections
- Page or screen displays
- Hover
- IVA or chat intents

The intention is to capture the actual true or syntactic actions along with a navigation/breadcrumb label. The goal is not to add any business semantics to the event. There should be enough context in the label for another offline process (e.g. analytics process) to create another event with the business semantics of the users action.

In most systems, these are considered logging or debugging actions. By adding a session context as a correlation value and adding additional related business object information to the event, it makes it easier for analytics processes to tie a users session together to identify key trends.

User Experience Action Event Type Header Specifications

JSON Name for *Event Type Header*: uxEventHeader

Header Attributes

Table 4. Schema Fields Table

Field Name	Attributes
channel	String; Required
userDevice	String; Required
deviceTimestamp	Array; Optional
sessionId	String; Optional
sessionStartTimestamp	Timestamp; Optional
applicationVersion	String; Optional

Schema Field Definitions

channel	Describes the channel (or UI application) where the event generated.
userDevice	Identifies the device used by end-user.
deviceTimestamp	Represents the timestamp on the device (May be different from the publisher timestamp). The timestamp must be in the RFC 3339/ISO 8601 date format standard. See Appendix for details.
sessionId	Represents the unique session of end user on our channels.
sessionStartTimestamp	Session creation or start time. The timestamp must be in the RFC 3339/ISO 8601 date format standard. See Appendix for details.
applicationName	User Experience application name
applicationVersion	Version of the application

EventName Standards

For the eventName, the standard will be the following fields separated by a colon (":") in camel case.

- UxControlName
- UserAction

Tag Definition

In the Ux channels, there are an unbounded set of device actions a user can take: pressing buttons, displaying pages, starting process flows. In addition, they are an unbounded set of specific controls (buttons, etc) throughout the interface. For reporting and other activities, there is a need to capture that a specific control has been acted upon: pressing a specific button within a specific group of controls within a page within a business process flow.

To reduce the complexity in trying to capture all the levels and types of components, the recommendation is to are encode all hierarchical information (i.e. breadcrumbs) into a single label or tag using a structured format. This tag along with the user action on this tag should reduce the complexity of the event structure and make it easier for the consuming tools to do their work. This will also make it easier for the UX developer since they will not be dealing with the business aspect of the action. They only need to produce an event (a.k.a. log) with a label and the actual mouse/keyboard action. The interpretation of the label/action will be a outside downstream activity.

To make it more human readable, there will be an encoding standard to make it more human readable and make it easier to parse the tag if necessary. The tag values need to take into account all types of user interfaces and devices. There is a need to support new and emerging interfaces beyond web and mobile channels. The following sections discuss the naming approach.

Tag Component Valid Values

Web Channel	• Flow - A user’s perceived outcome process or unit of work; Denotes flow of interaction (pages) or conversation between user and system ◦ Page ◦ Widget or Multiple Control Component • Elemental Ux Control ◦ Button, includes clickable icons - Clickable ◦ Link - Clickable ◦ CheckBox - Selectable ◦ Text - Display, Hover, Table Element ◦ TextBox - Keyboard Actions → Tabbing ,Enter pressed ◦ Bounded Lists → Radio Buttons or checkboxes or DropDown Lists or Dials - Selectable
Mobile	TBD
Smart Assistant/AlexaIVA/Chat	TBD
Other on Non-Channel	Treatment or Theme Example xxxA/xxxB

Format

- Ordered sets of tuples separated by underscore '_'
- The tuple is the following fields separated by dash '-'
    - LogicalName determined by Ux Designer and Data Analyst
    - UxControl Valid Value in all caps
- The order is from highest level (aFlow) to specific UX Control, (Button)

Format: <Flow_Name>-FLOW_<Page_Name>-PAGE_<ButtonLabel>-BUTTON
Example: Retirement-FLOW_LandingPage-PAGE_OK-BUTTON (which means the user accepted their retirement elections and they will be processed)

Action Definition

The action defines the actual keyboard/mouse actions taken by the user when interacting with the channel/device.
Sample Values for userAction:

- Displayed
- Clicked
- Entered

Body Definition Considerations

- The eventData section is named **eventData**
    - **eventData** can be any valid JSON schema
- Contains one predefined element **extension**
    - Extension is a private area that can contain its own schema
    - The field is an map/array with:
        - Namespace as a key and,
        - Any valid JSON schema as its value
- This can be any significant data or data of interest for reporting at the time of the UX Event

Consumer Goal Event

These are events related to the action taken by the consumer in the context of reaching a personal goal.

A goal is non-transactional outcome the consumer is trying to attain.
For example, the person wants to lose 20lbs as a health goal

Actions may include: * Started * Completed

Consumer Goal Event Type Header Specification

The Personal goal only requires the main header
JSON Name for *Event Type Header*: pgEventHeader

Tag Definition
The tag represents the name of the personal goal in a machine readable format.

Format: * Free format single alpha numeric value * No formal specification is defined

Action Definition
The action defines the type of task actions taken against a personal goal.

Action Component Sample Values : * Started * Completed

Body Definition Considerations

- The eventData section is named **eventData**
  - eventData can be any valid JSON schema
- Contains one predefined element **extension**
  - Extension is a private area that can contain its own schema
  - The field is an map/array with:
  - Namespace as a key and,
  - Any valid JSON schema as its value
  - This can be any significant data or data of interest for reporting at the time of the UX Event

Chapter 5 - System Event Example - Runtime Operations Events

Runtime Operation Events are events that are occurring during the running of the application on a specific platform or system resource (server, data base, etc). They are focused mostly on system resource issues. Debug events would also fall into this category

Platform Processing Event

These are events related to the action of completing a discreet process or unit of work and providing the resource computation of that unit of work.

These events reflect the fact that:

- the application process occurred, which is used for counting instances of the process
- the time stamps of when it occurred, which is used for elapse time and windows of time,
- the timestamp of the additional resource usage, which is used for more detailed resource consumption analytics and
- any additional status and metadata related to the process of the unit of work. The event can be used for both operational and application events where counting, and resource utilization reporting is of interest to the business.

Platform processing units of work being metered include:

- Docker Containers → Service API calls
- Enterprise CI/CD Build pipeline → API Service builds

System Resource Manager Event

Under consideration

These are events related to the actions of managing a system resource manager. This includes the overall operations of the resource manager and the system administrator actions to administer the resource manager.

A resource manager has the following characteristics: (Examples of a resource manager: a server(Hardware with OS), JVM, Web Server, Docker container, data base manager and queue manager/message broker)

- The platform contains important application data
- The resource is shared by multiple applications

- The resource is managed centrally by an system operations staff, who is responsible for the installation, upgrade and overall operations of the resource
- Privileged access authority is required to perform system administration functions

Resource manager operational event actions include:

- Started
- Stopped
- Aborted
- Restarted

System Administrator event actions include:

- Logon
- Password changed
- Commands (?)
- Group Functions (?)

Potential Future Operational Events

- Runtime System Error Events - Runtime Events because of hardware or software issues
- Code Deployment Events - DevOps Events because of program/business logic development. Include both code and configuration
- Client Deployment Events - Events related to the deployment of client assets, in particular provision migrations. Might also include client level processing runtime errors

Operations Platform Process Event

Platform Process Event Header

JSON Name: oppEventHeader

Table 5. Schema Fields Table

Field Name	Attributes
platformProcessStartTimestamp	Timestamp; Required
platformProcessEndTimestamp	Timestamp; Required

Field Name	Attributes
platformProcessElapsedTime	Float; Optional
platformProcessEffectiveDate	Date; Optional
platformProcessStatus	String; Optional

Schema Field Definitions

platformProcessStartTimestamp	Timestamp of when the resource consumption started. The timestamp must be in the RFC 3339/ISO 8601 date format standard. See Appendix for details.
platformProcessEndTimestamp	Timestamp of when the resource consumption ended. The timestamp must be in the RFC 3339/ISO 8601 date format standard. See Appendix for details.
platformProcessElapsedTime	Elapse time in milliseconds
platformProcessEffectiveDate	Effective Date of process
platformProcessStatus	Process status. Sample Values: • Aborted • Completed • Canceled

Potential Platform Process Header Fields

- API/Program call Request and Response size
- Memory size at event publication
  - Heap sizes , etc
- CPU utilization for process
- Thread count at event publication

EventName Standards

For the eventName, the standard will be the following fields separated by a colon (':') in camel case.

- PlatformProcess
- Action

Tag Definition

- Format
    - Ordered sets of tuples separated by underscore '_'

Action Definition

The action defines the type of action or state changes of the process.

Action Component Valid Values

Process Action Sample Values:

- Started
- Completed
- StateChanged

Body Definition Considerations

- The eventData section is named 'eventData'
    - 'eventData' can be any valid JSON schema
- Contains one predefined element 'extension'
    - Extension is a private area that can contain its own schema
    - The field is an map/array with:
        - Namespace as a key and,
        - Any valid JSON schema as its value This can be any significant data or data of interest for reporting at the time of the process state change.

Chapter 6 Command Message Specifications

Overview

❗ **The Command message is very similar to the Event message. The difference is that the event is published after the fact, whereas a command is a present request to take action.**

The Command message shares many of the same key features as events (See Event section for more details):

- Provides a Unique and Global Message Identifier
- Strives to be as independent, stand alone and self-contained as much as possible.
- Provides simple headers and metadata to facilitate routing and filtering within the event processing network
- Supports schema version control and message validation
- Provides the ability to store and find the key of the key business object or subject in a system of record - Need to think more on this
- Provides the ability to submit test or synthetic event for testing
- Contains metadata about the event which helps in stream processing

The additional key features are:

- Contains Requestor metadata which can be used in callback situations

Command Types

More work needs to be done in this area

The command types are:

- Business Process State Change Request
- Business Object Data Change Request
- Communication Composition/Delivery

Command Message Overview

The command message is a JSON document containing one JSON object named “message”. It follow the same general pattern as the event’s Common Message Header, as there are some common fields that are common to all message types. Some fields are added to support commands and other are removed because they relate specifically to events.

Command JSON Structure

Like events, the command message has a very flexible structure and is basically an unstructured document. The goal is to have a schema for the header and every command data (i.e eventData) itself. We would like to have a schema dictionary which has a JSON or AVRO schema as it values and it’s keyed by some name. The hierarchy is as follows:

- There is only one header schema (key name: header)
- To determine the body schema name, the header.commandBodyDataSchema field determine the name for the body schema

T

The internal command structure looks as follows:

{"message" :
"header" : { … },
"*commandTypeHeader*" : { … },
"body" : { … } }

Command Message Header Field Specification

Table 6. Schema Fields Table

Field Name	Attributes
messageId	String; Required
messageType	String; Required
eventHeaderSchema	String; Required
messageVersion	String; Required
messageTopic	String
commandName	String
commandBodyDataSchema	String
contextTag	String; Required

Field Name	Attributes
action/request	String; Required - Should this be request
tagObjectId	String; Optional
messageTimestamp	String; Required
correlationId	String; Required - Relevant to Command?
correlationIdType	String; Required - Relevant to Command?
messageCriticality	String
messageExpiry	String
businessDomain	String; Required
requestorId	String; Required
requestorApplicationName	String; Required
requestorApplicationInstanceId	String
requestingPlatformsHistory	Object; Array; Required - Relevant to Command?
- requestorId	String; Required
- requestorApplicationName	String; Required
- requestorApplicationInstanceId	String
- messageId	String; Required; Required
- messageTopic	String; Required
- commandName	String; Required
- messageTimestamp	String; Required
- sequenceNumber	String
businessObjectSystemOfRecord	Object; Array; Optional - Relevant to Command?
- systemOfRecordSystemId	String; Required
- systemOfRecordApplicationName	String; Required
- systemOfRecordApplicationInstanceId	String
- systemOfRecordDatabaseSchema	String

Field Name	Attributes
- platformInternalId	String; Required
- platformExternalId	String
correlatedResources	Object; Array; Optional - Relevant to Command?
- correlatedResourceType	String
- correlatedResourceIdentifier	String
- correlatedResourceState	String
- correlatedResourceDescription	String
isSyntheticCommand	String

Schema Field Definitions

messageId	Global and Unique (UUID) Identifier of message.
messageType	Describes the type of message. Valid Values: • Command
eventHeaderSchema	eventHeaderSchema is used to distinguish between different types of messages (events vs commands), source (internal vs external), and schema versions to avoid collision and help in processing the messages. The eventHeaderSchema can be used as an external endpoint to provide the schema and other machine-readable information for the command type and the latest major version. Used to provide message definition and validation. Sample Values: • com.hilcoTech.messages/commands/aCommand
messageVersion	Conveys the version number (major.minor) of the message, and describes the structure of the overall message at hand. Valid values managed by governance. • Example: 1.1

messageTopic	String Logical name to describe the type of command. Note: this is not the physical topic name (i.e kafka topic) of the messaging system.
commandName	Provides a standard name of the actual command that happened based on a user’s behavior action on the Ux channel or sensor. It will be treated as a label/code and used for filtering, routing, general analytics and simple processing of commands in the ecosystem. It should be a combination of the business process name and action taken on that process. There are specific naming conventions used to determine the value of the field. It is a field that would require governance approval.
commandBodyDataSchema	Describes the specific schema and version of the body field structure of the command. The body structure and metadata details are understood based on this combination. This field is optional and only be set if there is a structure or schema for the body. If there is not body, then this field should not be sent.
tag	Machine readable generic label for the command type. Its purpose is to provide a label that encodes some additional context for the command. It is highly structured, follows a specific format and provides valid values to allow program and applications, like analytics, to easily consume the values. See command type for more details on the values. To reduce the complexity in trying to capture all the level and types of components, we are going to encode all contextual or hierarchical information into a single label or tag. This tag along with the user action on this tag should reduce the complexity of the command structure and make it easier for the consuming tools to do their work without having to get into the details of the body structure. To make it more human readable, there will be an encoding standard to make it more human readable and make it easier to parse the tag if necessary.

action/request	Represents the action being requested by the consumer on. See command type for more details on the valid values. For commands, the action should be described in the present tense and the name should be initial caps.
tagObjectId	Used to provide a separate identifier for the object of the tag. If the tag represents a general category and there are instances of that category that contain a key /identifier, this field can be used to provide the identifier. The recommended best practice is to put the identifier in the tag itself. This field, along with the generic tag value, provides an alternate to that approach
messageTimestamp	Describes the date and time at which the actual command was generated by requesting systems. To be provided by producer component and should not be derived by message requesting framework(s) or component(s). The timestamp must be in the RFC 3339/ISO 8601 date format standard. See Appendix for details.
messageCriticality	Provides a way for the requestor to indicate a priority for handling of the message. The processor of the command is not required to honor this field. Valid Values: • High • Medium • Low
messageExpiry	Number in seconds Used to determine if the message is still valid to process. This helps in the determination of whether this message should still be processed and is set against the messageTimestamp. If the current time is past the messageTimestamp plus this value, then the message should be ignored.

businessDomain	Describes the business domain under which the event/command was generated. Sample Values: • Person • Worker • PersonWorker • Health • DefinedContribution • DefinedBenefit • Operations • N/A (for domains that do not match up to our organization service domains.

requestorApplicationName

Describes the name of the requesting application platform or service.

requestorApplicationInstanceId

Describes the specific instance of the requestor application or service.

isSyntheticCommand

Is this a synthetic or fake command? If true, assumes this is an command that should be processed under special circumstance, meaning don’t change state or issue commands. Used for testing/monitoring in production by sending in fake commands

Potential Future Command Fields

consumerCallbackInstructions	HEADER field on how to execute the callback. This could be: • An Id of a function or policy to execute • Actual source code that can be interpreted and executed (DSL, Lambda
consumerCallbackInputs	Inputs unique to this callback logic. This would be an array of name value pairs
consumerCallbackScript	Actual scripting code/logic to execute which may update a database or call a rest service, etc…

consumerCallbackCredentials	This could be: • Token based → Short lived token and Expiration Date • Functional UserID/Password → for internal use only • SAML like approach
consumerCallbackErrorInstructions	HEADER field on how to execute the callback if there are errors. This could be: • An Id of a function or policy to execute • Actual source code that can be interpreted and executed (DSL, Lambda
consumerCallbackErrorInputs	Inputs unique to the error callback logic. Array of name value pairs
consumerCallbackErrorScript	Actual scripting code/logic for errors to execute which may update a database or call a rest service, etc…
queryParameters	BODY field - This would be GET parameters command input if the callback is a REST API
requestBody	BODY field - This would be PUT/POST parameters command input if the callback is a REST API

Chapter 7 - Domain Command Example

This chapter ....

Chapter 8 Audit Message Specifications

Overview

Key features of Audit Message Specification are:

Audit Types

The follow sections provide the specification for the types of Audits support by the architecture. (Note: Some Audit types are in the prototype stage)
The Audit types are:

Audit Message Overview

The standard audit message is a JSON document containing one JSON object named **message**. The **message** object contains 3 child JSON objects:

- Common Message Header - Common across all messages, independent of Audit type and audit
- Audit Type Header - Common header for specific Audit types
- Audit Message eventData - Free form eventData for each Audit defined with the Audit type

Internal Audit JSON Structure

To keep it simple and easy to produce and consume, the audit message has a very flexible structure and is basically an unstructured document. The goal is to have a schema for the header, each audit type header and every audit data (i.e eventData) itself. We would like to have a schema dictionary which has a JSON or AVRO schema as it values and it's keyed by some name. The hierarchy is as follows:

- There is only one header schema (key name: header)
- To determine the <auditTypeheader> name, the header.messageNamespace field contains the name of the audit type
- To determine the eventData schema name, the header.auditBodyNamespace field determine the name for the eventData schema

The internal audit structure looks as follows:

{"message" :

> "header" : { … },
> "*auditTypeHeader*" : { … },
> "eventData" : { … } }

Chapter 9 - Domain Audit Example

This chapter ….

Chapter 10 - Message Governance

References

- [cloudEvents] CloudEvents; CNCF Serverless Working Group *https://cloudevents.io*

Appendix A: CloudEvents Data Attributes Comparison versus Common Event Message Header

Common Elements (CloudEvent → Event Specification)

id → messageId
source → eventName and Namespace
specVersion → messageVersion
dataContentType → JSON only
dataSchema → messageNamespace, eventNamespace (eventData)
subject → Subject/Business Object
time → messageTimestamp

In CloudEvents but not Event Specification

<TBD>

In Event Specification but not cloudEvents

<TBD>

Appendix B: Business Process Definition

This book uses the term "business process" frequently and there is even a Event Type to support this entity. There are many term used for this type of concept: Workflow, Activity, Transaction,even Event. This appendix provides the authors definition of a **Business Process**.

A **business process** is series of steps/actions/tasks described it a graph like flow (think flow chart) that accomplishes a specific domain related objective. Examples are the entire enrollment process through carrier notification, the entire dependent verification flow from request to final verification. In each of these cases, there is a clear domain objective the users is attempting to complete. This may be done in a short duration single transaction (i.e., committed unit of work) or a longer running flow over time (days or weeks) with time pauses throughout.

Some of the aspects of a business process are:

- There is some event or trigger that initiates the business process (sometimes is called a workflow)
  - This can be a user action/notification (i.e. Birth of a child) or time based (Starting enrollment on a specific date; opening an enrollment window)
- They contain a series of steps, events that trigger action logic.
  - There is a “business process template”, that defines the starting trigger, steps or the logic involved in the process and the definition of what is consider “finished”. For examples, the steps of an annual enrollment are defined here. Then:
  - The individual instances are created from that template. Tim’s annual enrollment, John’s annual enrollment…
  - They can form a flow graph with sequential steps, if with branches and loops
  - Each step is a discreet piece of logic sticking to the “Single Responsibility” pattern. Examples are updating coverage, sending an email, etc)
  - Multiple steps can be combined into a committable “unit of work” or executed as a single action
  - Each step needs to define:
    - Event that triggers the work if the process is in a paused state
    - The logic to perform one the event is identified
    - What to do if the event is not received (Timeout Policy)
    - What to do if a compensation action, moving backward in the flow, is needed
- A well-defined business process has clear outcome that determines when the business

process is completed. If the definition of the process causes it to stay pending for a long period of time, it’s probably not well defined

- The process does not have to be all automated and there could be some manual processes outside the process that move the process along, For EOI, the process might be waiting for that to come in, but the actual act of process the EOI is outside the process.

Example: New Hire Benefits Enrollment:

When a client enters a new employee into their HR system that employee will be create in the benefits enrollment system in near real time so that the employee can log in and access the new hire enrollment instantly. And once the enrollment is complete the elections will be sent to the carriers in real time as well. At this point, an employee could just go to the drug store across the street and get their prescription filled because Rx provider would have their benefit information available already.

The business process is **New Hire Enrollment**.

The trigger is benefits administration system getting a Request for New Hire Enrollment via an API call from the employer. The outcome is Enrollment completed at the carrier(s). In this case, CVS for Rx. The steps defined in the template would be (timeouts and compensation policies/actions removed).
1. Email step: Send employee an email with link to enrollment – Pause until the employee logs in (trigger)
2. User Experience Step: Employee completes and confirms elections – System updates enrollment data base record
3. Integration Step(s): Carriers/Vendors notified of new elections
4. Email Step: Send employee email stating coverage available and they can enroll as member with carrier

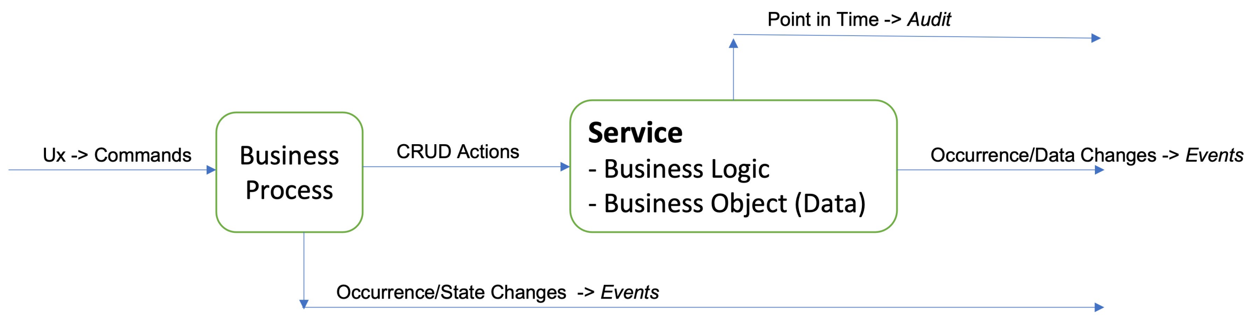
At this point the employee can go to CVS. From a unit of work standpoint, Step 1 is one unit of work, followed by a pause, then Steps 2 is another with a commit point and then Step 3-4 as another unit of work. If something bad happens with the carriers, the business process will stay in the state of Step 2 completed and can be restarted to complete the remaining steps.

In general, there are two types of business processes:

1. **Straight Through**, where everything is done in one unit of work and it is all or nothing is there is a rollback
2. **State Machine Based**, where there are a set of state where the process can pause and then resume. These are used when there is time lags awaiting responses from a user. Event triggers are then used to restart the process

One of the main issues with using HTTP REST based APIs is that the API interaction is not guaranteed. The endpoint might not be available, or it might get lost in the network. This is particularly an issue if bulk or batch work is done via API. An HTTP REST API should complete in less than a second to avoid performance problems. This causes operational issues for both sides:consumer and providers. On the consumer side, this means they must remember where the flow is at and be able to restart at that point when the issue has been resolved. On the provider side, the API must be idempotent

Appendix C: Business Object Anatomy



Big Three

1. Business Object – Service = DATA & LOGIC
2. Business Process – Ux Experience
3. Messages (Events, Commands, Audit)

Appendix D: JSON Event Schema

```
{
  "$schema": "http://json-schema.org/schema",
  "$id": "https://timhilco.com/event.schema.json",
  "title": "Event",
  "description": "An event",
  "type": "object",
  "properties": {
    "Event": {
      "description": "An message",
      "type": "object",
      "properties": {
        "eventHeader": {
          "$ref": "#/$defs/eventHeader"
        },
        "eventCategoryHeader": {
          "oneOf": [
            {
              "$ref": "#/$defs/bpEventHeader"
            },
            {
              "$ref": "#/$defs/boEventHeader"
            }
          ]
        },
        "eventData": {
          "$ref": "#/$defs/eventData"
        }
      }
    }
  },
  "required": [
    "eventHeader",
    "eventCategoryHeader",
    "eventData"
  ],
  "$defs": {
    "eventHeader": {
      "description": "The message header",
      "type": "object",
      "properties": {
        "eventId": {
          "description": "The event global Id",
          "type": "string",
          "example": "c8ae150b-7363-487b-9c08-edafcc4966d2"
        },
        "version": {
          "description": "The event version",
          "type": "string",
          "example": "1.0.0"
        },
        "topic": {
          "description": "The event version",
          "type": "string"
        },
        "eventCategoryHeaderDataSchema": {
          "description": "The ...",
          "type": "string"
        },
        "eventBodyDataSchema": {
          "description": "The ...",
          "type": "string"
        },
        "eventName": {
          "description": "The ...",
          "type": "string"
        },
        "contextTag": {
          "description": "The ...",
          "type": "string"
        },
        "action": {
          "description": "The ...",
          "type": "string"
        },
        "creationTimestamp": {
          "description": "The ...",
          "type": "string",
          "format": "date-time"
        },
        "businessDomain": {
          "description": "The ...",
          "type": "string"
        },
        "correlationId": {
          "description": "The ...",
          "type": "string"
        },
        "correlationTIdType": {
          "description": "The ...",
          "type": "string"
        },
        "subjectIdentifier": {
          "description": "The ...",
          "type": "string"
        },
        "publisherId": {
          "description": "The ...",
          "type": "string"
        },
        "publisherApplicationName": {
          "description": "The ...",
          "type": "string"
        },
        "publisherApplicationInstanceId": {
          "description": "The ...",
          "type": "string"
        },
        "publishingPlatformsHistory": {
          "description": "The ...",
          "type": "array",
          "items": {
            "$ref": "#/$defs/publishingPlatformItem"
          }
        },
        "systemOfRecord": {
          "description": "The ...",
          "type": "object",
          "items": {
            "$ref": "#/$defs/systemOfRecord"
          }
        },
        "correlatedResources": {
          "description": "The ...",
          "type": "array",
          "items": {
            "$ref": "#/$defs/correlatedResourcesItem"
          }
        }
      }
    },
    "publishingPlatformItem": {
      "description": "The Event Type Header for Business Processes",
      "type": "object",
      "properties": {
        "publisherId": {
          "description": "The ...",
          "type": "string"
        },
        "publisherApplicationName": {
          "description": "The ...",
          "type": "string"
        },
        "publisherApplicationInstanceId": {
          "description": "The ...",
          "type": "string"
        },
        "eventId": {
          "description": "The event global Id",
          "type": "string",
          "example": "aUUID"
        },
        "topic": {
          "description": "The event version",
          "type": "string"
        },
        "eventName": {
          "description": "The ...",
          "type": "string"
        },
        "creationTimestamp": {
          "description": "The ...",
          "type": "string",
          "format": "date-time"
        }
      }
    },
    "systemOfRecord": {
      "description": "The system of Record where the event was originated",
      "type": "object",
      "properties": {
        "systemOfRecordId": {
          "description": "The ...",
          "type": "string"
        },
        "systemOfRecordApplicationName": {
          "description": "The ...",
          "type": "string"
        },
        "systemOfRecordApplicationInstance": {
          "description": "The ...",
          "type": "string"
        },
        "systemOfRecordIdDatabaseSchema": {
          "description": "The ...",
          "type": "string"
        },
        "platformInternalId": {
          "description": "The ...",
          "type": "string"
        },
        "platformExternalId": {
          "description": "The ...",
          "type": "string"
        }
      }
    },
    "correlatedResourcesItem": {
      "description": "The Event Type Header for Business Processes",
      "type": "object",
      "properties": {
        "correlatedResourcesType": {
          "description": "The ...",
          "type": "string"
        },
        "correlatedResourceId": {
          "description": "The ...",
          "type": "string"
        },
        "correlatedResourceState": {
          "description": "The ...",
          "type": "string"
        },
        "correlatedResourceDescription": {
          "description": "The ...",
          "type": "string"
        }
      }
    },
    "bpEventHeader": {
      "description": "The Event Type Header for Business Processes",
      "type": "object",
      "properties": {
        "businessProcessReferenceId": {
          "description": "The ...",
          "type": "string"
        },
        "businessProcessId": {
          "description": "The ...",
          "type": "string"
        },
        "businessProcessDescription": {
          "description": "The ...",
          "type": "string"
        },
        "businessProcessStatus": {
          "description": "The ...",
          "type": "string"
        },
        "businessProcessEffectiveDate": {
          "description": "The ...",
          "type": "string",
          "format": "date"
        },
        "businessProcessChangeTimestamp": {
          "description": "The ...",
          "type": "string",
          "format": "date-time"
        }
      }
    },
    "boEventHeader": {
      "description": "The Event Type Header for BusinessObjects",
      "type": "object",
      "properties": {
        "businessObjectResourceType": {
          "description": "The message Id",
          "type": "string"
        },
        "businessObjectIdentifier": {
          "description": "The message Id",
          "type": "string"
        },
        "additionalBusinessObjectResource": {
          "description": "The message Id",
          "type": "array",
          "items": {
            "properties": {
              "additionalBusinessObjectResourceType": {
                "description": "The message Id",
                "type": "string"
              },
              "additionalBusinessObjectResourceId": {
                "description": "The message Id",
                "type": "string"
              }
            }
          }
        },
        "dataChangeTimestamp":  {
          "description": "The message Id",
          "type": "string"
        }
      }
    },
    "eventData": {
      "description": "An event data context as JSON string",
      "type": "object"
    }
  }
}
```

Appendix E: JSON Data Change Event Examples

```
{
  "eventHeader": {
    "eventId": "c8ae150b-7363-487b-9c08-edafcc4966d2",
    "version": "1.0.0",
    "topic": "PersonEmailAddressChanged",
    "eventCategoryHeaderDataSchema":
"com.hilcoTech.messages/events/dataChangeEvent",
    "eventBodyDataSchema": "com.hilcoTech.messages/events/emailAddressChange",
    "eventName": "PersonEmailAddress:Changed",
    "contextTag": "Person_0010010001_EmailAddress",
    "action": "Changed",
    "creationTimestamp": "2022-09-01T07:20:50.52Z",
    "businessDomain": "HR-Payroll",
    "correlationId": "c8ae150b-7363-487b-9c08-edafcc4966d2",
    "correlationTIdType": "Session",
    "subjectIdentifier": "001010001",
    "publisherId": "HilcoTech",
    "publisherApplicationName": "HilcoTechHR",
    "publisherApplicationInstanceId": "System1",
    "publishingPlatformsHistory": [
      {
        "publisherId": "HilcoTech",
        "publisherApplicationName": "HilcoTechPayroll",
        "publisherApplicationInstanceId": "System1",
        "eventId": "c8ae150b-7363-487b-9c08-edafcc4966d23",
        "topic": "PersonEmailAddressChanged",
        "eventName": "PersonEmailAddress:Changed",
        "creationTimestamp": "2022-09-01T07:20:49.52Z"
      }
    ],
    "systemOfRecord": {
      "systemOfRecordId": "HilcoTech",
      "systemOfRecordApplicationName": "HilcoTechPayroll",
      "systemOfRecordApplicationInstance": "EastRegion",
      "systemOfRecordIdDatabaseSchema": "PDB01",
      "platformInternalId": "001010001",
      "platformExternalId": "001-01-0001"
    },
    "correlatedResources": [
      {
          "correlatedResourceType": "BusinessProcess",
          "correlatedResourceId": "c8ae150b-7363-487b-9c08-edafcc4966d23",
          "correlatedResourceState": "Completed",
          "correlatedResourceDescription": "Email Change Business Process"
        }
      ]
    },
    "boEventHeader": {
      "businessObjectResourceType": "Person",
      "businessObjectIdentifier": "001010001",
      "additionalBusinessObjectResource": [
        {
          "additionalBusinessObjectResourceType": "",
          "additionalBusinessObjectResourceId": ""
        }
      ],
      "dataChangeTimestamp": "2022-09-01T07:20:50.52Z"
    },
    "eventData": {
      "priorEmailAddress": "john.sample@gmail.com",
      "currentEmailAddress": "john.sample2@gmail.com"
    }
  }
```