JSON Event Format for CloudEvents - Version 1.0.2

Abstract

The JSON Format for CloudEvents defines how events are expressed in JavaScript Object Notation (JSON) Data Interchange Format (RFC8259).

1. Introduction

CloudEvents is a standardized and protocol-agnostic definition of the structure and metadata description of events. This specification defines how the elements defined in the CloudEvents specification are to be represented in the JavaScript Object Notation (JSON) Data Interchange Format (RFC8259).

The Attributes section describes the naming conventions and data type mappings for CloudEvents attributes.

The Envelope section defines a JSON container for CloudEvents attributes and an associated media type.

1.1. Conformance

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC2119.

2. Attributes

This section defines how CloudEvents attributes are mapped to JSON. This specification does not explicitly map each attribute, but provides a generic mapping model that applies to all current and future CloudEvents attributes, including extensions.

For clarity, extension attributes are serialized using the same rules as specification defined attributes. This includes their syntax and placement within the JSON object. In particular, extensions are placed as top-level JSON properties. Extensions MUST be serialized as a top-level JSON property. There were many reasons for this design decision and they are covered in more detail in the Primer.

2.1. Base Type System

The core CloudEvents specification defines a minimal abstract type system, which this mapping leans on.

2.2. Type System Mapping

The CloudEvents type system MUST be mapped to JSON types as follows, with exceptions noted below.

CloudEvents	JSON
Boolean	boolean
Integer	number, only the integer component optionally prefixed with a minus sign is permitted
String	string
Binary	string, Base64-encoded binary
URI	string following RFC 3986
URI-reference	string following RFC 3986

CloudEvents	JSON
Timestamp	string following RFC 3339 (ISO 8601)

Unset attributes MAY be encoded to the JSON value of `null`. When decoding attributes and a `null` value is encountered, it MUST be treated as the equivalent of unset or omitted.

Extension specifications MAY define secondary mapping rules for the values of attributes they define, but MUST also include the previously defined primary mapping.

For instance, the attribute value might be a data structure defined in a standard outside of CloudEvents, with a formal JSON mapping, and there might be risk of translation errors or information loss when the original format is not preserved.

An extension specification that defines a secondary mapping rule for JSON, and any revision of such a specification, MUST also define explicit mapping rules for all other event formats that are part of the CloudEvents core at the time of the submission or revision.

If required, like when decoding Maps, the CloudEvents type can be determined by inference using the rules from the mapping table, whereby the only potentially ambiguous JSON data type is `string`. The value is compatible with the respective CloudEvents type when the mapping rules are fulfilled.

2.3. Examples

The following table shows exemplary attribute mappings:

CloudEvents	Type	Exemplary JSON Value
type	String	"com.example.someevent"
specversion	String	"1.0"
source	URI-reference	"/mycontext"
subject	String	"larger-context"
subject	String (null)	null
id	String	"1234-1234-1234"
time	Timestamp	"2018-04-05T17:31:00Z"
time	Timestamp (null)	null
datacontenttype	String	"application/json"

2.4. JSONSchema Validation

The CloudEvents JSONSchema for the spec is located here and contains the definitions for validating events in JSON.

3. Envelope

Each CloudEvents event can be wholly represented as a JSON object.

Such a representation MUST use the media type `application/cloudevents+json` .

All REQUIRED and all not omitted OPTIONAL attributes in the given event MUST become members of the JSON object, with the respective JSON object member name matching the attribute name, and the member's type and value being mapped using the type system mapping.

OPTIONAL not omitted attributes MAY be represeted as a `null` JSON value.

3.1. Handling of "data"

The JSON representation of the event "data" payload is determined by the runtime type of the `data` content and the value of the `datacontenttype` attribute.

3.1.1. Payload Serialization

Before taking action, a JSON serializer MUST first determine the runtime data type of the `data` content.

If the implementation determines that the type of data is `Binary` , the value MUST be represented as a JSON string expression containing the Base64 encoded binary value, and use the member name `data_base64` to store it inside the JSON representation. If present, the `datacontenttype` MUST reflect the format of the original binary data.

If the type of data is not `Binary` , the implementation will next determine whether the value of the `datacontenttype` attribute declares the `data` to contain JSON-formatted content. Such a content type is defined as one having a media subtype equal to `json` or ending with a `+json` format extension. That is, a `datacontenttype` declares JSON-formatted content if its media type, when stripped of parameters, has the form `*/json` or `*/*+json` . If the `datacontenttype` is unspecified, processing SHOULD proceed as if the `datacontenttype` had been specified explicitly as `application/json` .

If the `datacontenttype` declares the data to contain JSON-formatted content, a JSON serializer MUST translate the data value to a JSON value, and use the member name `data` to store it inside the JSON representation. The data value MUST be stored directly as a JSON value, rather than as an encoded JSON document represented as a string. An implementation MAY fail to serialize the event if it is unable to translate the runtime value to a JSON value.

Otherwise, if the `datacontenttype` does not declare JSON-formatted data content, a JSON serializer MUST store a string representation of the data value, properly encoded according to the `datacontenttype`, in the `data` member of the JSON representation. An implementation MAY fail to serialize the event if it is unable to represent the runtime value as a properly encoded string.

Out of this follows that the presence of the `data` and `data_base64` members is mutually exclusive in a JSON serialized CloudEvent.

Furthermore, unlike attributes, for which value types are restricted by the type-system mapping, the `data` member JSON value is unrestricted, and MAY contain any valid JSON if the `datacontenttype` declares the data to be JSON-formatted. In particular, the `data` member MAY have a value of `null`, representing an explicit `null` payload as distinct from the absence of the `data` member.

3.1.2. Payload Deserialization

When a CloudEvents is deserialized from JSON, the presence of the `data_base64` member clearly indicates that the value is a Base64 encoded binary data, which the deserializer MUST decode into a binary runtime data type. The deserializer MAY further interpret this binary data according to the `datacontenttype`.

When a `data` member is present, the decoding behavior is dependent on the value of the `datacontenttype` attribute. If the `datacontenttype` declares the `data` to contain JSON-formatted content (that is, its subtype is `json` or has a `+json` format extension), then the `data` member MUST be treated directly as a JSON value and decoded using an appropriate JSON type mapping for the runtime. Note: if the `data` member is a string, a JSON deserializer MUST interpret it directly as a JSON String value; it MUST NOT further deserialize the string as a JSON document.

If the `datacontenttype` does not declare JSON-formatted data content, then the `data` member SHOULD be treated as an encoded content string. An implementation MAY fail to deserialize the event if the `data` member is not a string, or if it is unable to interpret the `data` with the `datacontenttype`.

When a `data` member is present, if the `datacontenttype` attribute is absent, a JSON deserializer SHOULD proceed as if it were set to `application/json`, which declares the data to contain JSON-formatted content. Thus, it SHOULD treat the `data` member directly as a JSON value as specified above. Furthermore, if a JSON-formatted event with no `datacontenttype` attribute, is deserialized and then re-serialized using a different format or protocol binding, the `datacontenttype` in the re-serialized event SHOULD be set explicitly to the implied `application/json` content type to preserve the semantics of the event.

3.2. Examples

Example event with `Binary`-valued data:

```
{
    "specversion" : "1.0",
    "type" : "com.example.someevent",
    "source" : "/mycontext",
    "id" : "A234-1234-1234",
    "time" : "2018-04-05T17:31:00Z",
    "comexampleextension1" : "value",
    "comexampleothervalue" : 5,
    "datacontenttype" : "application/vnd.apache.thrift.binary",
    "data_base64" : "... base64 encoded string ..."
}
```

The above example re-encoded using HTTP Binary Content Mode:

```
ce-specversion: 1.0
ce-type: com.example.someevent
ce-source: /mycontext
ce-id: A234-1234-1234
ce-time: 2018-04-05T17:31:00Z
ce-comexampleextension1: value
ce-comexampleothervalue: 5
content-type: application/vnd.apache.thrift.binary

...raw binary bytes...
```

Example event with a serialized XML document as the `String` (i.e. non- `Binary`) valued `data` , and an XML (i.e. non-JSON-formatted) content type:

```
{
    "specversion" : "1.0",
    "type" : "com.example.someevent",
    "source" : "/mycontext",
    "id" : "B234-1234-1234",
    "time" : "2018-04-05T17:31:00Z",
    "comexampleextension1" : "value",
    "comexampleothervalue" : 5,
    "unsetextension": null,
    "datacontenttype" : "application/xml",
    "data" : "<much wow=\"xml\"/>"
}
```

The above example re-encoded using HTTP Binary Content Mode:

```
ce-specversion: 1.0
ce-type: com.example.someevent
ce-source: /mycontext
ce-id: B234-1234-1234
```

```
ce-time: 2018-04-05T17:31:00Z
ce-comexampleextension1: value
ce-comexampleothervalue: 5
content-type: application/xml

<much wow="xml"/>
```

Example event with JSON Object-valued `data` and a content type declaring JSON-formatted data:

```
{
    "specversion" : "1.0",
    "type" : "com.example.someevent",
    "source" : "/mycontext",
    "subject": null,
    "id" : "C234-1234-1234",
    "time" : "2018-04-05T17:31:00Z",
    "comexampleextension1" : "value",
    "comexampleothervalue" : 5,
    "datacontenttype" : "application/json",
    "data" : {
        "appinfoA" : "abc",
        "appinfoB" : 123,
        "appinfoC" : true
    }
}
```

The above example re-encoded using HTTP Binary Content Mode:

```
ce-specversion: 1.0
ce-type: com.example.someevent
ce-source: /mycontext
ce-id: C234-1234-1234
ce-time: 2018-04-05T17:31:00Z
ce-comexampleextension1: value
ce-comexampleothervalue: 5
content-type: application/json

{
  "appinfoA" : "abc",
  "appinfoB" : 123,
  "appinfoC" : true
}
```

Example event with a literal JSON string as the non- `Binary` -valued `data` and no `datacontenttype` . The data is implicitly treated as if the `datacontenttype` were set to `application/json` :

```
{
    "specversion" : "1.0",
    "type" : "com.example.someevent",
    "source" : "/mycontext",
    "subject": null,
    "id" : "D234-1234-1234",
    "time" : "2018-04-05T17:31:00Z",
    "comexampleextension1" : "value",
    "comexampleothervalue" : 5,
    "data" : "I'm just a string"
}
```

The above example re-encoded using HTTP Binary Content Mode. Note that the Content Type is explicitly set to the `application/json` value that was implicit in JSON format. Note also that the content is quoted to indicate that it is a literal JSON string. If the quotes were missing, this would have been an invalid event because the content could not be decoded as `application/json`:

```
ce-specversion: 1.0
ce-type: com.example.someevent
ce-source: /mycontext
ce-id: D234-1234-1234
ce-time: 2018-04-05T17:31:00Z
ce-comexampleextension1: value
ce-comexampleothervalue: 5
content-type: application/json

"I'm just a string"
```

4. JSON Batch Format

In the *JSON Batch Format* several CloudEvents are batched into a single JSON document. The document is a JSON array filled with CloudEvents in the JSON Event format.

Although the *JSON Batch Format* builds ontop of the *JSON Format*, it is considered as a separate format: a valid implementation of the *JSON Format* doesn't need to support it. The *JSON Batch Format* MUST NOT be used when only support for the *JSON Format* is indicated.

4.1. Mapping CloudEvents

This section defines how a batch of CloudEvents is mapped to JSON.

The outermost JSON element is a JSON Array, which contains as elements CloudEvents rendered in accordance with the JSON event format specification.

4.2. Envelope

A JSON Batch of CloudEvents MUST use the media type `application/cloudevents-batch+json` .

4.3. Examples

An example containing two CloudEvents: The first with `Binary` -valued data, the second with JSON data.

```
[
  {
      "specversion" : "1.0",
      "type" : "com.example.someevent",
      "source" : "/mycontext/4",
      "id" : "B234-1234-1234",
      "time" : "2018-04-05T17:31:00Z",
      "comexampleextension1" : "value",
      "comexampleothervalue" : 5,
      "datacontenttype" : "application/vnd.apache.thrift.binary",
      "data_base64" : "... base64 encoded string ..."
  },
  {
      "specversion" : "1.0",
      "type" : "com.example.someotherevent",
      "source" : "/mycontext/9",
      "id" : "C234-1234-1234",
      "time" : "2018-04-05T17:31:05Z",
      "comexampleextension1" : "value",
      "comexampleothervalue" : 5,
      "datacontenttype" : "application/json",
      "data" : {
          "appinfoA" : "abc",
          "appinfoB" : 123,
          "appinfoC" : true
      }
  }
]
```

An example of an empty batch of CloudEvents (typically used in a response, but also valid in a request):

```
[]
```

5. References

- RFC2046 Multipurpose Internet Mail Extensions (MIME) Part Two: Media Types
- RFC2119 Key words for use in RFCs to Indicate Requirement Levels
- RFC4627 The application/json Media Type for JavaScript Object Notation (JSON)

- RFC4648 The Base16, Base32, and Base64 Data Encodings
- RFC6839 Additional Media Type Structured Syntax Suffixes
- RFC8259 The JavaScript Object Notation (JSON) Data Interchange Format