cloudevents / spec

Code Issues 42 Pull requests 2 Actions Projects Wiki Security

spec / cloudevents / primer.md

alexec Avro Compact Format (#1213) last month

1036 lines (859 loc) · 49.8 KB

Preview Code Blame Raw

CloudEvents Primer - Version 1.0.3-wip

Abstract

This non-normative document provides an overview of the CloudEvents specification. It is meant to complement the CloudEvent specification to provide additional background and insight into the history and design decisions made during the development of the specification. This allows the specification itself to focus on the normative technical details.

Table of Contents

History

The CNCF Serverless Working group was originally created by the CNCF's Technical Oversight Committee to investigate Serverless Technology and to recommend some possible next steps for some CNCF related activities in this space. One of the recommendations was to investigate the creation of a common event format to aid in the portability of functions between Cloud providers and the interoperability of processing of event streams. As a result, the CloudEvents specification was created.

While initially the work on CloudEvents was done as part of the Serverless Working group, once the specification reached its v0.1 milestone, the TOC approved the CloudEvents work as a new stand-alone CNCF sandbox project.

CloudEvents Concepts

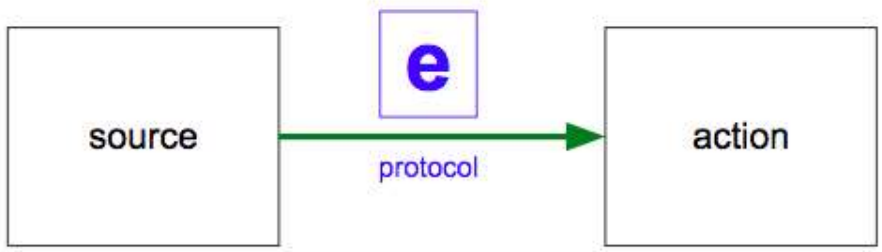
An event includes context and data about an occurrence. Each *occurrence* is uniquely identified by the data of the *event*.

Events represent facts and therefore do not include a destination, whereas messages convey intent, transporting data from a source to a given destination.

Eventing

Events are commonly used in server-side code to connect disparate systems where the change of state in one system causes code to execute in another. For example, a source may generate an event when it receives an external signal (e.g. HTTP or RPC) or observes a changing value (e.g. an IoT sensor or period of inactivity).

To illustrate how a system uses CloudEvents, the simplified diagram below shows how an event from a source triggers an action.



The source generates a message where the event is encapsulated in a protocol. The event arrives to a destination, triggering an action which is provided with the event data.

A *source* is a specific instance of a source-type which allows for staging and test instances. Open source software of a specific *source-type* may be deployed by multiple companies or providers.

Events can 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.

An action processes an event defining a behavior or effect which was triggered by a specific *occurrence* from a specific *source*. While outside of the scope of the specification, the purpose of generating an *event* is typically to allow other systems to easily react to changes in a source that they do not control. The *source* and action are typically built by different developers. Often the *source* is a managed service and the *action* is custom code in a serverless Function (such as AWS Lambda or Google Cloud Functions).

Design Goals

CloudEvents are typically used in a distributed system to allow for services to be loosely coupled during development, deployed independently, and later can be connected to create new applications.

The goal of the CloudEvents specification is to define interoperability of event systems that allow services to produce or consume events, where the producer and consumer can be developed and deployed independently. A producer can generate events before a consumer is listening, and a consumer can express an interest in an event or class of events that is not yet being produced. Note that the specifications produced by this effort are focused on interoperability of the event format and how it appears while being sent on various protocols, such as HTTP. The specifications will not focus on the processing model of either the event producer or event consumer.

CloudEvents, at its core, defines a set of metadata, called "context attributes", about the event being transferred between systems, and how those pieces of metadata should appear in that message. This metadata is meant to be the minimal set of information needed to route the request to the proper component and to facilitate proper processing of the event by that component. So, while this might mean that some of the application data of the event itself might be duplicated as part of the CloudEvent's set of attributes, this is to be done solely for the purpose of proper delivery, and processing, of the message. Data that is not intended for that purpose should instead be placed within the event (`data`) itself.

Additionally, it is assumed that the metadata needed by the protocol layer to deliver the message to the target system is handled entirely by the protocol and therefore is not included within the CloudEvents attributes. See the Non-Goals section for more details.

Along with the definition of these attributes, there will also be specifications of how to serialize the event in different formats (e.g. JSON) and protocols (e.g. HTTP, AMQP, Kafka).

Batching of multiple events into a single API call is natively supported by some protocols. To aid interoperability, it is left up to the protocols if and how batching is implemented. Details may be found in the protocol binding or in the protocol specification. A batch of CloudEvents carries no semantic meaning and is not ordered. An Intermediary can add or remove batching as well as assign events to different batches.

The purpose, or semantic meaning, of an event is out of scope for the CloudEvents specification. As long as the message being sent conforms to the specification, then it is a valid CloudEvent. As an example that might not be obvious to everyone, errors or exceptions can be transmitted as CloudEvents. It would then be up to the event producer to define the CloudEvents attribute values that would be used, just like any other event it might generate.

Since not all event producers generate their events as CloudEvents, there are a set of adapters defined that show how to map events from some popular event producers into CloudEvents. These adapters are non-normative but are the specification authors' best guess as to how the CloudEvents attribute would be populated if the event producer produced them natively.

Non-Goals

The following are considered beyond the scope of the specification:

- Function build and invocation process
- Language-specific runtime APIs
- Selecting a single identity/access control system
- Inclusion of protocol-level routing information
- Event persistence processes
- Mechanism for Authorization, Data Integrity and Confidentiality

The CloudEvents spec will not include protocol-level routing information (e.g. a destination URL to which the event is being sent). This is a common suggestion by those new to the concepts of CloudEvents. After much deliberation, the working group has come to the conclusion that routing is unnecessary in the spec: any protocol (e.g. HTTP, MQTT, XMPP, or a Pub/Sub bus) already defines semantics for routing. For example, the CloudEvents HTTP binding dictates headers and request body contents. CloudEvents don't need to include a destination URL in the spec to be HTTP compatible; the HTTP spec already includes one in the Request-Line.

Routing information is not just redundant, it detracts. CloudEvents should increase interoperability and decouple the producer and consumer of events. Prohibiting routing information from the events format allows CloudEvents to be redelivered to new actions, or to be delivered over complex relays that include multiple channels. For example, a CloudEvent that was intended for a webhook should be deliverable to a dead-letter queue if the webhook address is unavailable. That dead-letter queue should be able to feed events to new actions that the original event emitter never imagined.

The CloudEvents that are produced and consumed within and across systems trigger behaviors that derive value. As such, archiving and/or replaying events can be valuable for debugging or replication purposes. However, persisting an event removes the contextual information available during transmission such as the identity and rights of the producer, fidelity validation mechanisms, or confidentiality protections. Additionally, persistence can add complexity and challenge to meeting user's requirements. For example, the repeated use of a private key for encryption or signing purposes increases the information available to attackers and thereby reduces security. It is expected that attributes may be defined that facilitate meeting persistence requirements but it is expected that these will continuously evolve along with industry best practices and advancements.

The CloudEvents spec currently does not mandate or advocate any specific mechanism or principles in the matter of Authorization, Data Integrity and Confidentiality, as the current intention behind this spec is not to define Security principles with regards to CloudEvents. Every implementor has a different principle for enhancing their security model. We leave it up to the implementor of the spec to provide additional details for hardening their security model as an extension attribute, which can be furthermore interpreted by the components implemented by the implementor of the specification themselves. However, if the community observes a pattern in usage of certain extension attributes, as a standard way to deal with the topic of data integrity. In that case, such extension attributes can be declared as official extensions to the CloudEvent specification.

Architecture

The CloudEvents specification set defines four different kinds of protocol elements that form a layered architecture model.

1. The base specification defines an abstract information model made up of context attributes (key-value pairs) and associated rules for what constitutes a CloudEvent. This includes *core attributes* defined by the specification. Some core attributes are required on all CloudEvents, while others are optional.
2. The extensions add use-case specific and potentially overlapping sets of extension attributes and associated rules, e.g. to support different tracing standards.
3. The event format encodings, e.g. JSON, define how the information model of the base specification together with the chosen extensions is encoded for mapping it to header and payload elements of an application protocol.
4. The protocol bindings, e.g. HTTP, defines how the CloudEvent is bound to an application protocol's transport frame, in the case of HTTP to the HTTP message. The protocol binding does not constrain how the transport frame is used, meaning that the HTTP binding can be used with any HTTP method and with request and response messages.

If required to assure broader interoperability, the CloudEvents specification set will provide specific constraints for event delivery using a particular application protocol. The HTTP Webhook specification is not specific to CloudEvents and can be used to post any kind of one-way event and notifications to a conformant HTTP endpoint. However, the lack of such a specification elsewhere makes it necessary for CloudEvents to define it.

Interoperability Constraints

As stated in the Design Goals section, interoperability is a key objective of the specification. Therefore, there are places in the specification where restrictions are recommended. For example, in the Size Limits section it hints that event sizes should not exceed 64KB. It is important to note that constraints such as these, where they are not mandated via a "MUST", are recommendations to increase the likelihood of interoperability between multiple implementations and deployments. Specific uses of the specification are free to ignore these recommendations but it is incumbent on those environments to ensure that all components involved in the delivery of the CloudEvents are able to leave the boundaries of the recommendations.

Protocol Error Handling

The CloudEvents specification, for the most part, does not dictate a processing model associated with the creation or processing of CloudEvents. As such, if there are errors during the processing of a CloudEvent, the software encountering the error is encouraged to use the normal protocol-level error reporting to report them.

Versioning of CloudEvents

For many CloudEvents, the schema of the data within the event might change over time. The CloudEvents specification does not mandate any particular pattern to be used, or even the use or consideration of versioning at all. This decision is up to each event producer.

However, event producers are encouraged to consider how they might evolve their schema without breaking consumers. Two specific context attributes are particularly important in this respect, but differ in expected usage: `type` and `dataschema`. The differences between these attributes with respect to versioning are described below.

Event producers are encouraged to consider versioning from the outset, before first declaring a particular CloudEvent to be "stable". Documentation of the chosen versioning scheme, including the rationale behind it, is likely to be appreciated by consumers.

The role of the `type` attribute within versioning

The `type` attribute is expected to be the primary means by which consumers identify the type of event that they receive. This could be achieved by subscribing to a specific CloudEvent type, or by filtering all received CloudEvents by type locally. But consumers who have identified a CloudEvent type will generally expect the data within that type to only change in backwardly-compatible ways, unless clearly indicated otherwise. The precise meaning of "backwardly-compatible" will vary by data content type.

When a CloudEvent's data changes in a backwardly-compatible way, the value of the `type` attribute should generally stay the same.

When a CloudEvent's data changes in a backwardly-incompatible way, the value of the `type` attribute should generally change. The event producer is encouraged to produce both the old event and the new event for some time (potentially forever) in order to avoid disrupting consumers.

When considering the value of the `type` attribute, event producers may choose any versioning scheme they wish, such as a version number (`v1` , `v2`) or a date (`2018-01-01`) as part of the value. They may also use the `type` attribute to convey that a particular version is not yet stable, and may go through breaking changes. Alternatively, they may choose not to include a version in the type value at all.

The role of the `dataschema` attribute within versioning

The `dataschema` attribute is expected to be informational, largely to be used during development and by tooling that is able to provide diagnostic information over arbitrary CloudEvents with a data content type understood by that tooling.

When a CloudEvent's data changes in a backwardly-compatible way, the value of the `dataschema` attribute should generally change to reflect that. An alternative approach is for the URI to stay the same, but for the content served from that URI to change to reflect the updated schema. The latter approach may be simpler for event producers to implement, but is less convenient for consumers who may wish to cache the schema content by URI.

When a CloudEvent's data changes in a backwardly-incompatible way, the value of `dataschema` attribute should generally change, along with the `type` attribute as described above.

CloudEvent Core Attributes

This section provides additional background and design points related to some of the CloudEvent core attributes.

id

The `id` attribute is meant to be a value that is unique across all events related to one event source (where each event source is uniquely identified by its CloudEvents `source` attribute value). While the exact value used is producer defined, receivers of CloudEvents from a single event source can be assured that no two events will share the same `id` value. We are implicitly making a claim here that no two events will share the same `id` value, but do not provide an explanation as to how this is guaranteed, since this is out of the scope of this spec. The only exception to this is if some replay of the event is supported, and in those cases, the `id` can then be used to detect this.

Since a single occurrence may result in the generation of more than one event, in the cases where all of those events are from the same event source, each CloudEvent constructed will have a unique `id`. Take the example of the creation of a DB entry, this one occurrence might generate a CloudEvent with a type of `create` and a CloudEvent with a type of `write`. Each of those CloudEvents would have a unique `id`. If there is the desire for some correlation between those two CloudEvents to indicate they are both related to the same occurrence, then some additional data within the CloudEvent would be used for that purpose.

In this respect, while the exact value chosen by the event producer might be some random string, or a string that has some semantic meaning in some other context, for the purposes of this CloudEvent attribute those meanings are not relevant and therefore using `id` for some other purpose beyond uniqueness checking is out of scope of the specification and not recommended.

CloudEvent Extension Attributes

In order to achieve the stated goals, the specification authors will attempt to constrain the number of metadata attributes they define in CloudEvents. To that end, context attributes defined by this project will fall into two categories:

- core
- extensions

The core attributes are those defined within the specification itself. Core attributes then have subcategories of required or optional. As the category names imply, "required" attributes will be the ones that the group considers vital to all events in all use cases, while "optional" ones will be used in a majority of the cases.

When the group determines that an attribute is not common enough to fall into those two categories but would still benefit from the level of interoperability that comes from being well-defined, this repo's documented extensions provides a shared collaboration space. Each extension specification defines how it will appear within a CloudEvent. Extensions which are documented in this way have no special status, and may take breaking changes (including being removed entirely). They are not part of the main versioned CloudEvents specification, and are only present as a form of collaboration.

In determining which category a proposed attribute belongs, or even if it will be included at all, the group uses use-cases and user-stories to explain the rationale and need for them. This supporting information will be added to the Prior Art section of this document.

Extension attributes to the CloudEvent specification are meant to be additional metadata that needs to be included to help ensure proper routing and processing of the CloudEvent. Additional metadata for other purposes, that is related to the event itself and not needed in the transportation or processing of the CloudEvent, should instead be placed within the proper extensibility points of the event (`data`) itself.

Extension attributes should be kept minimal to ensure the CloudEvent can be properly serialized and transported. For example, the Event producers should consider the technical limitations that might be encountered when adding extensions to a CloudEvent. For example, the HTTP Binary Mode uses HTTP headers to transport metadata; most HTTP servers will reject requests with excessive HTTP header data, with limits as low as 8kb. Therefore, the aggregate size and number of extension attributes should be kept minimal.

If an extension becomes popular then the specification authors might consider moving it into the specification as a core attribute. This means that the extension mechanism/process can be used as a way to vet new attributes prior to formally adding them to the specification.

JSON Extensions

As mentioned in the Attributes section of the JSON Event Format for CloudEvents specification, CloudEvent extension attributes are serialized as siblings to the specification defined attributes - meaning, at the top-level of the JSON object. The authors of the specification spent a long time considering all options and decided that this was the best choice. Some of the rationale follows.

Since the specifications follow semver, this means that new properties can be defined by future versions of the core specification without requiring a major version number change - as long as these properties are optional. In those cases, consider what an existing consumer would do with a new (unknown) top-level property. While it would be free to ignore it, since it is optional, in most cases it is believed that these properties would still want to be exposed to the application receiving those events. This would allow those applications to support these properties even if the infrastructure doesn't. This means that unknown top-level properties (regardless of who defined them - future versions of the spec or the event producer) are probably not going to be ignored. So, while some other specifications define a specific property under which extensions are placed (e.g. a top-level `extensions` property), the authors decided that having two different locations within an incoming event for unknown properties could lead to interoperability issues and confusion for developers.

Often extensions are used to test new potential properties of specifications prior to them being formally adopted. If there were an `extensions` type of property, in which this new property was serialized, then if that property were to ever be adopted by the core specification it would be promoted (from a serialization perspective) from the `extensions` property to be a top-level property. If we assume that this new property will be optional, then as it is adopted by the core specification it will be just a minor version increment, and all existing consumers should still continue to work. However, consumers will not know where this property will appear - in the `extensions` property or as a top-level property. This means they might need to look in both places. What if the property appears in both place but with different values? Will producers need to place it in both places since they could have old and new consumers? While it might be possible to define clear rules for how to solve each of the potential problems that arise, the authors decided that it would be better to simply avoid all of them in the first place by only having one location in the serialization for unknown, or even new, properties. It was also noted that the HTTP specification is now following a similar pattern by no longer suggesting that extension HTTP headers be prefixed with `X-`.

Creating CloudEvents

The CloudEvents specification purposely avoids being too prescriptive about how CloudEvents are created. For example, it does not assume that the original event source is the same entity that is constructing the associated CloudEvent for that occurrence. This allows for a wide variety of implementation choices. However, it can be useful for implementors of the specification to understand the expectations that the specification authors had in mind as this might help ensure interoperability and consistency.

As mentioned above, whether the entity that generated the initial event is the same entity that creates the corresponding CloudEvent is an implementation choice. However, when the entity that is constructing/populating the CloudEvents attributes is acting on behalf of the event source, the values of those attributes are meant to describe the event or the event source and not the entity calculating the CloudEvent attribute values. In other words, when the split between the event source and the CloudEvents producer are not materially significant to the event consumers, the spec defined attributes would typically not include any values to indicate this split of responsibilities.

This isn't to suggest that the CloudEvents producer couldn't add some additional attributes to the CloudEvent, but those are outside the scope of the interoperability defined attributes of the spec. This is similar to how an HTTP proxy would typically minimize changes to the well-defined HTTP headers of an incoming message, but it might add some additional headers that include proxy-specific metadata.

It is also worth noting that this separation between original event source and CloudEvents producer could be small or large. Meaning, even if the CloudEvent producer were not part of the original event source's ecosystem, if it is acting on behalf of the event source, and its presence in the flow of the event is not meaningful to event consumers, then the above guidance would still apply.

When an entity is acting as both a receiver and sender of CloudEvents for the purposes of forwarding, or transforming, the incoming event, the degree to which the outbound CloudEvent matches the inbound CloudEvent will vary based on the processing semantics of this entity. In cases where it is acting as proxy, where it is simply forwarding CloudEvents to another event consumer, then the outbound CloudEvent will typically look identical to the inbound CloudEvent with respect to the spec defined attributes - see previous paragraph concerning adding additional attributes.

However, if this entity is performing some type of semantic processing of the CloudEvent, typically resulting in a change to the value of the `data` attribute, then it may need to be considered a distinct "event source" from the original event source. And as such, it is expected that CloudEvents attributes related to the event producer (such as `source` and `id`) would be changed from the incoming CloudEvent.

There might exist special cases in which it is necessary to create a CloudEvent that contains another CloudEvent. Although the specification does not define nesting explicitly, it is possible. When doing so, the inner event will always use structured mode, this is because if the inner event was in binary mode then its metadata could be misinterpreted as metadata for the outer event. The outer event can use either binary or structured mode, however, its `datacontenttype` attribute must not be set to `application/cloudevents+json` or any other media type that is used to denote the usage of structured mode. This is because it would then be difficult to know if the use of `application/cloudevents+json` is due to the outer event using structured mode or due to the inner event being a CloudEvent. One option in this case is to use a value of `application/json` instead, as seen in this example:

```
Content-Type: application/json
ce-specversion: 1.0
ce-type: myevent
ce-id: 1234-1234-1234
ce-source: example.com

{
  "specversion": "1.0",
  "type": "coolevent",
  "id": "xxxx-xxxx-xxxx",
  "source": "bigco.com",
  "data": { ... }
}
```

Note that by doing so the receiver can no longer know the inner event is a CloudEvent by examination of the `Content-Type` HTTP header. If this is a concern then using structured mode for the outer event might be preferable.

Qualifying Protocols and Encodings

The explicit goal of the CloudEvents effort, as expressed in the specification, is "describing event data in a common way" and "to define interoperability of event systems that allow services to produce or consume events, where the producer and consumer can be developed and deployed independently".

The foundations for such interoperability are open data formats and open protocols, with CloudEvents aiming to provide such an open data format and projections of its data format onto commonly used protocols and with commonly used encodings.

While each software or service product and project can obviously make its own choices about which form of communication it prefers, its unquestionable that a proprietary protocol that is private to such a product or project does not further the goal of broad interoperability across producers and consumers of events.

Especially in the area of messaging and eventing, the industry has made significant progress in the last decade in developing a robust and broadly supported protocol foundation, like HTTP 1.1 and HTTP/2 as well as WebSockets or events on the web, or MQTT and AMQP for connection-oriented messaging and telemetry transfers.

Some widely used protocols have become de-facto standards emerging out of strong ecosystems of top-level consortia of three or more companies, and some out of the strong ecosystems of projects released by a single company, and in either case largely in parallel to the evolution of the previously mentioned standards stacks.

The CloudEvents effort shall not become a vehicle to even implicitly endorse or promote project- or product-proprietary protocols, because that would be counterproductive towards CloudEvents' original goals.

For a protocol or encoding to qualify for a core CloudEvents event format or protocol binding, it must belong to either one of the following categories:

- The protocol has a formal status as a standard with a widely-recognized multi-vendor protocol standardization body (e.g. W3C, IETF, OASIS, ISO)
- The protocol has a "de-facto standard" status for its ecosystem category, which means it is used so widely that it is considered a standard for a given application. Practically, we would like to see at least one open source implementation under the umbrella of a vendor-neutral open-source organization (e.g. Apache, Eclipse, CNCF, .NET Foundation) and at least a dozen independent vendors using it in their products/services.

Aside from formal status, a key criterion for whether a protocol or encoding shall qualify for a core CloudEvents event format or protocol binding is whether the group agrees that the specification will be of sustained practical benefit for any party that is unrelated to the product or project from which the protocol or encoding emerged. A base requirement for this is that the protocol or encoding is defined in a fashion that allows alternate implementations independent of the product or project's code.

All other protocol and encoding formats for CloudEvents are welcome to be included in a list pointing to the CloudEvents binding information in the respective project's own public repository or site.

Proprietary Protocols and Encodings

To encourage adoption of CloudEvents, this repository will collect CloudEvent specs for proprietary protocols and encodings without endorsement. Repository maintainers are not responsible for creating, maintaining, or notifying maintainers of proprietary specs of drift from the CloudEvents spec.

Proprietary specs will be hosted in their own repository or documentation site, and collected in the proprietary-specs file. Proprietary specs should follow the same format as the other specs for core protocols and encodings.

Proprietary specs will receive less scrutiny than a core spec, and as the CloudEvents spec evolves, it is the responsibility of the maintainers of the respective protocols and encodings to keep specs in sync with the CloudEvents spec. If a proprietary spec falls too far out of date, CloudEvents may mark the link to that spec as deprecated or remove it.

In the case that multiple, incompatible specs are created for the same protocol, the repository maintainers will be agnostic about which spec is correct and list links to all specs.

Prior Art

This section describes some of the input material used by the group during the development of the CloudEvent specification.

Roles

The list below enumerates the various participants, and scenarios, that might be involved in the producing, managing or consuming of events.

In these, the roles of event producer and event consumer are kept distinct. A single application context can always take on multiple roles concurrently, including being both a producer and a consumer of events.

1. Applications produce events for consumption by other parties. Examples of this include: providing consumers with insights about end-user activities, state changes or environment observations, or for allowing complementing the application's capabilities with event-driven extensions.

   Events are typically produced related to a context or a producer-chosen classification. For example, a temperature sensor in a room might be context-qualified by mount position, room, floor, and building. A sports result(event) might be classified by league and team.

   The producer application could run anywhere, such as on a server or a device.

   The produced events might be rendered and emitted directly by the producer or by an intermediary; as example for the latter, consider event data transmitted by a device over payload-size-constrained networks such as LoRaWAN or ModBus, and where events compliant to this specification will be rendered by a network gateway on behalf of the producer.

   For example, a weather station transmits a 12-byte, proprietary event payload indicating weather conditions once every 5 minutes over LoRaWAN. A LoRaWAN gateway is then used to publish the event to an Internet destination in the CloudEvents format. The LoRaWAN gateway is the event producer, publishing on behalf of the weather station, and will set event metadata appropriately to reflect the source of the event.

2. Applications consume events for the purposes such as display, archival, analytics, workflow processing, monitoring the condition and/or providing transparency into the operation of a business solution and its foundational building blocks.

   The consumer application could run anywhere, such as on a server or a device.

   A consuming application will typically be interested in:

   - distinguishing events such that the exact same event is not processed twice.
   - identifying and selecting the origin context or the producer-assigned classification.
   - identifying the temporal order of the events relative to the originating context and/or relative to a wall-clock.
   - understanding the context-related detail information carried in the event.
   - correlating event instances from multiple event producers and send them to the same consumer context.

   In some cases, the consuming application might be interested in:

   - obtaining further details about the event's subject from the originating context, like obtaining detail information about a changed object that requires privileged access authorization. For example, a HR solution might only publish very limited information in events for privacy reasons, and any event consumer needing more data will have to

obtain details related to the event from the HR system under their own authorization context.
- interact with the event's subject at the originating context, for instance reading a storage blob after having been informed that this blob has just been created.

Consumer interests motivate requirements for which information producers ought to include an event.

3. Middleware routes events from producers to consumers, or onwards to other middleware. Applications producing events might delegate certain tasks arising from their consumers' requirements to middleware:

   - Management of many concurrent interested consumers for one of multiple classes or originating contexts of events
   - Processing of filter conditions over a class or originating context of events on behalf of consumers.
   - Transcoding, like encoding in MsgPack after decoding from JSON
   - Transformation that changes the event's structure, like mapping from a proprietary format to CloudEvents, while preserving the identity and semantic integrity of the event.
   - Instant "push-style" delivery to interested consumers.
   - Storing events for eventual delivery, either for pick-up initiated by the consumer ("pull") or initiated by the middleware ("push") after a delay.
   - Observing event content or event flow for monitoring or diagnostics purposes.

   To satisfy these needs, middleware will be interested in:

   - A metadata discriminator, usable for classification or contextualization of events so that consumers can express interest in one or multiple such classes or contexts. For instance, a consumer might be interested in all events related to a specific directory inside a file storage account.
   - A metadata discriminator, that allows distinguishing the subject of a particular event of that class or context. For instance, a consumer might want to filter out all events related to new files ending with ".jpg" (the file name being the "new file" event's subject) for the purpose of describing a specific directory inside a file storage account that it has registered interest on.
   - An indicator for the encoding of the event and its data.
   - An indicator for the structural layout (schema) for the event and its data.

   Whether its events are available for consumption via a middleware is a delegation choice of the producer.

   In practice, middleware can take on the role of a Producer when it changes the semantic meaning of an event, a Consumer when it takes action based on an event, or Intermediary when it routes events without making semantic changes.

4. Frameworks and other abstractions make interactions with the event platform infrastructure simpler, and often expose common API surface areas for other event platform infrastructures.

   Frameworks are often used for turning events into an object graph, and to dispatch the event to some specific handling user-code or user-rule that permits the consuming application to react to a particular kind of occurrence in the originating context and on a particular subject.

   Frameworks are most interested in semantic metadata commonality across the platforms they abstract, so that similar activities can be handled uniformly.

   For a sports application, a developer using the framework might be interested in all events from today's game (subject) of a team in a league (topic of interest) but wanting to handle reports of "goal" differently than reports of "substitution". For this, the framework will need a suitable metadata discriminator that frees it from having to understand the event details. To be clear, the suitable metadata discriminator should be populated by the producer, and would not be the responsibility of the framework.

Value Proposition

This section describes some of the use-cases that explain the value of CloudEvents.

Normalizing Events Across Services & Platforms

Major event publishers, (e.g. AWS, Microsoft, Google, etc.) publish events in different formats on their respective platforms. There are even a few cases where services on the same provider publish events in different formats (e.g. AWS). This forces event consumers to implement custom logic to read or munge event data across platforms and occasionally across services on a single platform.

CloudEvents can offer a single experience for authoring consumers that handle events across all platforms and services.

Facilitating Integrations Across Services & Platforms

Event data being transported across environments is increasingly common. However, without a common way of describing events, delivery of events across environments is hindered. There is no single way of determining where an event came from and where it might be going. This prevents tooling to facilitate successful event delivery and consumers from knowing what to do with event data.

CloudEvents offers useful metadata which middleware and consumers can rely upon to facilitate event routing, logging, delivery and receipt.

Increasing Portability of Functions-as-a-Service

Functions-as-a-Service (also known as serverless computing) is one of the fastest growing trends in IT and it is largely event-driven. However, a primary concern of FaaS is vendor lock-in. This lock-in is partially caused by differences in function APIs and signatures across providers, but the lock-in is also caused by differences in the format of event data received within functions.

CloudEvents' common way of describing event data increases the portability of Functions-as-a-Service.

Improving Development & Testing of Event-Driven/Serverless Architectures

The lack of a common event format complicates development and testing of event-driven and serverless architectures. There is no easy way to mock events accurately for development and testing purposes, and help emulate event-driven workflows in a development environment.

CloudEvents can enable better developer tools for building, testing and handling the end-to-end lifecycle of event-driven and serverless architectures.

Event Data Evolution

Most platforms and services version the data model of their events differently (if they do this at all). This creates an inconsistent experience for publishing and consuming the data model of events as those data models evolve.

CloudEvents can offer a common way to version and evolve event data. This will help event publishers safely version their data models based on best practices, and this help event consumers safely work with event data as it evolves.

Normalizing Webhooks

Webhooks is a style of event publishing which does not use a common format. Consumers of webhooks don't have a consistent way to develop, test, identify, validate, and overall process event data delivered via webhooks.

CloudEvents can offer consistency in webhook publishing and consumption.

Policy Enforcement

The transiting of events between systems may need to be filtered, transformed, or blocked due to security and policy concerns. Examples may be to prevent ingress or egress of the events such as event data containing sensitive information or wanting to disallow the information flow between the sender and receiver.

A common event format would allow easier reasoning about the data being transited and allow for better introspection of the data.

Event Tracing

An event sent from a source may result in a sequence of additional events sent from various middleware devices such as event brokers and gateways. A CloudEvents extension could be defined to include the metadata required to associate each individual CloudEvent with an event sequence, for the purpose of event tracing and troubleshooting.

IoT

IoT devices send and receive events related to their functionality. For example, a connected thermostat will send telemetry on the current temperature and could receive events to change temperatures. These devices typically have a constrained operating environment (cpu, memory) requiring a well-defined event message format. In a lot of cases these messages are binary encoded instead of textual. Whether directly from the device or transformed via a gateway, CloudEvents would allow for a better description of the origin of the message and the format of the data contained within the message.

Event Correlation

A serverless application/workflow could be associated with multiple events from different event sources/producers. For example, a burglary detection application/workflow could involve both a motion event and a door/window open event. A serverless platform could receive many instances of each type of events, e.g. it could receive motion events and window open events from different houses.

The serverless platform needs to correlate one type of event instance correctly with other types of event instances and map a received event instance to the correct application/workflow instance. CloudEvents will provide a standard way for any event consumer (e.g. the serverless platform) to locate the event correlation information/token in the event data and map a received event instance to the correct application/workflow instance.

Existing Event Formats

As with the previous section, the examination (and understanding) of the current state of the world was very important to the group. To that end, a sampling of existing current event formats that are used in practice today was gathered.

Microsoft - Event Grid

```
{
  "topic": "/subscriptions/{subscription-id}",
  "subject": "/subscriptions/{subscription-id}/resourceGroups/{resource-group}/provic
  "eventType": "Microsoft.Resources.ResourceWriteSuccess",
  "eventTime": "2017-08-16T03:54:38.2696833Z",
  "id": "25b3b0d0-d79b-44d5-9963-440d4e6a9bba",
  "data": {
    "authorization": "{azure_resource_manager_authorizations}",
    "claims": "{azure_resource_manager_claims}",
```

```
    "correlationId": "54ef1e39-6a82-44b3-abc1-bdeb6ce4d3c6",
    "httpRequest": "",
    "resourceProvider": "Microsoft.EventGrid",
    "resourceUri": "/subscriptions/{subscription-id}/resourceGroups/{resource-group}/
    "operationName": "Microsoft.EventGrid/eventSubscriptions/write",
    "status": "Succeeded",
    "subscriptionId": "{subscription-id}",
    "tenantId": "72f988bf-86f1-41af-91ab-2d7cd011db47"
  }
}
```

Documentation

Google - Cloud Functions (potential future)

```
{
  "data": {
    "@type": "types.googleapis.com/google.pubsub.v1.PubsubMessage",
    "attributes": {
      "foo": "bar"
     },
     "messageId": "12345",
     "publishTime": "2017-06-05T12:00:00.000Z",
     "data": "somebase64encodedmessage"
  },
  "context": {
    "eventId": "12345",
    "timestamp": "2017-06-05T12:00:00.000Z",
    "eventTypeId": "google.pubsub.topic.publish",
    "resource": {
      "name": "projects/myProject/topics/myTopic",
      "service": "pubsub.googleapis.com"
    }
  }
}
```

AWS - CloudWatch Events

A high proportion of event-processing systems on AWS are converging on the use of this format.

```
{
  "version": "0",
  "id": "6a7e8feb-b491-4cf7-a9f1-bf3703467718",
  "detail-type": "EC2 Instance State-change Notification",
  "source": "aws.ec2",
  "account": "111122223333",
  "time": "2017-12-22T18:43:48Z",
  "region": "us-west-1",
```

```
  "resources": [
    "arn:aws:ec2:us-west-1:123456789012:instance/i-1234567890abcdef0"
  ],
  "detail": {
    "instance-id": "i-1234567890abcdef0",
    "state": "terminated"
  }
}
```

IBM - OpenWhisk - Web Action Event

```
{
  "__ow_method": "post",
  "__ow_headers": {
    "accept": "*/*",
    "connection": "close",
    "content-length": "4",
    "content-type": "text/plain",
    "host": "172.17.0.1",
    "user-agent": "curl/7.43.0"
  },
  "__ow_path": "",
  "__ow_body": "Jane"
}
```

OpenStack - Audit Middleware - Event

```
{
  "typeURI": "http://schemas.dmtf.org/cloud/audit/1.0/event",
  "id": "d8304637-3f63-5092-9ab3-18c9781871a2",
  "eventTime": "2018-01-30T10:46:16.740253+00:00",
  "action": "delete",
  "eventType": "activity",
  "outcome": "success",
  "reason": {
    "reasonType": "HTTP",
    "reasonCode": "204"
  },
  "initiator": {
    "typeURI": "service/security/account/user",
    "name": "user1",
    "domain": "domain1",
    "id": "52d28347f0b4cf9cc1717c00adf41c74cc764fe440b47aacb8404670a7cd5d22",
    "host": {
      "address": "127.0.0.1",
      "agent": "python-novaclient"
    },
    "project_id": "ae63ddf2076d4342a56eb049e37a7621"
  },
```

```
  "target": {
    "typeURI": "compute/server",
    "id": "b1b475fc-ef0a-4899-87f3-674ac0d56855"
  },
  "observer": {
    "typeURI": "service/compute",
    "name": "nova",
    "id": "1b5dbef1-c2e8-5614-888d-bb56bcf65749"
  },
  "requestPath": "/v2/ae63ddf2076d4342a56eb049e37a7621/servers/b1b475fc-ef0a-4899-87f
}
```

Documentation

Adobe - I/O Events

```
{
  "event_id": "639fd17a-d0bb-40ca-83a4-e78612bce5dc",
  "event": {
    "@id": "82235bac-2b81-4e70-90b5-2bd1f04b5c7b",
    "@type": "xdmCreated",
    "xdmEventEnvelope:objectType": "xdmAsset",
    "activitystreams:to": {
      "xdmImsUser:id": "D13A1E7053E46A220A4C86E1@AdobeID",
      "@type": "xdmImsUser"
    },
    "activitystreams:generator": {
      "xdmContentRepository:root": "https://cc-api-storage.adobe.io/",
      "@type": "xdmContentRepository"
    },
    "activitystreams:actor": {
      "xdmImsUser:id": "D13A1E7053E46A220A4C86E1@AdobeID",
      "@type": "xdmImsUser"
    },
    "activitystreams:object": {
      "@type": "xdmAsset",
      "xdmAsset:asset_id": "urn:aaid:sc:us:4123ba4c-93a8-4c5d-b979-ffbbe4318185",
      "xdmAsset:asset_name": "example.jpg",
      "xdmAsset:etag": "6fc55d0389d856ae7deccebba54f110e",
      "xdmAsset:path": "/MyFolder/example.jpg",
      "xdmAsset:format": "image/jpeg"
    },
    "activitystreams:published": "2016-07-16T19:20:30+01:00"
  }
}
```

Documentation