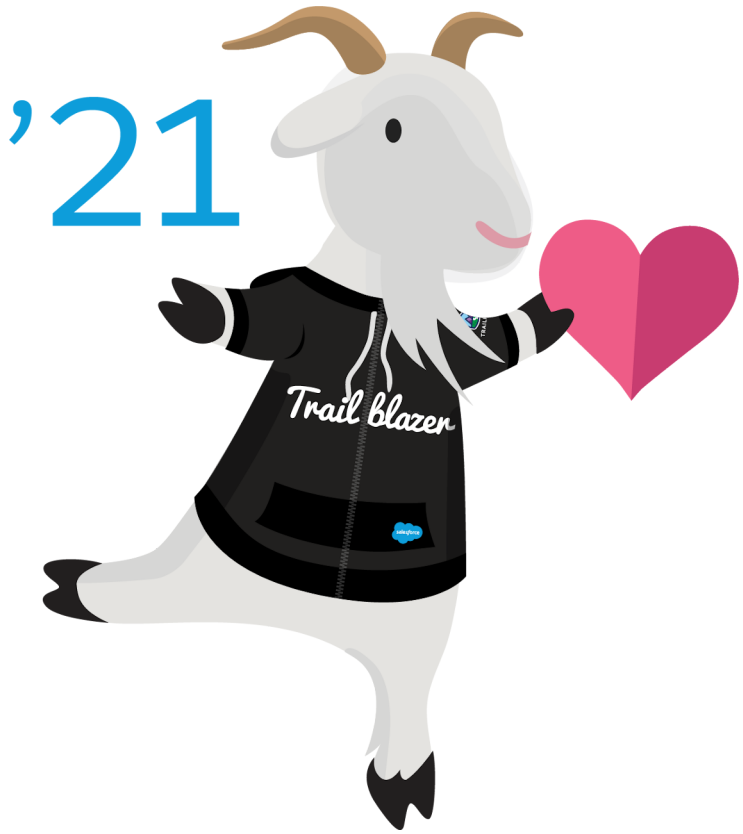
Change Data Capture Developer Guide

Version 52.0, Summer '21



@salesforcedocs

Last updated: July 21, 2021

© Copyright 2000–2021 salesforce.com, inc. All rights reserved. Salesforce is a registered trademark of salesforce.com, inc., as are other names and marks. Other marks appearing herein may be trademarks of their respective owners.

CONTENTS

Change Data Capture 1

Keep Your External Data Current with Change Data Capture 2
- When Do You Use Change Data Capture? 2
- Change Event Object Support 3
- Select Objects for Change Notifications in the User Interface 4
- Select Objects for Change Notifications with Metadata API and Tooling API 4

Change Event Message Structure 5
- Change Event Header Fields 7
- Change Event Body Fields 9

Merged Change Events 10

Other Types of Change Events: Gap and Overflow Events 11
- Gap Events 11
- Overflow Events 12

Subscribe to Change Events 13
- Change Event Storage and Delivery 14
- Subscription Channels 15
- Compose Streams of Change Data Capture Notifications with Custom Channels 16
- Example Diagrams for Channels and Channel Members 17
- High-Level Replication Steps 20
- Subscribe with CometD Using the EMP Connector Sample 22
- Enrich Change Events with Extra Fields When Subscribed with CometD 23
- Subscribe with Apex Triggers 31

Monitor Change Event Publishing and Delivery Usage 41

Security Considerations 43
- Required Permissions for Change Events Received by CometD Subscribers 43
- Field-Level Security 44
- Change Events for Encrypted Salesforce Data 44

Change Event Considerations 46
- General Considerations 47
- Change Data Capture Allocations 47

Standard Object Notes 51
- Change Events for Tasks and Events 51
- Change Events for Person Accounts 57
- Change Events for Users 60
- Change Events for Lead Conversion 61

Change Events for Fields 65
- Sending Data Differences for Fields of Updated Records 65
- Change Events for Custom Field Type Conversions 69

CHANGE DATA CAPTURE

Receive near-real-time changes of Salesforce records, and synchronize corresponding records in an external data store. Change Data Capture publishes change events, which represent changes to Salesforce records. Changes include creation of a new record, updates to an existing record, deletion of a record, and undeletion of a record.

EDITIONS

Available in: both Salesforce Classic and Lightning Experience

Available in: **Enterprise**, **Performance**, **Unlimited**, and **Developer** editions

IN THIS SECTION:

Keep Your External Data Current with Change Data Capture

Use Change Data Capture to update data in an external system instead of doing periodic exports and imports of data or repeated API calls. Capturing changes with Change Data Capture event notifications ensures that your external data can be updated in real time and stays fresh.

Change Event Message Structure

A change event message has the following structure. All change events contain the same header fields.

Merged Change Events

For efficiency, sometimes change events for one transaction are merged into one event if the same change occurred in multiple records of the same object type during one second.

Other Types of Change Events: Gap and Overflow Events

Other types of change events are provided to handle special situations, such as capturing changes not caught in the Salesforce application servers, or handling high loads of changes.

Subscribe to Change Events

You subscribe to change events with CometD or Apex triggers. CometD is a messaging library that enables listening to events through long polling and simulates push technology. Apex triggers for change events are similar to Apex triggers on platform events.

Monitor Change Event Publishing and Delivery Usage

To get usage data for event publishing and CometD-client delivery, query the PlatformEventUsageMetric object. Usage data is available for the last 24 hours, ending at the last hour, and for historical daily usage. PlatformEventUsageMetric is available in API version 50.0 and later.

Security Considerations

Learn about the user permissions required for subscription, field-level security, and Shield Platform Encryption.

Change Event Considerations

Keep in mind change event considerations and allocations when subscribing to change events.

Standard Object Notes

Learn about the characteristics of change events for some standard objects and the fields included in the event messages.

Change Events for Fields

Learn about the change event characteristics for fields.

Keep Your External Data Current with Change Data Capture

Use Change Data Capture to update data in an external system instead of doing periodic exports and imports of data or repeated API calls. Capturing changes with Change Data Capture event notifications ensures that your external data can be updated in real time and stays fresh.

IN THIS SECTION:

When Do You Use Change Data Capture?

You can think of Change Data Capture as part of the real-time data replication process for the cloud.

Change Event Object Support

Change events are available for all custom objects defined in your Salesforce org and a subset of standard objects.

Select Objects for Change Notifications in the User Interface

To receive notifications on the default standard channel for record changes, select the custom objects and supported standard objects that you're interested in on the Change Data Capture page.

Select Objects for Change Notifications with Metadata API and Tooling API

Use PlatformEventChannelMember in Metadata API or Tooling API to create or retrieve object event selections for the default standard channel or a custom channel. The default standard channel, ChangeEvents, corresponds to the selections that you configure in Setup in the Change Data Capture page. For a custom channel, the object selections are set when you create the channel.

When Do You Use Change Data Capture?

You can think of Change Data Capture as part of the real-time data replication process for the cloud.

Data replication includes these stages.

1. Initial (day 0) copy of the entire data set to the external system.
2. Continuous synchronization of new and updated data to the external system.
3. Reconciliation of duplicate data between the two systems.

Change Data Capture is the continuous synchronization part of replication (step 2). It publishes the deltas of Salesforce data for new and changed records. Change Data Capture requires an integration app for receiving events and performing updates in the external system.

For example, you have a human resource (HR) system with copies of employee custom object records from Salesforce. You can synchronize the employee records in the HR system by receiving change events. You can then process the corresponding insert, update, delete, or undelete operations in the HR system. Because the changes are received in near real time, the data in your HR system stays up to date.

Change Data Capture enables secure and scalable event streaming to downstream systems. An integration app can receive millions of events per day and synchronize data with another system. The event retention of three days enables a CometD subscriber to get past event messages. Encryption and field-level security enable secure event storage and communication.

Use Change Data Capture to:

- Keep external systems in sync with Salesforce data.
- Receive notifications of Salesforce record changes, including create, update, delete, and undelete operations.
- Subscribe using CometD or Apex triggers.
- Capture field changes for all records.
- Get broad access to all data regardless of sharing rules.
- Deliver only the fields a user has access to based on field-level security.

- Encrypt change event fields at rest.
- Get information about the change in the event header, such as the origin of the change, which allows ignoring changes that your client generates.
- Perform data updates using transaction boundaries.
- Use a versioned event schema.
- Subscribe to mass changes in a scalable way.
- Get access to retained events for up to three days.

We don't recommend using Change Data Capture to:

- Perform audit trails based on record and field changes.
- Update the UI for many users in apps subscribed with CometD. Change Data Capture is intended to keep downstream systems in sync but not individual users. If many users are subscribed with CometD clients, the concurrent client limit can be hit. For more information, see Change Data Capture Allocations.

Change Data Capture Reliability

The temporary storage of change events in the event bus enhances the reliability of event delivery. CometD subscribers can catch up on events that were missed due to an offline subscriber or a connection error. For more information about how to replay events using CometD, see Message Durability in the *Streaming API Developer Guide*.

Change events are temporarily persisted to and served from an industry-standard distributed system. A distributed system doesn't have the same semantics or guarantees as a transactional database. Change events are queued and buffered, and Salesforce attempts to publish the events asynchronously. In rare cases, the event message might not be persisted in the distributed system during the initial or subsequent attempts. In those cases, the events aren't delivered to subscribers and aren't recoverable.

Change Event Object Support

Change events are available for all custom objects defined in your Salesforce org and a subset of standard objects.

For a list of objects that support change events, see `StandardObjectName`ChangeEvent in the *Object Reference for Salesforce and Lightning Platform*.

Note: Not all objects may be available in your org. Some objects require specific feature settings and permissions to be enabled.

SEE ALSO:

Object Reference for Salesforce and Lightning Platform: Standard Objects

Salesforce Help: Create Partner Users

Loyalty Management Developer Guide: Standard Objects

Select Objects for Change Notifications in the User Interface

To receive notifications on the default standard channel for record changes, select the custom objects and supported standard objects that you're interested in on the Change Data Capture page.

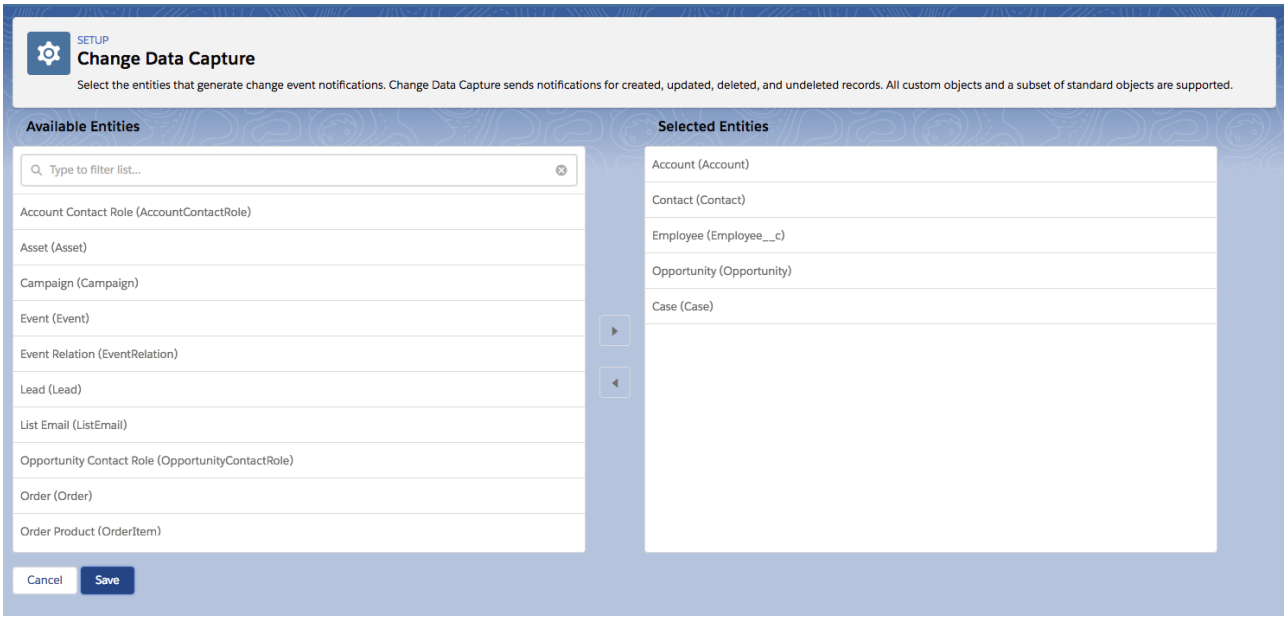
From Setup, in the Quick Find box, enter `Change Data Capture`, and click **Change Data Capture**. The Available Entities list shows the objects available in your Salesforce org for Change Data Capture. You can select up to five entities, including standard and custom objects. To enable more entities, contact Salesforce to purchase an add-on license. The add-on license removes the limit on the number of entities you can select. Also, it increases the CometD event delivery allocation. With the add-on license, you can select up to 10 entities at a time in the Available Entities list. After selecting the first 10 entities, you can add more.

Each list entry is in the format "Entity Label (API Name)." Because an entity label can be renamed, the API name is provided in parentheses to better identify the entity.

USER PERMISSIONS

To view the Change Data Capture page:
- View Setup and Configuration

To add or modify entity selections:
- Customize Application



Note:

- To filter the list of entities by typing a name in the filter field, use the same case as the entity because the filter is case-sensitive.
- The Change Data Capture page shows the object selections for the default standard channel. It doesn't show the selections for custom channels. For more information about custom channels, see Compose Streams of Change Data Capture Notifications with Custom Channels.

Select Objects for Change Notifications with Metadata API and Tooling API

Use PlatformEventChannelMember in Metadata API or Tooling API to create or retrieve object event selections for the default standard channel or a custom channel. The default standard channel, ChangeEvents, corresponds to the selections that you configure in Setup in the Change Data Capture page. For a custom channel, the object selections are set when you create the channel.

To learn how to select objects with Metadata API, see PlatformEventChannelMember in the Metadata API Developer Guide.

To learn how to select objects with Tooling API, see PlatformEventChannelMember in the Tooling API Developer Guide.

To learn about custom channels, see Compose Streams of Change Data Capture Notifications with Custom Channels, PlatformEventChannel in the Metadata API Developer Guide, and PlatformEventChannel in the Tooling API Developer Guide.

Starting with API version 47.0, you define channel member components and channels separately in Metadata API. In API version 45.0 and 46.0, members are included in the PlatformEventChannel component.

Change Event Message Structure

A change event message has the following structure. All change events contain the same header fields.

```
{
  "data": {
    "schema": "<schema_ID>",
    "payload": {
      "ChangeEventHeader": {
         "entityName" : "...",
         "recordIds" : "...",
         "changeType" : "...",
         "changedFields": [...],
         "changeOrigin" : "...",
         "transactionKey" : "...",
         "sequenceNumber" : "...",
         "commitTimestamp" : "...",
         "commitUser" : "...",
         "commitNumber" : "..."
      },
     "field1":"...",
     "field2":"...",
     . . .
    },
    "event": {
      "replayId": <replayID>
    }
  },
  "channel": "/data/<channel>"
}
```

Note: The order of the fields in the JSON event message isn't guaranteed. The order is based on the underlying Avro schema that change events are based on. When an event is expanded into JSON format, the order of the fields doesn't always match the schema depending on the client used to receive the event.

Change Event Fields

The fields that a change event can include correspond to the fields on the associated parent Salesforce object, with a few exceptions. For example, AccountChangeEvent fields correspond to the fields on Account.

The fields that a change event doesn't include are:

- The `IsDeleted` system field.
- The `SystemModStamp` system field.
- Any field whose value isn't on the record and is derived from another record or from a formula, except roll-up summary fields, which are included. Examples are formula fields. Examples of fields with derived values include `LastActivityDate` and `PhotoUrl`.

Each change event also contains header fields. The header fields are included inside the `ChangeEventHeader` field. They contain information about the event, such as whether the change was an update or delete and the name of the object, like Account.

In addition to the event payload, the event schema ID is included in the `schema` field. Also included is the event-specific field, `replayId`, which is used for retrieving past events.

API Version and Event Schema

When you subscribe to change events, the subscription uses the latest API version regardless of the API version that the client uses. The event messages received reflect the latest field definitions of the corresponding Salesforce object. When the object schema changes, such as when a field is added or a field type is changed, the schema ID changes. The change event contains the new schema ID in the `schema` field.

To get the full schema of a change event message, make a GET request to the REST API that includes the schema ID sent in the event message:

```
/vXX.X/event/eventSchema/<Schema_ID>?payloadFormat=COMPACT
```

Or make a GET request to this resource:

```
/vXX.X/sobjects/<EventName>/eventSchema?payloadFormat=COMPACT
```

<EventName> is the name of a change event, such as AccountChangeEvent.

The event schema REST API resources return the schema ID in the `uuid` field. To compare the schema with a previous version, retrieve the schema with a previous schema ID and the current schema ID.

The event schema REST API resources are also used for platform events. For more information, see Platform Event Schema by Event Name and Platform Event Schema by Schema ID in the REST API Developer Guide.

Change Event Example

The following event is sent for a new account.

```
{
  "schema": "I8b-dYxvxs5wOtCBr4qsew",
  "payload": {
    "ChangeEventHeader": {
      "entityName": "Account",
      "changeType": "CREATE",
      "changedFields": [],
      "changeOrigin": "com/salesforce/api/soap/47.0;client=SfdcInternalAPI/",
      "transactionKey": "000177a7-c079-6e50-73af-5af790992cc1",
      "sequenceNumber": 1,
      "commitTimestamp": 1564443438000,
      "commitNumber": 74523894988,
      "commitUser": "<User_ID>",
      "recordIds": [
        "<record_ID>"
      ]
    },
    "Name": "Acme",
    "Description": "Everyone is talking about the cloud. But what does it mean?",
    "OwnerId": "<Owner_ID>",
    "CreatedDate": "2019-07-29T23:37:18.000Z",
    "CreatedById": "<User_ID>",
    "LastModifiedDate": "2019-07-29T23:37:18.000Z",
    "LastModifiedById": "<User_ID>",
```

```
  },
  "event": {
    "replayId": 10
  }
}
```

For more change event examples, see Subscribe to an Event Channel in the Change Data Capture Basics Trailhead module.

IN THIS SECTION:

Change Event Header Fields
Check out the descriptions of the fields that the change event header contains.

Change Event Body Fields
The body of a change event message includes the fields and values for the corresponding Salesforce record.

Change Event Header Fields

Check out the descriptions of the fields that the change event header contains.

Field Name	Field Type	Description
`entityName`	string	The API name of the standard or custom object that the change pertains to. For example, Account or MyObject__c.
`recordIds`	string[]	One or more record IDs for the changed records. Typically, this field contains one record ID. If in one transaction the same change occurred in multiple records of the same object type during one second, Salesforce merges the change notifications. In this case, Salesforce sends one change event for all affected records and the `recordIds` field contains the IDs for all records that have the same change. Examples of operations with same changes are: • Update of fieldA to valueA in Account records. • Deletion of Account records. • Renaming or replacing a picklist value that results in updating the field value in all affected records. The `recordIds` field can contain a wildcard value when a change event message is generated for custom field type conversions that cause data loss. In this case, the `recordIds` value is the three-character prefix of the object, followed by the wildcard character `*`. For example, for accounts, the value is `001*`. For more information, see Conversions That Generate a Change Event.
`changedFields`	string[]	A list of the fields that were changed in an update operation, including the `LastModifiedDate` system field. This field is empty for other operations, including record creation.
`changeType`	Enumeration	The operation that caused the change.Can be one of the following values: • CREATE • UPDATE

Field Name	Field Type	Description
		• DELETE • UNDELETE For gap events, the change type starts with the GAP_ prefix. • GAP_CREATE • GAP_UPDATE • GAP_DELETE • GAP_UNDELETE For overflow events, the change type is GAP_OVERFLOW.

`changeOrigin`	string	Only populated for changes done by API apps or from Lightning Experience; empty otherwise. The Salesforce API and the API client ID that initiated the change, if set by the client. Use this field to detect whether your app initiated the change to not process the change again and potentially avoid a deep cycle of changes.
		The format of the `changeOrigin` field value is:
		`com/salesforce/api/<API_Name>/<API_Version>;client=<Client_ID>`
		• `<API_Name>` is the name of the Salesforce API used to make the data change. It can take one of these values: soap, rest, bulkapi, xmlrpc, oldsoap, toolingsoap, toolingrest, apex, apexdebuggerrest.
		• `<API_Version>` is the version of the API call that made the change and is in the format *XX.X*.
		• `<Client_ID>` is a string that contains the client ID of the app that initiated the change. If the client ID is not set in the API call, `client=<Client_ID>` is not appended to the `changeOrigin` field.
		Example:
		`com/salesforce/api/soap/49.0;client=Astro`
		The client ID is set in the Call Options header of an API call. For an example on how to set the Call Options header, see:
		• REST API: Sforce-Call-Options Header. (Bulk API also uses the Sforce-Call-Options header.)
		• SOAP API: CallOptions Header. (Apex API also uses the CallOptions element.)

`transactionKey`	string	A string that uniquely identifies each Salesforce transaction. You can use this key to identify and group all changes that were made in the same transaction.
`sequenceNumber`	int	The sequence of the change within a transaction. The sequence number starts from 1. A lead conversion is an example of a transaction that can have multiple changes. A lead conversion results in the following sequence of changes, all within the same transaction. **1.** Create an account

Field Name	Field Type	Description
		2. Create a contact **3.** Create an opportunity **4.** Update a lead For more information, see Change Events for Lead Conversion.
`commitTimestamp`	long	The date and time when the change occurred, represented as the number of milliseconds since January 1, 1970 00:00:00 GMT.
`commitUser`	string	The ID of the user that ran the change operation.
`commitNumber`	long	The system change number (SCN) of a committed transaction, which increases sequentially. This field is provided for diagnostic purposes. The field value is not guaranteed to be unique in Salesforce—it is unique only in a single database instance. If your Salesforce org migrates to another database instance, the commit number might not be unique or sequential.

Change Event Body Fields

The body of a change event message includes the fields and values for the corresponding Salesforce record.

JSON Change Event Messages

The fields that Salesforce includes in a JSON event message that a CometD client receives depend on the operation performed.

Create
For a new record, the event message body includes all non-empty fields and system fields, such as the `CreatedDate` and `OwnerId` fields.

Update
For an updated record, the body includes only the changed fields. It includes empty fields only if they are updated to an empty value (null). It also includes the `LastModifiedDate` system field. The body includes the `LastModifiedById` field only if it has changed—if the user who modified the record is different than the previous user who saved it.

Delete
For a deleted record, the body doesn't include any fields or system fields.

Undelete
For an undeleted record, the body includes all non-empty fields from the original record, in addition to system fields.

Apex Change Event Messages

Fields in a change event message are statically defined, just like in any other Apex type. As a result, all record fields are available in the change event message received in an Apex trigger, regardless of the operation performed. The event message can contain empty (null) fields.

Create
For a new record, the event message contains all fields, whether populated or empty. It includes fields with default values and system fields, such as `CreatedDate` and `OwnerId`.

Update
For an updated record, the event message contains field values only for changed fields. Unchanged fields are present and empty (null), even if they contain a value in the record. The event message also contains the `LastModifiedDate` system field. The body includes the `LastModifiedById` field only if it has changed—if the user who modified the record is different than the previous user who saved it.

Delete
For a deleted record, all record fields in the event message are empty (null).

Undelete
For an undeleted record, the event message contains all fields from the original record, including empty (null) fields and system fields.

Merged Change Events

For efficiency, sometimes change events for one transaction are merged into one event if the same change occurred in multiple records of the same object type during one second.

When change events are merged, Salesforce sends one change event for all affected records and the `recordIds` field contains the IDs for all records that have the same change.

Examples of operations with same changes are:

- Update of fieldA to valueA in Account records.
- Deletion of Account records.
- Renaming or replacing a picklist value that results in updating the field value in all affected records.

For more information about the `recordIds` field, see Change Event Header Fields.

Example: If you update the `Industry` field to `Apparel` of three Account records in a single update Apex DML statement, one merged change event is sent as shown in this example. The `recordIds` field contains the IDs of the Account records that have the same change.

```
{
  "schema": "I8b-dYxvxs5wOtCBr4qsew",
  "payload": {
    "LastModifiedDate": "2021-01-20T22:33:10Z",
    "Industry": "Apparel",
    "ChangeEventHeader": {
        "commitNumber": 10708862594919,
        "commitUser": "005B0000006GudtIAC",
        "sequenceNumber": 1,
        "entityName": "Account",
        "changeType": "UPDATE",
        "changedFields": [
            "Industry",
            "LastModifiedDate"
        ],
        "changeOrigin": "com/salesforce/api/soap/51.0;client=devconsole",
        "transactionKey": "0003d969-c182-f926-e9fd-146339d7288c",
        "commitTimestamp": 1611181990000,
        "recordIds": [
            "001B000001LiHPKIA3",
            "001B000001Lw7SFIAZ",
```

```
            "001B000001Lw7SKIAZ"
        ]
    }
  },
  "event": {
    "replayId": 899
  }
}
```

Other Types of Change Events: Gap and Overflow Events

Other types of change events are provided to handle special situations, such as capturing changes not caught in the Salesforce application servers, or handling high loads of changes.

IN THIS SECTION:

Gap Events
Salesforce sometimes sends gap events instead of change events to inform subscribers about errors, or if it's not possible to generate change events. A gap event contains information about the change in the header, such as the change type and record ID. It doesn't include details about the change, such as record fields.

Overflow Events
To capture changes more efficiently, overflow events are generated for single transactions that exceed a threshold.

Gap Events

Salesforce sometimes sends gap events instead of change events to inform subscribers about errors, or if it's not possible to generate change events. A gap event contains information about the change in the header, such as the change type and record ID. It doesn't include details about the change, such as record fields.

The conditions that cause gap events include:

- The change event size exceeds the maximum 1 MB message size.
- Some field type conversions of custom fields. For more information, see Conversions That Generate a Gap Event on page 70.
- When an internal error occurs in Salesforce preventing the change event from being generated.
- Changes that occur outside the application server transaction and are applied directly in the database. For example, archiving of activities or a data cleanup job in the database. To not miss these operations, gap events are generated to notify you about those changes.

Gap events can have one of these `changeType` values in the event header.

- `GAP_CREATE`
- `GAP_UPDATE`
- `GAP_DELETE`
- `GAP_UNDELETE`

Note: A `changeType` value of `GAP_OVERFLOW` means that the event is an overflow event. For more information, see Overflow Events.

Upon receiving a gap event message, your application can retrieve the Salesforce record using the record ID value to get the current data for your system. You can get the Salesforce record with a SOAP API `retrieve()` call or through a SOQL query.

The gap event's `transactionKey` represents the internal database transaction ID if the change was applied at the database layer, outside an application server transaction. If the gap event was emitted due to other reasons, such as hitting the 1 MB event size limit or an internal error, the `transactionKey` holds the application server transaction ID.

Note: If the same type of change occurs on the same Salesforce entity within the same transaction, sometimes multiple gap events are merged into a single gap event. The IDs of the changed records are included in the `recordIds` header field.

Example: This sample gap event is for an archived task and contains information about the change in the header. The change type for archiving is `GAP_UPDATE`. The `sequenceNumber` field is always set to 1.

```
{
  "data":{
    "schema":"5GiC-KFwBGmSyDn8mqrXgg",
    "payload":{
      "ChangeEventHeader":{
        "commitNumber":288672011,
        "commitUser":"005xx000001TT96AAG",
        "sequenceNumber":1,
        "entityName":"Task",
        "changeType":"GAP_UPDATE",
        "changedFields": [],
        "changeOrigin":"",
        "transactionKey":"0Mcxx000000000B",
        "commitTimestamp":1509502088161,
        "recordIds":[
            "00Uxx000000fvRQEAY"  ]
      }
    },
    "event":{
        "replayId":15
    }
  },
  "channel":"/data/ChangeEvents"
}
```

SEE ALSO:

SOAP API Developer Guide: retrieve()

Force.com SOQL and SOSL Reference

Overflow Events

To capture changes more efficiently, overflow events are generated for single transactions that exceed a threshold.

The first 100,000 changes generate change events. The set of changes beyond that amount generates one overflow event for each entity type included in that set. An overflow event is generated when a single transaction contains more than 100,000 changes. An overflow event contains only header fields. The changeType field header value is `GAP_OVERFLOW` instead of the specific type of change. The object type corresponding to the change is in the `entityName` field. An overflow event doesn't include details about the change, such as the record fields or record ID.

A record creation, deletion, or undeletion counts as one change toward the threshold. However, in a record update, each field change counts toward the overflow threshold. For example, if three field values are modified in one record update, they count as three operations against the overflow threshold.

Transactions with a high volume of operations aren't frequent, but they can occur in certain situations, such as for a recurring event with hundreds of occurrences and attendees. Another example is a cascade delete of accounts associated with many opportunities, contacts, and activities that results in deleting many more records in the same transaction. If the cascade delete results in the deletion of 120,000 account, opportunity, contact, and activity records in the same transaction, the deletions of the first 100,000 records generate delete change events. The remaining 20,000 records generate overflow events.

Note: Because changes are sometimes merged in one change event, the number of generated change events isn't always equal to the number of changes. For example, the consecutive deletion of accounts can be merged into one change event. For more information, see the `recordIds` field in Change Event Header Fields. If Apex triggers fire and create other records, more change events are generated in the same transaction.

Example: This overflow event is for a task and contains information about the change in the header. The change type is `GAP_OVERFLOW`. The record ID for the change is always set to `000000000000000AAA`, which is the empty record ID.

```
{
  "data": {
    "schema": "DSE-RYlg-96DubFClb3e4Q",
    "payload": {
      "ChangeEventHeader": {
        "commitNumber": 170898760,
        "commitUser": "005xx000001SySSAA0",
        "sequenceNumber": 11,
        "entityName": "Task",
        "changeType": "GAP_OVERFLOW",
        "changedFields": [],
        "changeOrigin": "",
        "transactionKey": "0000161d-7324-6ed7-36f8-16ebb5d61ec9",
        "commitTimestamp": 1513039469186,
        "recordIds": [
          "000000000000000AAA"
        ]
      }
    },
    "event": {
      "replayId": 13
    }
  },
    "channel": "/data/ChangeEvents"
}
```

Subscribe to Change Events

You subscribe to change events with CometD or Apex triggers. CometD is a messaging library that enables listening to events through long polling and simulates push technology. Apex triggers for change events are similar to Apex triggers on platform events.

IN THIS SECTION:

Change Event Storage and Delivery

Change events are stored temporarily and subscribers can retrieve them during the retention window. The order of events delivered is based on the order of the corresponding committed transactions. Users with the proper permissions can receive events on a channel.

Subscription Channels

A subscription channel is a stream of change events that correspond to one or more entities. You can subscribe to a channel to receive change event notifications for record create, update, delete, and undelete operations. Change Data Capture provides predefined standard channels and you can create your own custom channels. Use the subscription channel that corresponds to the change events you want to receive. The channel name is case-sensitive.

Compose Streams of Change Data Capture Notifications with Custom Channels

Create a custom channel if you have multiple subscribers and each subscriber receives change events from a different set of entities. Also, use a custom channel with event enrichment to isolate sending enriched fields in change events on a specific channel. Custom channels group and isolate change events for each subscriber so subscribers receive only the types of events they need.

Example Diagrams for Channels and Channel Members

Discover the relationship between channels, channel members, and enriched fields with the Entity Relationship Diagram (ERD). Also, understand the benefits of using custom channels through the example diagrams.

High-Level Replication Steps

To maintain an accurate replica of your Salesforce org's data in another system, subscribe using a transaction-based approach.

Subscribe with CometD Using the EMP Connector Sample

Salesforce provides a connector to CometD that subscribes to streaming events. The EMP connector sample is a thin wrapper around the CometD library and simplifies subscribing to data change events in Java. The tool subscribes to a channel, receives notifications, and supports replaying events with durable streaming.

Enrich Change Events with Extra Fields When Subscribed with CometD

Change event messages include values for new and changed fields, but sometimes unchanged field values are needed for processing or replicating data. For example, use enrichment when your app needs an external ID field for matching records in an external system. Or always include a field that provides important information about the changed record. You can select any field whose type is supported.

Subscribe with Apex Triggers

With Apex triggers, you can capture and process change events on the Lightning Platform. Change event triggers run asynchronously after the database transaction is completed. Perform resource-intensive business logic asynchronously in the change event trigger, and implement transaction-based logic in the Apex object trigger. By decoupling the processing of changes, change event triggers can help reduce transaction processing time.

SEE ALSO:

General Considerations

Streaming API Developer Guide

Change Event Storage and Delivery

Change events are stored temporarily and subscribers can retrieve them during the retention window. The order of events delivered is based on the order of the corresponding committed transactions. Users with the proper permissions can receive events on a channel.

Temporary Storage in the Event Bus

Change events are based on platform events and share some of their characteristics for storage. Change event messages are stored in the event bus for three days. You can retrieve stored event messages from the event bus. Each event message contains the `ReplayId` field, which identifies the event in the stream and enables replaying the stream after a specific event. For more information, see Message Durability in the Streaming API Developer Guide.

Order of Events

The order of change events stored in the event bus corresponds to the order in which the transactions corresponding to the record changes are committed in Salesforce. If a transaction includes multiple changes, like a lead conversion, a change event is generated for each change with the same `transactionKey` but different `sequenceNumber` in the header. The `sequenceNumber` is the order of the change within the transaction.

When Salesforce receives a change event, it assigns a replay ID value to it and persists it in the event bus. Subscribers receive change events from the event bus in the order of the replay ID.

User Permissions Required

The subscriber must have one or more of the following permissions depending on the subscription channel: View All Data, View All Users, and View All for an object. See Required Permissions for Change Events Received by CometD Subscribers.

Subscription Channels

A subscription channel is a stream of change events that correspond to one or more entities. You can subscribe to a channel to receive change event notifications for record create, update, delete, and undelete operations. Change Data Capture provides predefined standard channels and you can create your own custom channels. Use the subscription channel that corresponds to the change events you want to receive. The channel name is case-sensitive.

Using CometD-based apps such as EMP Connector or the `empApi` Lightning component, you can subscribe to channels by supplying the channel endpoint. For example, to subscribe to events for all selected entities, subscribe to `/data/ChangeEvents`. Apex triggers can't subscribe to channels but can subscribe to a single event. For example, you can create an Apex trigger on AccountChangeEvent to subscribe to only Account change events.

Standard Channels

The ChangeEvents standard channel contains change events from one or more selected entities in a single stream that you can subscribe to. If you expect change events from more than one entity, use the ChangeEvents standard channel. To receive change events on the ChangeEvents channel, select the entities for Change Data Capture. For more information, see Select Objects for Change Notifications in the User Interface and Select Objects for Change Notifications with Metadata API and Tooling API. Then subscribe to the appropriate channel.

If you expect change events for only a single entity, use single-entity channels. With single-entity channels, you can subscribe to change events from only one custom object or standard object. Select the entity for notifications on the Change Data Capture page in Setup or in a custom channel.

Standard Channel for All Selected Entities

```
/data/ChangeEvents
```

Single-Entity Channel for a Standard Object

```
/data/<Standard_Object_Name>ChangeEvent
```

For example, the channel to subscribe to change events for Account records is:

```
/data/AccountChangeEvent
```

Single-Entity Channel for a Custom Object

```
/data/<Custom_Object_Name>__ChangeEvent
```

For example, the channel to subscribe to change events for Employee__c custom object records is:

```
/data/Employee__ChangeEvent
```

Custom Channels

Create a custom channel if you have multiple subscribers and each subscriber receives change events from a different set of entities. Also, use a custom channel with event enrichment to isolate sending enriched fields in change events on a specific channel. Custom channels group and isolate change events for each subscriber so subscribers receive only the types of events they need. Entities are automatically selected for change event notifications when you create a custom channel that includes them. A custom channel has the following format.

```
/data/YourChannelName__chn
```

For example, if your channel name is `SalesEvents`, the subscription channel is:

```
/data/SalesEvents__chn
```

SEE ALSO:

- Required Permissions for Change Events Received by CometD Subscribers
- Compose Streams of Change Data Capture Notifications with Custom Channels
- Enrich Change Events with Extra Fields When Subscribed with CometD
- Example Diagrams for Channels and Channel Members

Compose Streams of Change Data Capture Notifications with Custom Channels

Create a custom channel if you have multiple subscribers and each subscriber receives change events from a different set of entities. Also, use a custom channel with event enrichment to isolate sending enriched fields in change events on a specific channel. Custom channels group and isolate change events for each subscriber so subscribers receive only the types of events they need.

For example, if a subscriber uses real-time information about sales objects such as Account, Contact, or Order, you can create a custom channel with these objects. When you subscribe to the custom channel, you receive change events only for these objects. Your subscriber doesn't receive change events of entities selected in another channel.

You can create a custom channel with Metadata API or Tooling API. When you create a custom channel, the objects are selected for notifications when you add a PlatformEventChannelMember. Custom channels can't be created or viewed in the user interface on the Change Data Capture page. Use Metadata API to deploy or retrieve channel metadata in your org with a supported tool. Use Tooling API to create channels using REST and query channel metadata with SOQL.

Also, you can package channels to distribute with your apps.

In Metadata API, use the PlatformEventChannel metadata type to create a custom channel and the PlatformEventChannelMember type to add the selected event entities. For more information, see PlatformEventChannel and PlatformEventChannelMember in the Metadata API Developer Guide.

In Tooling API, use the PlatformEventChannel object to create a custom channel and PlatformEventChannelMember to add the selected event entities. For more information, see PlatformEventChannel and PlatformEventChannelMember in the Tooling API.

SEE ALSO:

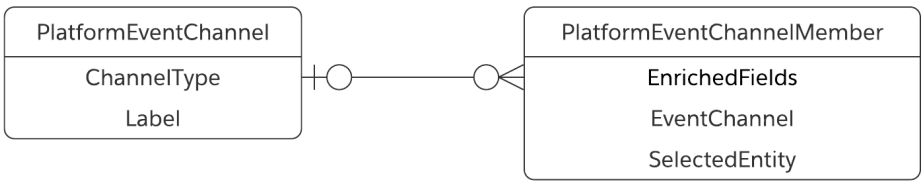
Subscription Channels
Required Permissions for Change Events Received by CometD Subscribers
Change Data Capture Allocations
Enrich Change Events with Extra Fields When Subscribed with CometD
Example Diagrams for Channels and Channel Members

Example Diagrams for Channels and Channel Members

Discover the relationship between channels, channel members, and enriched fields with the Entity Relationship Diagram (ERD). Also, understand the benefits of using custom channels through the example diagrams.

Entity Relationship Diagram for Channel and Channel Member

This ERD shows the channel and channel member entities and the relationships between them. You can access the entities by their corresponding types and objects in Metadata API and Tooling API. The entities in this diagram don't include the FullName field. FullName is the unique name of the Metadata API component or Tooling API object and is used to perform operations on them.



A channel can have zero or more channel members. A channel member can have zero or more enriched fields. You can add and update enriched fields through the PlatformEventChannelMember entities in Metadata API or Tooling API.

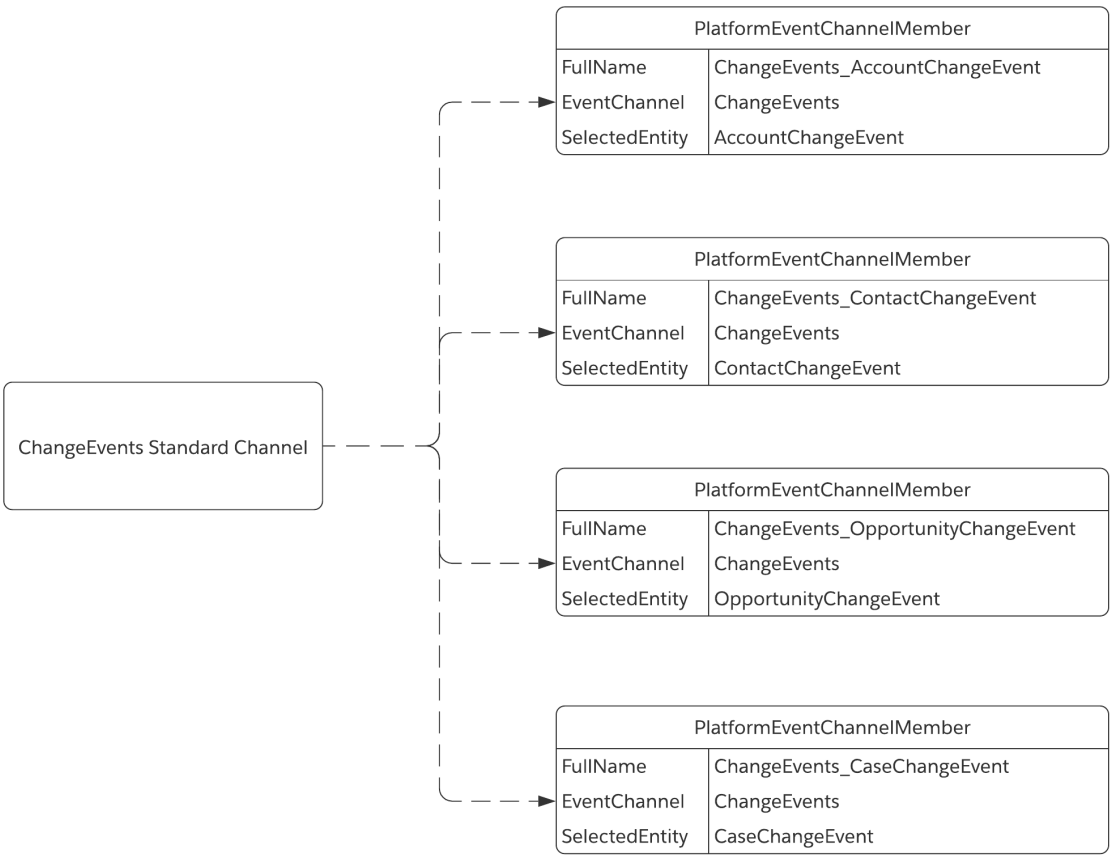
In API version 47.0 and later, the PlatformEventChannel in Metadata API and Tooling API represents a custom channel but not the standard ChangeEvents channel. The reason is you can't create or modify the ChangeEvents standard channel.

The diagrams in the next sections show examples of selected entities on the default ChangeEvents channel and on custom channels. The examples show the benefits of using custom channels. Some channel members include enriched fields.

Example for the ChangeEvents Standard Channel

This diagram is an example of four entities selected for the ChangeEvents standard channel (Account, Contact, Opportunity, and Case). The channel members don't contain enriched fields. Even though you can add enriched fields to members belonging to the ChangeEvents channel, it is best to add them on custom channels. That way, you isolate change events with enriched fields when using multiple subscribers.

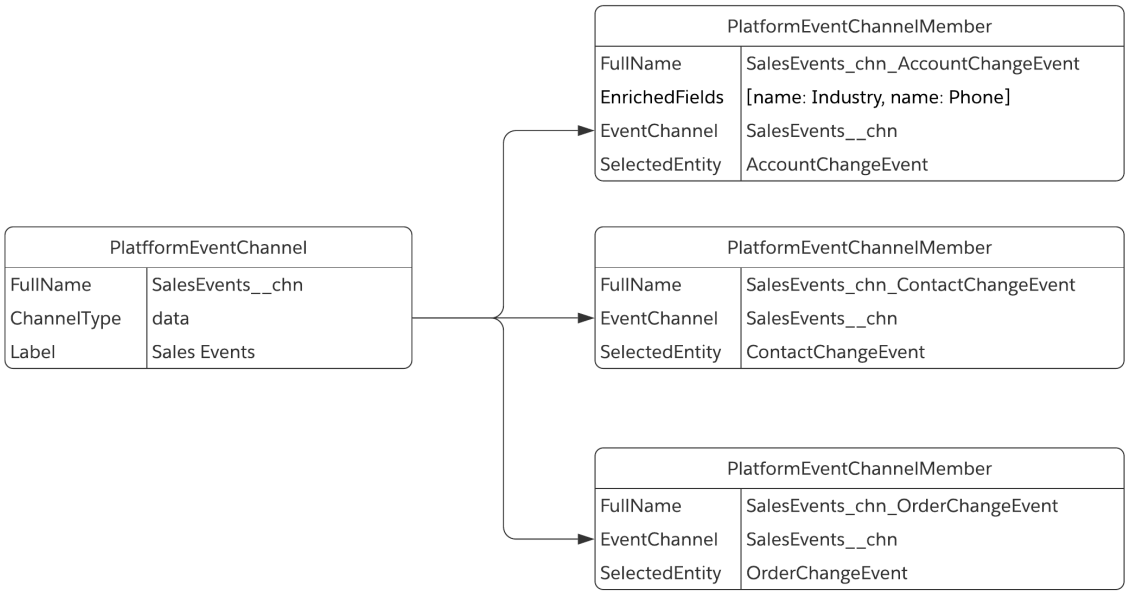
In this diagram, the ChangeEvents channel is not represented as a PlatformEventChannel entity. The reason is, in API version 47.0 and later, the ChangeEvents standard channel can't be manipulated directly through Metadata API or Tooling API. After all, it is a standard channel.



The AccountChangeEvent selected entity in one channel member is also part of custom channel examples in the next sections. But on the custom channels, AccountChangeEvent contains enriched fields. These fields are not present on the ChangeEvents channel, so they aren't sent to subscribers on this channel.

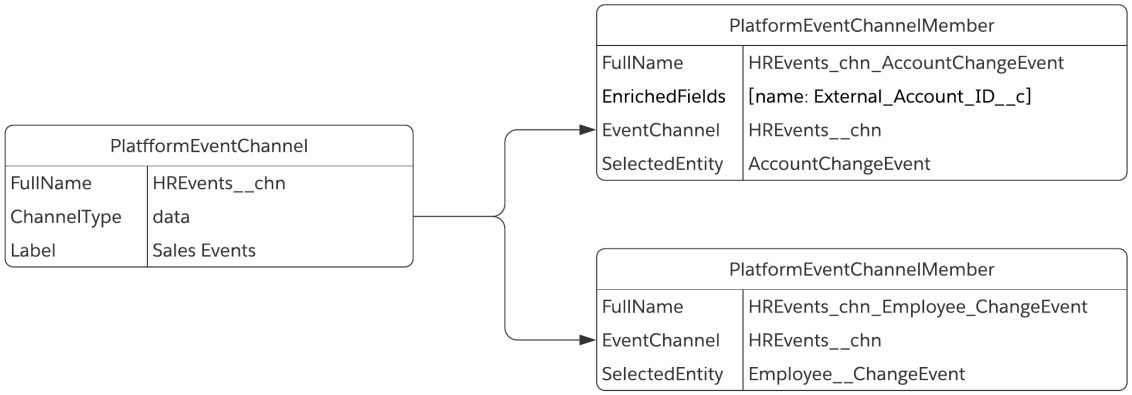
Example for SalesEvents__chn Custom Channel

This diagram shows how you can use a custom channel to isolate change events so a subscriber receives only the events they are interested in. In our example, the SalesEvents__chn channel contains a subset of the selected entities of the ChangeEvents channel: Account and Contact. It also contains one other entity: Order. One of the channel member's selected entities, AccountChangeEvent, contains two enriched fields in the EnrichedFields field. The EnrichedFields field is an array containing the name of each enriched field: the Industry and the Phone fields. The enriched field values are available in the account change events on this channel and not on other channels when not specified.



Example for HREvents__chn Custom Channel

This diagram shows an example of a custom channel, HREvents__chn, designed for entities related to an HR app: Account and the Employee__c custom object. The AccountChangeEvent selected entity is also part of the other example channels. In this channel, AccountChangeEvent contains the External_Account_ID__c custom field as an enriched field. The subscriber to the HREvents__chn custom channel receives the enriched field values in account change events.



AccountChangeEvent is a selected entity in members of two other channels: the standard ChangeEvents channel and the custom SalesEvents__chn. A subscriber on the ChangeEvents channel receives account change events without enriched fields. A subscriber on the SalesEvents__chn custom channel receives account change events enriched with the Industry and Phone field values.

SEE ALSO:

- Enrich Change Events with Extra Fields When Subscribed with CometD
- *Metadata API Developer Guide*: PlatformEventChannel
- *Metadata API Developer Guide*: PlatformEventChannelMember
- *Tooling API Developer Guide*: PlatformEventChannel
- *Tooling API Developer Guide*: PlatformEventChannelMember

High-Level Replication Steps

To maintain an accurate replica of your Salesforce org's data in another system, subscribe using a transaction-based approach.

Types of Events That Change Data Capture Can Generate: Change Events, Gap Events, and Overflow Events

Generally, Salesforce captures record changes by sending change events, which the subscriber receives to synchronize data in an external system. Sometimes, gap events or overflow events are generated.

Gap events are generated when change events can't be generated. They inform subscribers about errors or operations done outside of Salesforce application servers. Gap events don't contain record data, but they contain the record ID, which enables you to retrieve the record from Salesforce. Ensure that the subscriber expects to receive gap events and handles them properly, as outlined in the next section. The `changeType` field in the gap event header identifies the gap event and the associated operation, and can take one of these values:

- `GAP_CREATE`
- `GAP_UPDATE`
- `GAP_DELETE`
- `GAP_UNDELETE`

For more information about gap events, see Gap Events.

Overflow events are generated when a single transaction involves more than 100,000 changes. The first 100,000 changes generate change events. The set of changes beyond that amount generates one overflow event for each entity type included in that set. Overflow events include header fields but no record data and no record ID. Ensure that the subscriber handles overflow events. The `changeType` field header value is `GAP_OVERFLOW` instead of the specific type of change.

For more information about overflow events, see Overflow Events.

Transaction-Based Replication Approach

Each change event contains a transaction key in the header that uniquely identifies the transaction that the change is part of. Each change event also contains a sequence number that identifies the sequence of the change within a transaction. The sequence number is useful for operations that include multiple steps, such as lead conversion. If not all objects involved in a transaction are enabled for Change Data Capture, there will be a gap in the sequence numbers. We recommend that you replicate all the changes in one transaction as a single commit in your system. One approach is to buffer all changes related to a transaction and commit them all at once.

If you choose not to use a transaction-based replication process, your replicated data can be incomplete if your subscription stops. For example, if your subscription stops in the middle of an event stream for one transaction, only part of the transaction's changes are replicated in your system.

A transaction-based replication process involves these high-level steps.

1. In your subscribed client, allocate a transaction buffer for each transaction key. For example, create a map (`Map<String, List<ChangeEvent>>`) where the key is the `transactionKey` value.
2. Open a CometD subscription to the general `/data/ChangeEvents` channel that captures all enabled events.
3. For each change event received over the channel, check the `changeType` field.
   a. If the `changeType` field is `GAP_CREATE`, `GAP_UPDATE`, `GAP_DELETE`, or `GAP_UNDELETE`, the event is a gap event. Follow the recommended steps in How to Handle a Gap Event on page 21.
   b. If the `changeType` field is `GAP_OVERFLOW`, the event is an overflow event.
      i. Process the change events that you previously stored in the map. Commit the changes, and then purge the corresponding map entry.
      ii. For the overflow event, follow the recommended steps in How to Handle an Overflow Event on page 22.
4. If the event isn't a gap or overflow event, it's a change event. Deserialize the change event, and add it to the appropriate map entry for the transaction key.
5. When the `transactionKey` value changes in the next change event, commit the changes in the map entry for the previous transaction key, and then purge the map entry.
6. Repeat steps 3 through 5 for each new event received.

How to Handle a Gap Event

If the event that the subscriber receives is a gap event, get the latest data from Salesforce. The gap event includes the ID of the affected record enabling you to retrieve the record. After receiving the gap event, one approach is to mark the corresponding record as dirty and not process any change events for that record until it has been reconciled.

Let's look at an example to examine the steps a subscriber can take to handle a gap event while change events are also received. Records A and B are modified in a transaction and generate two change events. Then a change for record C generates a gap event. The subscriber receives three events: two change events for record A and B and one gap event for record C. The steps for the subscriber are:

1. Handle the change events according to the transaction-based replication process.
2. For the gap event, mark the corresponding record as dirty as of the date of the gap event. Use the `commitTimestamp` header field value of this gap event as the date to compare with the `commitTimestamp` values of other events received.
3. If you receive newer change events for the same record, don't process them. For example, if record C is modified again and a change event is received, ignore it because the corresponding record is marked as dirty.
4. Reconcile the data for record C. Make a Salesforce API call, such as a REST API call, to retrieve the full data for record C, and save it in your system. Then clear the dirty flag on that record.
5. Record C is modified again and a new change event is received. Process this change event according to the replication process because the record is no longer dirty.

Note: If the same type of change occurs on the same entity within the transaction, sometimes multiple gap events are merged into a single gap event. The IDs of the changed records are included in the `recordIds` header field. Use these IDs to reconcile all the referenced records.

How to Handle an Overflow Event

If a change results in more than 100,000 events in a single transaction, you receive overflow events for the events sent after the first 100,000. One overflow event is generated for each entity type. Mass changes aren't frequent. They can result from creating or modifying many records, such as changing a recurring calendar event series with many occurrences and invitees. A large change can also result from a cascade delete when deleting records with many related records.

An overflow event doesn't contain the record ID and only a dummy record ID, so one approach for data replication is to retrieve all records of the corresponding entity after an overflow event is received. Then you can update or delete those records in the external system. This approach can be the most process-intensive because it resyncs all the records for an entity. However, it's the simplest approach because it doesn't require figuring out which records changed in a particular timeframe and filtering out the records that resulted in change events. These steps outline the process of reconciling data when the overflow event is received.

1. After you receive an overflow event in your subscriber, unsubscribe from the channel, and stop processing further events. This step is in preparation of a full data synchronization for the entity.
2. Store the Replay ID of the overflow event. This ID is the starting point for the data reconciliation.
3. Reconcile the data for new, updated, and undeleted records.
   a. Retrieve all records for the entity. Depending on the volume of records stored, this process can take some time.
   b. Synchronize the data in your system by overwriting it with the retrieved data from Salesforce.
4. Reconcile the data for deleted records by performing one of the following steps.
   a. Get the non-deleted records from Salesforce, and synchronize.
      i. Identify all records for that entity in your system that weren't updated through the synchronization that you performed in step 3. These records are the deleted ones.
      ii. Delete the identified records from your system.
   b. Or get the deleted records from Salesforce, and synchronize.
      i. Query all records for the entity with `isDeleted=true`. You get all the soft-deleted records for that entity that are in the Recycle Bin.
      ii. Identify the records in your system that match the records returned in the previous step.
      iii. Delete the identified records from your system.
5. Resubscribe to the stored event bus stream starting from the Replay ID you saved earlier.
6. We recommend that you process all change events after that Replay ID. This way, you catch up on any data changes that happened during the synchronization and weren't saved in your system.
7. If you encounter an overflow event for another entity (`entityName` field value), repeat this process for that entity.

Subscribe with CometD Using the EMP Connector Sample

Salesforce provides a connector to CometD that subscribes to streaming events. The EMP connector sample is a thin wrapper around the CometD library and simplifies subscribing to data change events in Java. The tool subscribes to a channel, receives notifications, and supports replaying events with durable streaming.

Important: EMP Connector is a free, open-source, community-supported tool. Salesforce provides this tool as an example of how to subscribe to events using CometD. To contribute to the EMP Connector project with your own enhancements, submit pull requests to the repository at https://github.com/forcedotcom/EMP-Connector.

Prerequisites:

- Git. See Git Downloads.
- Apache Maven. This example uses Apache Maven to build the EMP Connector project. Download and install it from the Apache Maven site.
- Java Development Kit 8 or later. See Java Downloads.

Steps:

1. To get a local copy of the EMP-Connector GitHub repository:

```
$ git clone https://github.com/forcedotcom/EMP-Connector.git
```

2. To build the EMP-Connector tool, enter these commands:

```
$ cd EMP-Connector
$ mvn clean package
```

The generated JAR file includes the connector and the `DevLoginExample` functionality. The shaded JAR contains all the dependencies for the connector, so you don't have to download them separately. The JAR file has a -phat Maven classifier.

3. Run this command.

```
$ java -classpath
target/emp-connector-0.0.1-SNAPSHOT-phat.jar
com.salesforce.emp.connector.example.DevLoginExample
<login_URL> <username> <password> <channel>
```

For *<login_URL>*, use your org's My Domain login URL, including the https:// prefix. For example, `https://MyDomainName.my.salesforce.com`. If you don't have a deployed My Domain, use `https://login.salesforce.com` for a production org and `https://test.salesforce.com` for a sandbox or developer environment.

For *<channel>*, use the channel to subscribe to. For example, `/data/ChangeEvents`.

SEE ALSO:

Streaming API Developer Guide: Message Durability

Enrich Change Events with Extra Fields When Subscribed with CometD

Change event messages include values for new and changed fields, but sometimes unchanged field values are needed for processing or replicating data. For example, use enrichment when your app needs an external ID field for matching records in an external system. Or always include a field that provides important information about the changed record. You can select any field whose type is supported.

Event enrichment is supported for subscribers that use CometD, such as Streaming API clients or EMP Connector. Fields that you select for enrichment are included in change events for update and delete operations. Enriched fields aren't included in change events for create and undelete operations because these events contain all the populated fields. If the enriched fields have an empty value, they aren't included in the event messages. If the enriched fields are updated to null, they're included in the event as changed fields and not as enriched fields.

Note: If your client-side parsing code expects only changed fields in the event payload, the presence of enriched fields can require a change to the code. Check your client-side code and modify it if necessary. You can determine which fields changed by using the `changedFields` header field. For more information, see Change Event Header Fields.

Event enrichment is available for channels that support multiple entities, such as the standard `/data/ChangeEvents` channel, or custom channels, such as `/data/SalesEvents__chn`. You can't add enrichment directly to single-entity channels such as

`/data/<Entity>ChangeEvent`. For example, say that you want to add the Account `Industry` field for enrichment. You can do that to the custom channel, `SalesEvents__chn`, assuming AccountChangeEvent is a member of that channel. Then, if you subscribe to `/data/SalesEvents__chn`, the `Industry` field is included in account change events on that channel. If you subscribe to another channel that isn't enriched with this field, such as `/data/ChangeEvents`, or another custom channel, account change events don't include the `Industry` field.

We recommend that you configure event enrichment on a custom channel and not the standard `/data/ChangeEvents` channel. This way, other subscribers that receive change events on the standard channel don't receive unchanged fields that they don't expect. If you create a custom channel and configure event enrichment on it, you isolate the fields sent to only the clients that anticipate those fields. To learn how to create a custom channel, see Compose Streams of Change Data Capture Notifications with Custom Channels.

As part of the fields that a change event object contains, these field types are supported for enriched fields.

- Address
- Auto Number
- Checkbox
- Date, Date/Time, Time
- Email
- External Lookup Relationship
- Geolocation
- Hierarchical Relationship on User
- Lookup Relationship
- Master-Detail Relationship
- Name
- Number
- Percent
- Phone (and Fax)
- Picklist (not Multi-Select)
- Roll-Up Summary
- Text
- URL

Note:

- Compound fields, such as Name, Address, and Geolocation fields are supported for enriched fields. You can specify an entire compound field for enrichment but not directly the individual constituent fields. All non-empty constituent fields are returned as part of the compound field in the enriched change event. For example, you can enrich an event with the Lead `Name` field. The enriched change event contains the non-empty constituent fields as part of the `Name` field, including `FirstName`, `MiddleName`, `LastName`, and `Suffix`.
- For a relationship field, you can select only the field as an enriched field. You can't traverse the fields on the related object. The enriched change event contains the ID of the related record. For example, to enrich a contact change event with the ID of the related account, select the `Account` relationship field as the name of the enriched field for ContactChangeEvent, and not `Account.Name`. For custom relationship fields, specify the relationship field name with the `__c` suffix, such as `RelField__c`.

Select fields to enrich your change event messages using the PlatformEventChannelMember object in Tooling API or Metadata API. You can select up to 10 fields for a selected object across all channels. A compound field counts as one field. Only unique fields are counted across all channels for the same object, so the same field is counted only once. For example, AccountChangeEvent is a member of two

channels. It contains five enriched fields on the ChangeEvents channel and four fields on a custom channel, two of which are also on the ChangeEvents channel. The total number of enriched fields that count toward the maximum is seven.

IN THIS SECTION:

Example: Add Event Enrichment Fields with Tooling API
To add event enrichment fields, use the PlatformEventChannelMember Tooling API object, and specify the fields, the channel, and channel member.

Example: Add Event Enrichment Fields with Metadata API
To add event enrichment fields, use the PlatformEventChannelMember metadata type, and specify the fields, the channel, and channel member.

Example: Delivered Enriched Event Messages
Check out example event messages that contain enriched fields for update and delete operations.

Event Enrichment Considerations
Keep in mind these considerations when using enriched change events.

Example: Add Event Enrichment Fields with Tooling API

To add event enrichment fields, use the PlatformEventChannelMember Tooling API object, and specify the fields, the channel, and channel member.

Note: To carry out these steps in Trailhead and earn a badge, check out Create a Custom Channel and Enrich Change Events.

If the channel member you're enriching is part of a custom channel, create the custom channel first, as shown in this example. You can skip this step if using the ChangeEvents standard channel, or if you created the custom channel earlier.

Make a POST request to this REST endpoint:

```
/services/data/v52.0/tooling/sobjects/PlatformEventChannel
```

Request body for the custom channel:

```
{
  "FullName": "SalesEvents__chn",
  "Metadata": {
    "channelType": "data",
    "label": "Custom Channel for Sales App"
  }
}
```

To add enrichment fields, perform a REST request that creates a PlatformEventChannelMember component using Tooling API. In this example, the component contains two enriched fields in the `enrichedFields` array for AccountChangeEvent on the SalesEvents custom channel. `External_Account_ID__c` is a custom field defined in your org.

Make a POST request to this REST endpoint (API version 51.0 or later is supported for enrichment fields):

```
/services/data/v52.0/tooling/sobjects/PlatformEventChannelMember
```

Request body with enrichment fields added in a channel member:

```
{
  "FullName": "SalesEvents_AccountChangeEvent",
  "Metadata": {
    "enrichedFields": [
```

```
      {
        "name": "External_Account_ID__c"
      },
      {
        "name": "Industry"
      }
    ],
    "eventChannel": "SalesEvents__chn",
    "selectedEntity": "AccountChangeEvent"
  }
}
```

Query Enriched Fields

To find out which channel members and fields you configured, query the EnrichedField object in Tooling API. For example, this query returns the selected enriched field and the channel member ID.

```
SELECT ChannelMemberId,Field FROM EnrichedField ORDER BY ChannelMemberId
```

You can perform a query using the Query Editor in the Developer Console and by checking **Use Tooling API**. For more information, see Developer Console Query Editor in *Salesforce Help*.

Alternatively, you can run a query using REST API. Perform a GET request to the following URI. The URI includes the query with spaces replaced with `+`.

```
/services/data/v52.0/tooling/query/?q=SELECT+ChannelMemberId,Field+FROM+EnrichedField+ORDER+BY+ChannelMemberId
```

In these query results, the rows returned are for the same channel member. They contain two enriched fields: the `Industry` field and a custom field, whose value is an ID.

ChannelMemberId	Field
0v8RM00000000JsYAI	Industry
0v8RM00000000JsYAI	00NRM000001gEx32AE

Update a Channel Member with Enriched Fields

If there's an existing channel member for the same selected entity and channel, you can't create a duplicate channel member with a POST request. Instead, update the channel member with a PATCH request. Alternatively, you can delete the channel member and recreate it with the enriched fields.

To update a channel member, follow these steps.

1. If you're using a custom channel, get the channel ID by running this query:

   ```
   SELECT Id FROM PlatformEventChannel WHERE DeveloperName=Channel_Name
   ```

   `DeveloperName` doesn't contain the `__chn` suffix of a custom channel name. For example, for the SalesEvents__chn channel, the query would be:

   ```
   SELECT Id FROM PlatformEventChannel WHERE DeveloperName='SalesEvents'
   ```

2. Get the channel member ID with this Tooling API query. For a custom channel, replace *`Channel_ID`* with the ID you got in the previous step, or for the standard ChangeEvents channel, replace *`Channel_ID`* with `ChangeEvents`. Replace ***`Entity`***`ChangeEvent` with the selected entity name.

```
SELECT Id,DeveloperName,EventChannel,SelectedEntity FROM PlatformEventChannelMember
    WHERE EventChannel='Channel_ID' AND SelectedEntity='EntityChangeEvent'
```

   For example, for AccountChangeEvent on custom channel ID `0YLRM00000000434AA`, the query looks as follows.

```
SELECT Id,DeveloperName,EventChannel,SelectedEntity FROM PlatformEventChannelMember
WHERE EventChannel='0YLRM00000000434AA' AND SelectedEntity='AccountChangeEvent'
```

   Or for the standard ChangeEvents channel, the full URI would be:

```
SELECT Id,DeveloperName,EventChannel,SelectedEntity FROM PlatformEventChannelMember
    WHERE EventChannel='ChangeEvents' AND SelectedEntity='AccountChangeEvent'
```

3. Make a PATCH request to this URI and append the channel member ID you got in the previous step.

```
/services/data/v52.0/tooling/sobjects/PlatformEventChannelMember/Channel_Member_ID
```

   In the request body, include the JSON definition of the channel member. For example, to update AccountChangeEvent on the channel member ID of `0v8RM00000000cTYAQ` and set the enriched fields to be the `Phone` field only, make a PATCH request to this URI:

```
/services/data/v52.0/tooling/sobjects/PlatformEventChannelMember/0v8RM00000000cTYAQ
```

   With this request body:

```
{
  "FullName": "SalesEvents_chn_AccountChangeEvent",
  "Metadata": {
    "enrichedFields": [
      {
        "name": "Phone"
      }
    ],
    "eventChannel": "SalesEvents__chn",
    "selectedEntity": "AccountChangeEvent"
  }
}
```

   If the channel member was previously configured with enriched fields, the update clears them and replaces them with the fields specified in the request body. This example specifies only one enriched field, the `Phone` field. If the channel member didn't contain enriched fields, the update adds the specified enriched fields.

   For PATCH requests, include the full definition of a PlatformEventChannelMember. Partial definitions with only the enriched fields aren't supported.

SEE ALSO:

Tooling API Developer Guide: PlatformEventChannel

Tooling API Developer Guide: PlatformEventChannelMember

Example: Add Event Enrichment Fields with Metadata API

To add event enrichment fields, use the PlatformEventChannelMember metadata type, and specify the fields, the channel, and channel member.

If the channel member you are enriching is part of a custom channel, create the custom channel first, as shown in this example. You can skip this step if using the ChangeEvents standard channel, or if you created the custom channel earlier.

This sample metadata component is for a custom channel.

```
<?xml version="1.0" encoding="UTF-8"?>
<PlatformEventChannel xmlns="http://soap.sforce.com/2006/04/metadata">
    <channelType>data</channelType>
    <label>Custom Channel for Sales Events</label>
</PlatformEventChannel>
```

This `package.xml` references the previous definition. The custom channel name is `SalesEvents__chn`.

```
<?xml version="1.0" encoding="UTF-8"?>
<Package xmlns="http://soap.sforce.com/2006/04/metadata">
    <types>
        <members>SalesEvents__chn</members>
        <name>PlatformEventChannel</name>
    </types>
    <version>52.0</version>
</Package>
```

To add enrichment fields, deploy the PlatformEventChannelMember component containing the enriched fields. In this example, the component contains two enriched fields for AccountChangeEvent on the SalesEvents custom channel. `External_Account_ID__c` is a custom field defined in your org.

```
<?xml version="1.0" encoding="UTF-8"?>
<PlatformEventChannelMember xmlns="http://soap.sforce.com/2006/04/metadata">
    <enrichedFields>
        <name>External_Account_ID__c</name>
    </enrichedFields>
    <enrichedFields>
        <name>Industry</name>
    </enrichedFields>
    <eventChannel>SalesEvents__chn</eventChannel>
    <selectedEntity>AccountChangeEvent</selectedEntity>
</PlatformEventChannelMember>
```

Use this `package.xml` manifest file to deploy or retrieve the channel member definition. Only API version 51.0 or later is supported for enriched fields.

```
<?xml version="1.0" encoding="UTF-8"?>
<Package xmlns="http://soap.sforce.com/2006/04/metadata">
    <types>
        <members>SalesEvents_AccountChangeEvent</members>
        <name>PlatformEventChannelMember</name>
    </types>
    <version>52.0</version>
</Package>
```

Note: If the member is already part of the channel, you can update it with enriched fields by redeploying the PlatformEventChannelMember component with an updated definition.

SEE ALSO:

Metadata API Developer Guide: PlatformEventChannel

Metadata API Developer Guide: PlatformEventChannelMember

Example: Delivered Enriched Event Messages

Check out example event messages that contain enriched fields for update and delete operations.

This change event for an account update includes two enriched fields, `Industry` and the `External_Account_ID__c` custom field. The `changedFields` field indicates which fields changed. In this example, only the `Fax` field and the `LastModifiedDate` system field changed.

```
{
  "schema": "I8b-dYxvxs5wOtCBr4qsew",
  "payload": {
    "LastModifiedDate": "2020-12-15T23:09:18.000Z",
    "Industry": "Biotechnology",
    "External_Account_ID__c": "1ABC",
    "Fax": "4155551212",
    "ChangeEventHeader": {
      "commitNumber": 74487681550,
      "commitUser": "005RM000001hv4SYAQ",
      "sequenceNumber": 1,
      "entityName": "Account",
      "changeType": "UPDATE",
      "changedFields": [
        "Fax",
        "LastModifiedDate"
      ],
      "changeOrigin": "com/salesforce/api/soap/51.0;client=SfdcInternalAPI/",
      "transactionKey": "0005e192-9e62-2554-3880-3c3718bd6308",
      "commitTimestamp": 1608073758000,
      "recordIds": [
        "001RM000004UF9sYAG"
      ]
    }
  },
  "event": {
    "replayId": 12
  }
}
```

A change event message for a delete operation includes the non-empty enriched fields, `Industry` and `External_Account_ID__c`, but no other record fields.

```
{
  "schema": "I8b-dYxvxs5wOtCBr4qsew",
  "payload": {
    "Industry": "Biotechnology",
    "External_Account_ID__c": "1ABC",
```

```
    "ChangeEventHeader": {
      "commitNumber": 74488073004,
      "commitUser": "005RM000001hv4SYAQ",
      "sequenceNumber": 1,
      "entityName": "Account",
      "changeType": "DELETE",
      "changedFields": [],
      "changeOrigin": "com/salesforce/api/soap/51.0;client=SfdcInternalAPI/",
      "transactionKey": "0005e1af-70bd-cce1-59c2-00f45864bbfb",
      "commitTimestamp": 1608073975000,
      "recordIds": [
        "001RM000004UF9sYAG"
      ]
    }
  },
  "event": {
    "replayId": 13
  }
}
```


Event Enrichment Considerations

Keep in mind these considerations when using enriched change events.

No Apex Trigger Support

Event enrichment isn't available in Apex change event triggers.

Latest Enriched Field Value Returned When Replaying an Event Message

Change event messages are stored in the event bus for a temporary duration. Enriched fields aren't stored with the event message in the event bus. When you retrieve an event from the event bus using a replay option, enriched fields are retrieved from the database and added to the event message before delivery. As a result, if an enriched field is updated after the event was stored, the replayed event message contains the latest value and not the original value for the enriched field. The only exception is when the enriched fields are part of the changed fields for the event. In this case, their values reflect the correct changed values.

Duplicate Replay ID for Ungrouped Enriched Event Messages

Salesforce sometimes groups event messages when the same change occurs in multiple records of one object during a transaction. For more information, see the recordIds field in Change Event Header Fields. However, if change event messages are enriched, single event messages are sent because the external ID field values could be different for each record. Because these event messages are first grouped and then ungrouped, they contain duplicate ReplayId values and only one record ID in the recordIds field. You can still replay those events using the ReplayId option in Streaming API (CometD). Also, because change event messages aren't optimistically grouped before being delivered to subscribers, event allocation usage could be higher.

CampaignMember Change Event

If a CampaignMember is deleted from a campaign, the change event message doesn't include enriched fields because it's a hard deletion—the record no longer exists in the database. The system can't query the enriched field value for that record. However, if a CampaignMember is deleted as part of a cascade delete on the campaign, this deletion is a soft deletion, and the records are in the Recycle Bin. The system can query the soft-deleted record and obtain the enriched fields.

Gap and Overflow Events

Enriched fields aren't supported for gap or overflow events.

Subscribe with Apex Triggers

With Apex triggers, you can capture and process change events on the Lightning Platform. Change event triggers run asynchronously after the database transaction is completed. Perform resource-intensive business logic asynchronously in the change event trigger, and implement transaction-based logic in the Apex object trigger. By decoupling the processing of changes, change event triggers can help reduce transaction processing time.

IN THIS SECTION:

Change Event Triggers

Because change events are based on platform events, they share characteristics for subscription with Apex platform event triggers. Also, change event messages in triggers contain header and record fields, and some additional fields not present in JSON event messages.

Apex Trigger Quick Start

Create an Apex trigger that captures change event messages.

Apex Trigger Example

This sample trigger demonstrates a common use for change event triggers and provides a more complex trigger example than the quick start.

Test Change Event Triggers

Before you can package or deploy Apex change event triggers to production, you must provide Apex tests and sufficient code coverage.

Change Event Trigger Considerations

Keep these considerations in mind when working with change events in Apex triggers.

Obtain Apex Trigger Subscribers

To get information about the triggers that are subscribed to change events, query the EventBusSubscriber standard object using SOQL. EventBusSubscriber contains information about Apex triggers but not CometD subscribers.

Change Event Triggers

Because change events are based on platform events, they share characteristics for subscription with Apex platform event triggers. Also, change event messages in triggers contain header and record fields, and some additional fields not present in JSON event messages.

A change event trigger:

- Is an after-insert trigger (defined with the `after insert` keyword)? It fires after the change event message is published.

  For example, this empty trigger definition is for the Account standard object.

  ```
  trigger AccountChangeEventTrigger on AccountChangeEvent (after insert) {

  }
  ```

  And this empty trigger definition is for the Employee__c custom object.

  ```
  trigger EmployeeChangeEventTrigger on Employee__ChangeEvent (after insert) {

  }
  ```

- Runs under the Automated Process entity. As a result:

  - Debug logs corresponding to the trigger execution are created by Automated Process. The logs aren't available in the Developer Console's log tab, except for Apex tests. Set up a trace flag entity for the Automated Process entity on the Debug Logs page in Setup.
  - The system fields of records that the trigger processes, such as CreatedById and LastModifiedById, also reference the Automated Process entity.

- Executes asynchronously outside the Apex transaction that published the change event.
- Is subject to Apex synchronous governor limits.
- Has a maximum batch size of 2,000 event messages.

To override a change event trigger's default running user and batch size, use PlatformEventSubscriberConfig in Tooling API or Metadata API. PlatformEventSubscriberConfig also configures platform event triggers. For more information, see Configure the User and Batch Size for Your Platform Event Trigger in the *Platform Events Developer Guide*.

Apex Change Event Message Fields

Each change event captured in the trigger contains header and record fields. Fields in a change event message are statically defined, just like in any other Apex type. As a result, all record fields are present in the change event message, whether changed or not. Unchanged fields are null in the Apex change event message. For details, see Change Event Body Fields.

To obtain a header field, access the `ChangeEventHeader` field on the event object. For example, this code snippet gets the change event header and writes two header field values to the debug log.

```
EventBus.ChangeEventHeader header = event.ChangeEventHeader;
String changeEntity = header.entityName;
String changeOperation = header.changeType;
```

All header fields are provided in the `EventBus.ChangeEventHeader` Apex class. For more information, see ChangeEventHeader Class in the Apex Developer Guide. The Apex class also contains these two headers, which aren't present in JSON event messages.

`nulledfields`
Contains the names of fields whose values were changed to null in an update operation. Use this field in Apex change event messages to determine if a field was changed to null in an update and isn't an unchanged field.

> Note: Starting in API version 47.0, the `changedfields` header is present in Apex and JSON event messages. It contains all fields that were changed in an update operation, whether populated or set to null.

`difffields`
Contains the names of fields whose values are sent as a unified diff because they contain large text values.

SEE ALSO:

Other Types of Change Events: Gap and Overflow Events

Apex Trigger Quick Start

Create an Apex trigger that captures change event messages.

IN THIS SECTION:

Prerequisites
Before subscribing to change events with an Apex trigger, set up debug logs and select the Account object for notifications.

Add an Apex Trigger
The quick start adds a simple change event trigger that shows how to access header and record fields in a change event message.

Prerequisites

Before subscribing to change events with an Apex trigger, set up debug logs and select the Account object for notifications.

Set Up Debug Logs

To obtain debug logs for change event trigger execution, set up debug logs for the Automated Process entity.

1. From Setup, enter *Debug Logs* in the Quick Find box, then select **Debug Logs**.
2. Click **New**.
3. For Traced Entity Type, select **Automated Process**.
4. Select the time period to collect logs for and the debug level.
5. Click **Save**.

Enable Change Notifications for the Object

On the Change Data Capture page in Setup, enable change notifications for the Account object. See Select Objects for Change Notifications in the User Interface.

Add an Apex Trigger

The quick start adds a simple change event trigger that shows how to access header and record fields in a change event message.

Before you add and test the trigger, set up debug logging for the Automated Process entity and enable Account for Change Data Capture. See Prerequisites.

1. In the Developer Console, select **File** > **New** > **Apex Trigger**.
2. In the Name field, enter a name for the trigger: *MyAccountChangeTrigger*.
3. From the dropdown, select the change event object for Account: **AccountChangeEvent**.
   The trigger is created with the `after insert` keyword.
4. Replace the default content of the trigger with the following code.

```
trigger MyAccountChangeTrigger on AccountChangeEvent (after insert) {
    List<Task> tasks = new List<Task>();

    // Iterate through each event message.
    for (AccountChangeEvent event : Trigger.New) {
        // Get some event header fields
        EventBus.ChangeEventHeader header = event.ChangeEventHeader;
        System.debug('Received change event for ' + header.entityName +
                     ' for the ' + header.changeType + ' operation.');

        // Get account record fields
        System.debug('Account Name: ' + event.Name);
        System.debug('Account Phone: ' + event.Phone);

        // Create a followup task
        if (header.changetype == 'CREATE') {
```

```
            Task tk = new Task();
            tk.Subject = 'Follow up on new account for record or group of records: ' +
              header.recordIds;
            // Explicitly set the task owner ID to a valid user ID so that
            // it is not Automated Process.
            // For simplicity, we set it to the CommitUser header field,
            // which is available for all operations.
            tk.OwnerId = header.CommitUser;
            tasks.add(tk);
        }
        else if ((header.changetype == 'UPDATE')) {
            // For update operations, iterate over the list of changed fields
            System.debug('Iterate over the list of changed fields.');
            for (String field : header.changedFields) {
                if (null == event.get(field)) {
                    System.debug('Deleted field value (set to null): ' + field);
                } else {
                    System.debug('Changed field value: ' + field +
                        '. New Value: ' + event.get(field));
                }
            }
        }
    }

    // Insert all tasks in bulk.
    if (tasks.size() > 0) {
        insert tasks;
    }

}
```

This simple trigger writes header and field values to the debug log for each received change event message. The trigger uses the `changedFields` header field to determine which fields changed in an update operation. The trigger also creates a follow-up task for new accounts.

Note: The `changedFields` property is available in Apex saved using API version 47.0 or later.

5. To test the trigger, create an account with a name and phone.
6. Edit the account, change the name, delete the phone value, and save the record.
7. In Setup, enter *Debug Logs* in the Quick Find box, then select **Debug Logs**.
8. To view the debug logs corresponding to the record creation, click the second log in the list (logs are ordered by most recent first). The output of the `System.debug` statements looks similar to the following.

```
...|DEBUG|Received change event for Account for the CREATE operation.
...|DEBUG|Account Name: Quick Start Account
...|DEBUG|Account Phone: 4155551212
```

9. To view the debug logs corresponding to the record update, click the first log in the list. The output of the `System.debug` statements looks similar to the following. The account name's new value is listed. The phone field's value was deleted, and its value

was set to null. Also, because the system updates the `LastModifiedDate` field when the record is updated, this field is listed in the `changedFields` field and is part of the debug output.

```
...|DEBUG|Received change event for Account for the UPDATE operation.
...|DEBUG|Account Name: Quick Start Account Updated
...|DEBUG|Account Phone: null
...|DEBUG|Iterate over the list of changed fields.
...|DEBUG|Changed field value: Name. New Value: Quick Start Account Updated
...|DEBUG|Deleted field value (set to null): Phone
...|DEBUG|Changed field value: LastModifiedDate. New Value: 2019-08-14 23:20:15
```

Apex Trigger Example

This sample trigger demonstrates a common use for change event triggers and provides a more complex trigger example than the quick start.

Using an Apex Change Event Trigger to Predict Account Status

Because Apex change event triggers run asynchronously, they typically contain time-intensive processes that run after the database transaction is completed. This trigger example covers a more common real-world scenario than the trigger given in the quick start. It captures case changes and predicts the account trust status using an Apex class. The prediction is based on counting the account cases and checking case fields. In other scenarios, you might have complex prediction algorithms that are more resource intensive.

The trigger calls a method in an Apex class to perform the trust prediction on the accounts related to all cases in this trigger. An account status of Red means that the account trust level is low because too many cases are associated with this account among other criteria. An account status of Green means that the trust level is good. The associated class is listed after this trigger. This example assumes that you have the following prerequisites.

- A platform event named Red_Account__e with two fields: Account_Id__c of type Text and Rating__c of type Text
- The SLAViolation__c custom field on the Case object of type Picklist with values Yes and No

```
trigger CaseChangeEventTrigger on CaseChangeEvent (after insert) {

    List<CaseChangeEvent> changes = Trigger.new;

    Set<String> caseIds = new Set<String>();

    for (CaseChangeEvent change : changes) {
        // Get all Record Ids for this change and add to the set
        List<String> recordIds = change.ChangeEventHeader.getRecordIds();
        caseIds.addAll(recordIds);
    }

    // Perform heavy (slow) computation determining Red Account
    // status based on these Case changes
    RedAccountPredictor predictor = new RedAccountPredictor();
    Map<String, boolean> accountsToRedAccountStatus =
        predictor.predictForCases(new List<String>(caseIds));

    // Publish platform events for predicted red accounts
    List<Red_Account__e> redAccountEvents = new List<Red_Account__e>();
    for (String acctId : accountsToRedAccountStatus.keySet()) {
        String rating = accountsToRedAccountStatus.get(acctId) ? 'Red' : 'Green';
```

```
        if (rating=='Red') {
         redAccountEvents.add(new Red_Account__e(Account_Id__c=acctId,
             Rating__c=rating));
       }
    }
    System.debug('RED_ACCT: ' + redAccountEvents);
    if (redAccountEvents.size() > 0) {
        EventBus.publish(redAccountEvents);
    }
}
```

The `RedAccountPredictor` class performs the prediction of the account trust level. The first method that the trigger calls is `predictForCases`, which calls other methods in this class. The method returns a map of account ID to a Boolean value for account status.

```
public class RedAccountPredictor {

    private static final Integer MAX_CASES_EXPECTED = 2;

    public RedAccountPredictor() { }

    // First method to be called for performing account status prediction.
    // Get the account IDs related to the passed-in case IDs
    // and call a predictor method.
    // Return a map of account ID to account status Boolean.
    public Map<String, boolean> predictForCases(List<String> caseId) {
        List<Case> casesMatchingIds =
            [SELECT Id, Account.Id FROM Case WHERE Id IN :caseId];
        if (null != casesMatchingIds && casesMatchingIds.size() > 0) {
            List<String> accountIds = new List<String>();
            for (Case c : casesMatchingIds) {
                accountIds.add(c.Account.Id);
            }
            return predictForAccounts(accountIds);
        } else {
            return new Map<String, boolean>();
        }
    }

    // Perform slow, resource intensive calcuation to determine.
    // If Account is in RED status (think Einsein predictions, etc.)
    public Map<String, boolean> predictForAccounts(List<String> acctIds) {
        List<Case> casesForAccounts =
            [SELECT Id, Account.Id, Status, CaseNumber, Priority,
                 IsEscalated, SLAViolation__c
             FROM Case
             WHERE AccountId IN :acctIds AND Status !='Closed'];
        Map<String, List<Case>> accountsToCases = new Map<String, List<Case>>();
        for (Case c : casesForAccounts) {
            if (null == c.Account.Id) continue;
            if (!accountsToCases.containsKey(c.Account.Id)) {
                accountsToCases.put(c.Account.Id, new List<Case>());
            }
            accountsToCases.get(c.Account.Id).add(c);
        }
```

```
        Map<String, boolean> results = new Map<String, boolean>();
        for (String acctId : accountsToCases.keySet()) {
            results.put(acctId, predict(accountsToCases.get(acctId)));
        }
        return results;
    }

    // Perform the account status prediction.
    // Return true if account is red; otherwise, return false.
    private boolean predict(List<Case> casesForAccount) {
        boolean isEscalated = false;
        boolean hasSlaViolation = false;
        boolean hasHighPiority = false;
        boolean allStatusesResolved = true;

        for (Case openCase : casesForAccount) {
            isEscalated |= openCase.IsEscalated;
            hasSlaViolation |= (openCase.SLAViolation__c == 'Yes');
            hasHighPiority |= openCase.Priority == 'High';
            allStatusesResolved &= (openCase.Status == 'Closed'
                                    || openCase.Status == 'Part Received');
        }
        if (allStatusesResolved) {
            return false;
        }
        if (casesForAccount.size() > MAX_CASES_EXPECTED) {
            return true;
        } else if (isEscalated || hasSlaViolation) {
            return true;
        } else if (hasHighPiority) {
            return true;
        }
        return false;
    }
}
```

Test Change Event Triggers

Before you can package or deploy Apex change event triggers to production, you must provide Apex tests and sufficient code coverage.

Enable All Change Data Capture Entities for Notifications

To enable the generation of change event notifications for all supported Change Data Capture entities for an Apex test, call this method.

```
Test.enableChangeDataCapture();
```

Call the `Test.enableChangeDataCapture()` method at the beginning of your test method before performing DML operations and calling `Test.getEventBus().deliver()` or `Test.stopTest()`.

The `Test.enableChangeDataCapture()` method ensures that Apex tests can fire change event triggers regardless of the entities selected in Setup. This method doesn't affect the Change Data Capture entity selections for the org.

Deliver Test Change Events

To test your change event trigger, perform DML operations and then call the `Test.getEventBus().deliver()` method. The method delivers the event messages from the test event bus to the corresponding change event trigger and causes the trigger to fire. Finally, validate that the trigger executed as expected. For example, if the trigger creates or updates records, you can query those records with SOQL.

This test method outlines the order of statements that must be executed in a test, starting with enabling Change Data Capture entities.

```
@isTest static void testChangeEventTrigger() {
    // Enable all Change Data Capture entities for notifications.
    Test.enableChangeDataCapture();

    // Insert one or more test records
    // ...

    // Deliver test change events
    Test.getEventBus().deliver();

    // Verify the change event trigger's execution
    // ...
}
```

Alternatively, use the `Test.startTest()`, `Test.stopTest()` method block to fire a change event trigger. After `Test.stopTest()` executes, all test change event messages generated from DML operations are delivered to the associated trigger. The DML statements can be within the block or outside the block as long as they precede `Test.stopTest()`.

```
@isTest static void testChangeEventTriggerWithStopTest() {
    // Enable all Change Data Capture entities for notifications.
    Test.enableChangeDataCapture();

    Test.startTest();
    // Insert one or more test records
    // ...
    Test.stopTest();
    // The stopTest() call delivers the test change events and fires the trigger

    // Verify the change event trigger's execution
    // ...
}
```

Note: In test context, only up to 500 change event messages can be delivered as a result of record changes. If you exceed this limit, the Apex test stops execution with a fatal error.

Apex Test Example Based on Quick Start Trigger

The `testNewAccount` method in this test class shows you how to write a test for the `MyAccountChangeTrigger` trigger provided in the Add an Apex Trigger on page 33 quick start. The test method first enables all entities for change notifications. It creates a test account and then calls the `Test.getEventBus().deliver();` method. Next, the test verifies that the trigger's execution by querying Task records and validating that one task was created. The query returns only tasks that the trigger created in test contest. For that reason, the test expects only one task. Next, the test updates the account and verifies that no new task is created.

```
@isTest
public class TestMyAccountChangeTrigger {
    @isTest static void testNewAndUpdatedAccount() {
```

```
        // Enable all Change Data Capture entities for notifications.
        Test.enableChangeDataCapture();

        // Insert an account to generate a change event.
        Account newAcct = new Account(Name='TestAccount', Phone='4155551212');
        insert newAcct;

        // Call deliver to fire the trigger and deliver the test change event.
        Test.getEventBus().deliver();

        // VERIFICATIONS
        // Check that the change event trigger created a task.
        Task[] taskList = [SELECT Id,Subject FROM Task];
        System.assertEquals(1, taskList.size(),
            'The change event trigger did not create the expected task.');

        // Update account record
       Account queriedAcct = [SELECT Id,Phone,Website FROM Account WHERE Id=:newAcct.Id];

        // Debug
        System.debug('Retrieved account record: ' + queriedAcct);
        // Update one field and empty another
        queriedAcct.Website = 'www.salesforce.com';
        queriedAcct.Phone = null;
        update queriedAcct;

        // Call deliver to fire the trigger for the update operation
        Test.getEventBus().deliver();

        // VERIFICATIONS
        // Check that the change event trigger did NOT create a task.
        // We should still have only 1 task.
        Task[] taskList2 = [SELECT Id,Subject FROM Task];
        System.assertEquals(1, taskList2.size(),
            'The change event trigger created a task unepextedly.');
    }
}
```

This test class is an alternative example that uses the `Test.startTest()`, `Test.stopTest()` method block to deliver test change events and fire the trigger. For more information about these methods, see Using Limits, `startTest`, and `stopTest` in the *Apex Developer Guide*.

```
@isTest
public class TestMyAccountChangeTriggerWithStopTest {
  @isTest static void testNewAccount() {
    // Enable all Change Data Capture entities for notifications.
    Test.enableChangeDataCapture();

    Test.startTest();
    // Insert an account to generate a change event.
    insert new Account(Name='TestAccount', Phone='4155551212');
    Test.stopTest();
    // The stopTest() call fires the trigger with the test account change event.

    // VERIFICATIONS
```

```
    // Check that the change event trigger created a task.
    Task[] taskList = [SELECT Id,Subject FROM Task];
    System.assertEquals(1, taskList.size(),
        'The change event trigger did not create the expected task.');
  }
}
```

Note: If multiple DML operations are performed on a single record within the `Test.startTest()`, `Test.stopTest()` block, only one change event is generated. The change event contains the latest data and the initial change type. For more information, see Change Event Generation in a Transaction with Multiple Changes for the Same Record.

Properties of Change Events in Test Context

Test change events messages are published to the test event bus, which is separate from the Salesforce event bus. They aren't persisted in Salesforce and aren't delivered to event channels outside the test class. Properties of test change event messages, like the replay ID, are reset in test context and reflect only the values of test event messages. For more information, see Event and Event Bus Properties in Test Context in the Platform Events Developer Guide.

Change Event Trigger Considerations

Keep these considerations in mind when working with change events in Apex triggers.

Triggers for Non-Enabled Objects

You can save an Apex trigger for a change event object even if the object isn't selected for notifications on the Change Data Capture page. When the object isn't selected, the trigger doesn't fire. To ensure that the trigger fires, select the object for notifications. Any type of change event fires a change event trigger, including gap events and overflow events.

No Formula Field Support

Formula fields aren't supported in Change Data Capture. They're null in a change event trigger, regardless of whether they were changed. For information on which field values aren't included, so are null in a trigger, see Change Event Fields.

Null Name Field for Person Accounts

For a person account, the Name compound field in the AccountChangeEvent received in the trigger is null. The FirstName and LastName fields, which are included in the Name field, contain the person account first name and last name values. In contrast, the ContactChangeEvent Name field contains the concatenated values of the salutation, first name, and last name.

Infinite Trigger Loop

If your trigger updates records of the same object as the one that corresponds to the received change event, then the trigger can fire recursively and exceed limits. To avoid infinite trigger recursion, ensure that you limit your updates so they don't occur every time the trigger refires.

Obtain Apex Trigger Subscribers

To get information about the triggers that are subscribed to change events, query the EventBusSubscriber standard object using SOQL. EventBusSubscriber contains information about Apex triggers but not CometD subscribers.

Example: This example SOQL query selects several fields from EventBusSubscriber. For more information about EventBusSubscriber fields, see EventBusSubscriber in the *Object Reference for Salesforce and Lightning Platform*.

```
SELECT ExternalId, Name, Topic, Position, Status, Tip, Type FROM EventBusSubscriber
```

The returned result shows that there are two Apex triggers subscribed to change events. One trigger is subscribed to AccountChangeEvent and one to ContactChangeEvent.

ExternalId	Name	Topic	Position	Status	Tip	Type
01q2J000000g0kb	MyAccountChangeTrigger	AccountChangeEvent	226751	Running	-1	ApexTrigger
01q2J000000g0kg	MyContactChangeTrigger	ContactChangeEvent	226752	Running	-1	ApexTrigger

You can filter the query results by using a WHERE clause. For example, this query filters by the topic ContactChangeEvent.

```
SELECT ExternalId, Name, Topic, Position, Status, Tip, Type FROM EventBusSubscriber
WHERE Topic='ContactChangeEvent'
```

The query returns only the trigger subscribers to ContactChangeEvent, in this case, one trigger.

ExternalId	Name	Topic	Position	Status	Tip	Type
01q2J000000g0kg	MyContactChangeTrigger	ContactChangeEvent	226752	Running	-1	ApexTrigger

Monitor Change Event Publishing and Delivery Usage

To get usage data for event publishing and CometD-client delivery, query the PlatformEventUsageMetric object. Usage data is available for the last 24 hours, ending at the last hour, and for historical daily usage. PlatformEventUsageMetric is available in API version 50.0 and later.

Use PlatformEventUsageMetric to get visibility into your event usage and usage trends. The usage data gives you an idea of how close you are to your allocations and when you need more allocations. The usage metrics stored in PlatformEventUsageMetric are separate from the REST API limits values. Use the REST API limits to track your monthly delivery usage against your allocations. The monthly CometD-client delivery usage that the limits API returns is common for platform events and change data capture events. PlatformEventUsageMetric breaks down usage of platform events and change data capture events so you can track their usage separately.

Because dates are stored in Coordinated Universal Time (UTC), convert your local dates and times to UTC for the query. For the date format to use, see Date Formats and Date Literals in the *SOQL and SOSL Reference*.

Note:

- Usage data is stored for at least 45 days. Usage data is updated hourly and is available only when usage is nonzero for a 24-hour period. Usage data isn't available for 1-hour intervals or any other arbitrary interval. The only supported intervals are the last 24 hours and daily data. Also, usage data isn't available for standard-volume platform events.
- After a Salesforce major upgrade, usage data can be inaccurate for the day and the last 24 hours within the upgrade window. New usage data overwrites the data for the hour that the 5-minute upgrade occurs in. The new usage data includes metrics that start after the upgrade for that hour. For more information about Salesforce upgrades, see Salesforce Upgrades and Maintenance in *Help* and Salesforce Status.

For change events, you can query usage data for these metrics. The first value is the metric name value that you supply in the query.

- `CHANGE_EVENTS_PUBLISHED`—Number of change data capture events published
- `CHANGE_EVENTS_DELIVERED`—Number of change data capture events delivered to CometD clients

For platform events, you can query usage data for these metrics. The first value is the metric name value that you supply in the query.

- `PLATFORM_EVENTS_PUBLISHED`—Number of platform events published
- `PLATFORM_EVENTS_DELIVERED`—Number of platform events delivered to CometD clients

Obtain Usage Metrics for the Last 24 Hours

To get usage metrics for the last 24 hours, ending at the last hour, perform a query by specifying the start and end date and time in UTC, and the metric name.

For the last 24-hour period, the end date is the current date in UTC, with the time rounded down to the previous hour. The start date is 24 hours before the end date. Dates have hourly granularity.

Example: Based on the current date and time of August 4, 2020 11:23 in UTC, the last hour is 11:00. The query includes these dates.

- Start date in UTC format: 2020-08-03T11:00:00.000Z
- End date in UTC format: 2020-08-04T11:00:00.000Z

This query returns the usage for the number of platform events delivered to CometD clients for August 3, 2020 at 11:00 to August 4, 2020 at 11:00.

```
SELECT Name, StartDate, EndDate, Value FROM PlatformEventUsageMetric
WHERE Name='CHANGE_EVENTS_DELIVERED'
AND StartDate=2020-08-03T11:00:00.000Z AND EndDate=2020-08-04T11:00:00.000Z
```

The query returns this result for the last 24-hour usage.

Name	StartDate	EndDate	Value
CHANGE_EVENTS_DELIVERED	2020-08-03T11:00:00.000+0000	2020-08-04T11:00:00.000+0000	575

The time span between StartDate and EndDate is 24 hours for the stored 24-hour usage. Therefore, you can specify either StartDate or EndDate in the query and you get the same result.

Obtain Historical Daily Usage Metrics

To get daily usage metrics for one or more days, perform a query by specifying the start date and end date in UTC, and metric name.

Example: To get usage metrics for a period of 3 days, from July 19 to July 22, 2020, use these start and end dates. Time values are 0.

- Start date for the query: 2020-07-19T00:00:00.000Z
- End date for the query: 2020-07-22T00:00:00.000Z

This query selects usage metrics for the number of platform events delivered to CometD clients for a 3-day period.

```
SELECT Name, StartDate, EndDate, Value FROM PlatformEventUsageMetric
WHERE Name='CHANGE_EVENTS_DELIVERED'
AND StartDate>=2020-07-19T00:00:00.000Z and EndDate<=2020-07-22T00:00:00.000Z
```

The query returns these results for the specified date range.

Name	StartDate	EndDate	Value
CHANGE_EVENTS_DELIVERED	2020-07-19T00:00:00.000+0000	2020-07-20T00:00:00.000+0000	575
CHANGE_EVENTS_DELIVERED	2020-07-20T00:00:00.000+0000	2020-07-21T00:00:00.000+0000	899

Name	StartDate	EndDate	Value
CHANGE_EVENTS_DELIVERED	2020-07-21T00:00:00.000+0000	2020-07-22T00:00:00.000+0000	1,035

SEE ALSO:

Object Reference for Salesforce and Lightning Platform: PlatformEventUsageMetric

Security Considerations

Learn about the user permissions required for subscription, field-level security, and Shield Platform Encryption.

IN THIS SECTION:

Required Permissions for Change Events Received by CometD Subscribers

Change Data Capture ignores sharing settings and sends change events for all records of a Salesforce object. To receive change events on a channel, the subscribed user must have one or more permissions depending on the entities associated with the change events.

Field-Level Security

Change Data Capture respects your org's field-level security settings. Delivered events contain only the fields that a subscribed user is allowed to view. Before delivering a change event for an object, the subscribed user's field permissions are checked. If a subscribed user has no access to a field, the field isn't included in the change event message that the subscriber receives.

Change Events for Encrypted Salesforce Data

If Salesforce record fields are encrypted with Shield Platform Encryption, changes in encrypted field values generate change events. Change events are stored in the event bus for up to three days. To ensure that the events stored in the event bus are encrypted and not in clear text, create an event bus tenant secret and enable encryption.

Required Permissions for Change Events Received by CometD Subscribers

Change Data Capture ignores sharing settings and sends change events for all records of a Salesforce object. To receive change events on a channel, the subscribed user must have one or more permissions depending on the entities associated with the change events.

The permissions apply to CometD subscribers. Apex triggers run with system privileges under the Automated Process entity, so they don't require those permissions.

Change Event Permissions

To receive change events for	Required permission
A specific standard or custom object:	View All for the object
User:	View All Users
Standard objects that don't have the View All permission, such as Task and Event:	View All Data
All entities on a channel:	View All Data (AND View All Users, if User is one of the entities)

Permission Enforcement

For the standard `/data/ChangeEvents` channel and custom channels, user permissions are enforced on event delivery. Users can subscribe to the `/data/ChangeEvents` channel or to any custom channel regardless of their entity permissions. Users receive only change events associated with entities for which they have the necessary permissions and don't receive change events they don't have permissions for. If permissions change after subscription, the changes are enforced immediately and don't require resubscription.

For the single-entity standard channels, which include change events for one standard or custom object, user permissions are enforced initially on subscription. If users don't have sufficient permissions for the corresponding object, the subscription is denied and an error is returned. If permissions change after successful subscription and users no longer have access to the entity, they stop receiving the corresponding change events.

For more information about user permissions, see View All and Modify All Permissions Overview in *Salesforce Help*.

SEE ALSO:

Subscription Channels

Field-Level Security

Change Data Capture respects your org's field-level security settings. Delivered events contain only the fields that a subscribed user is allowed to view. Before delivering a change event for an object, the subscribed user's field permissions are checked. If a subscribed user has no access to a field, the field isn't included in the change event message that the subscriber receives.

When describing a change event of a Salesforce object, the describe call checks the user's field-level security settings for that object. The describe call returns only the fields that the user has access to in the describe result of the change event. You can describe a change event through SOAP API or REST API by using the change event name as the sObject name, such as AccountChangeEvent. See describeSObjects() in the SOAP API Developer Guide and sObject Describe in the REST API Developer Guide.

When getting the change event schema corresponding to a Salesforce object, the returned schema includes all object fields, even the fields that the user doesn't have access to. To get the event schema, use the eventSchema REST API resource. See Platform Event Schema by Event Name and Platform Event Schema by Schema ID in the REST API Developer Guide.

Change Events for Encrypted Salesforce Data

If Salesforce record fields are encrypted with Shield Platform Encryption, changes in encrypted field values generate change events. Change events are stored in the event bus for up to three days. To ensure that the events stored in the event bus are encrypted and not in clear text, create an event bus tenant secret and enable encryption.

To enable encryption of change events, first create an event bus tenant secret on the Key Management page in Setup. Then enable encryption of change events on the Encryption Policy page.

Warning: You must create an event bus tenant secret before enabling encryption. From Setup, the encryption setting is available only after you create an event bus tenant secret. In Metadata API, if you enable encryption using PlatformEncryptionSettings without having the tenant secret, you get an error.

IN THIS SECTION:

Generate a Tenant Secret

To enable encryption of change events, first generate an event bus tenant secret.

Enable Encryption of Change Events

After you create an event bus tenant secret, a checkbox becomes available in the Encryption Policy page that starts encryption of change events.

Capturing Changes and Encrypting the Event Payload

After capturing record changes, Change Data Capture creates a change event and stores it in the event bus. Because data changes are captured internally on the application servers in decrypted form, they must be encrypted before storing the corresponding change event that contains them. The entire event payload is encrypted using the data encryption key that is based on the Event Bus tenant secret type.

SEE ALSO:

Salesforce Help: Strengthen Your Data's Security with Shield Platform Encryption

Salesforce Help: Which User Permissions Does Shield Platform Encryption Require?

Generate a Tenant Secret

To enable encryption of change events, first generate an event bus tenant secret.

USER PERMISSIONS

To manage tenant secrets:

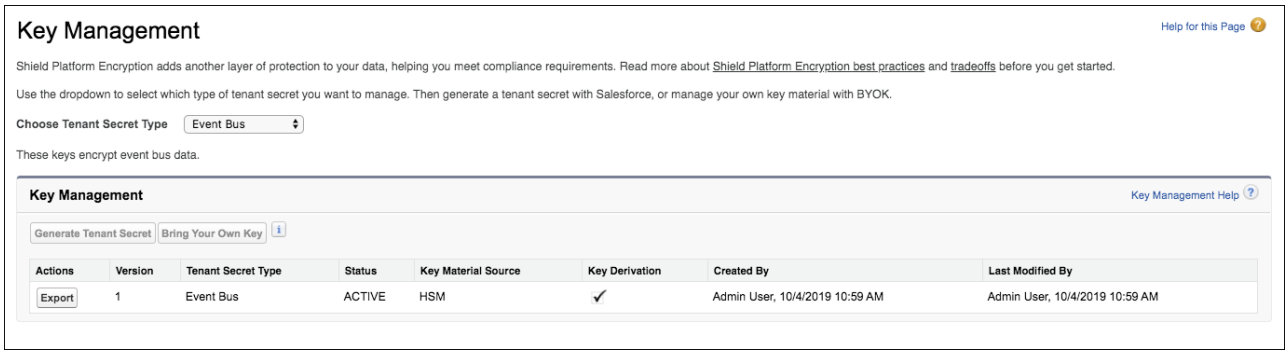
- Manage Encryption Keys

Prerequisites:

Only authorized users can generate tenant secrets from the Platform Encryption page. Ask your Salesforce admin to assign the Manage Encryption Keys permission to you.

Steps:

1. From Setup, in the Quick Find box, enter `Platform Encryption`, and then select **Key Management**.
2. In the Choose Tenant Secret Type dropdown list, choose **Event Bus**.
3. Click **Generate Tenant Secret** or, to upload a customer-supplied tenant secret, click **Bring Your Own Key**.



Note:

- If your Salesforce org has no tenant secrets, perform Step 3 before Step 2.
- You can generate or rotate an event bus tenant secret once every 7 days.
- You can also generate a tenant secret through SOAP API or REST API using the TenantSecret object and the Type field value of EventBus. For more information, see TenantSecret in the *Object Reference for Salesforce and Lightning Platform*.

SEE ALSO:

Salesforce Help: Generate a Tenant Secret with Salesforce

Enable Encryption of Change Events

After you create an event bus tenant secret, a checkbox becomes available in the Encryption Policy page that starts encryption of change events.

Prerequisites:

Create an Event Bus tenant secret

Steps:

1. From Setup, in the Quick Find box, enter *Platform Encryption*, and then select **Encryption Policy**.
2. Select **Encrypt change data capture events and platform events**.

   Note: You can access and control this setting in Metadata API, in PlatformEncryptionSettings. Ensure that the event bus tenant secret is created first before setting `enableEventBusEncryption` to true.

3. Click **Save**.

Note: When you enable encryption for change events, you also enable it for platform events. For more information, see Encrypting Platform Event Messages at Rest in the Event Bus in the *Platform Events Developer Guide*.

Capturing Changes and Encrypting the Event Payload

After capturing record changes, Change Data Capture creates a change event and stores it in the event bus. Because data changes are captured internally on the application servers in decrypted form, they must be encrypted before storing the corresponding change event that contains them. The entire event payload is encrypted using the data encryption key that is based on the Event Bus tenant secret type.

When Shield Platform Encryption is enabled, Change Data Capture encrypts the fields of all Salesforce objects that it tracks. Change Data Capture ignores the object and field selections set up for Shield Platform Encryption. Fields of all objects for which changes are tracked are encrypted before event storage, even objects not selected for Shield Platform Encryption. For example, suppose that only the Mailing Address of contacts is encrypted with Shield Platform Encryption. If data changes occur in accounts and contacts, change events for both accounts and contacts are encrypted.

Delivering Change Events

Before delivering a change event to a subscribed client, the change event payload is decrypted using the data encryption key. The change event is sent over a secure channel using HTTPS and TLS, which ensures that the data is protected and encrypted while in transit. If the encryption key was rotated and a new key is issued, stored events are not re-encrypted but they are decrypted before delivery using the archived key. If a key is destroyed, stored events can't be decrypted and aren't delivered.

Note: Classic Encryption is not supported.

Change Event Considerations

Keep in mind change event considerations and allocations when subscribing to change events.

IN THIS SECTION:

General Considerations

Change Data Capture Allocations

General Considerations

No Change Events Generated for Some Actions

The following actions don't generate change events for affected records.

- Any action related to state and country/territory picklists that you perform in Setup on the State and Country/Territory Picklists page.
- Changing the type of an opportunity stage picklist value.
- When a custom picklist field is defined on Contact in a person account org, the field is present on Account with the `__pc` suffix. Replacing or renaming a value of the custom picklist doesn't generate account change events but only contact change events for the affected records. But if the custom picklist field is defined on Account, the field isn't present on Contact and only account change events are generated, as expected.

Change Event Generation in a Transaction with Multiple Changes for the Same Record

If multiple DML operations are performed on a record within the same transaction boundary, only one change event is generated for the initial change type. The change event contains the data that is committed in Salesforce at the end of the transaction. No change events are generated for the additional operations because they're internal to the transaction. For example, a case record is created and an after-insert trigger queries this case before the transaction is committed. The trigger changes the case priority from `Medium` to `High` and performs an update operation. After the transaction is committed, one change event is generated with a `changeType` of CREATE and priority of `High`.

No Formula Field Support

Formula fields aren't supported in Change Data Capture. They aren't included in change events that are delivered to CometD clients. Formula fields are included in change events in Apex triggers, but they have a value of null regardless of whether they were changed. For information on which fields are included and excluded in a change event, see Change Event Fields.

Geolocation Compound Fields

When a geolocation compound field (of type location) is changed in a custom object, all its component fields are published in the change event whether they were changed or not. In contrast, when a geolocation field is changed in a standard object, only the changed field is published. For more information, see Geolocation Compound Field in the SOAP API Developer Guide.

Change Data Capture Allocations

Concurrent Client Allocation

Description	Performance and Unlimited Editions	Enterprise Edition	Developer Edition
Maximum number of concurrent CometD clients (subscribers) across all channels and for all event types	2,000	1,000	20

Note: The concurrent client allocation is shared with all types of events that you can subscribe to through Streaming API (CometD), including PushTopic, generic, platform events, and change events. The `empApi` Lightning component uses CometD and consumes the concurrent client allocation like any other CometD client. Each logged-in user using `empApi` counts as one concurrent client. If the user has multiple browser tabs using `empApi`, the streaming connection is shared and is counted as one client for that user. A client that exceeds the concurrent client allocation receives an error and can't subscribe. When one of the clients disconnects

and a connection is available, the new client can subscribe. For more information, see Streaming API Error Codes in the Streaming API Developer Guide.

Default Allocations

If your org has no add-on licenses, default allocations apply and can't be exceeded. Each Salesforce edition provides a default allocation for the number of events delivered to CometD clients, including the `empApi` Lightning component. This allocation doesn't apply to non-CometD subscribers, such as Apex triggers, flows, and processes. The default allocation is enforced daily to ensure fair sharing of resources in the multitenant environment and to protect the service. Default allocations aren't part of the usage-based entitlement.

The number of delivered events to CometD clients is counted per subscribed client. If you have multiple client subscribers, your usage is added across all subscribers. For example, you have an Unlimited Edition org with a default allocation of 50,000 events in a 24-hour period. Within a few hours, 20,000 event messages are delivered to two subscribed clients. So you consumed 40,000 events and are still entitled to 10,000 events within the 24-hour period.

If you exceed the default event delivery allocation, you receive an error. For more information, see Streaming API Error Codes in the Streaming API Developer Guide. Event messages that are generated after exceeding the allocation are stored in the event bus. You can retrieve stored event messages as long as they are within the retention window of 72 hours.

Table 1: Default Allocations

Description	Performance and Unlimited Editions	Enterprise Edition	Developer Edition
Maximum number of entities, including standard and custom objects, that you can select for Change Data Capture If the same entity is selected in multiple channels, it is counted once toward the allocation.	5	5	5
Maximum number of custom channels	100	100	100
Event Delivery: maximum number of delivered event notifications in the last 24 hours, shared by all CometD clients. (Applies to CometD clients and the `empApi` Lightning component only.)	50,000	25,000	10,000

Change Data Capture Add-On License and Usage-Based Entitlement

To remove the limit on the number of entities that you can select for change notifications, contact Salesforce to purchase the Change Data Capture add-on license.

If your org has the add-on license, your allocation for delivered events to CometD clients moves to a monthly entitlement model. The add-on increases the 24-hour allocation of delivered event notifications by 100,000 per day (3 million a month) as a usage-based entitlement. The entitlement gives you flexibility in how you use your allocations. The entitlement isn't as strictly enforced as the default allocation. With the entitlement, you can exceed your 24-hour event delivery allocation by a certain amount. The entitlement is reset every month after your contract start date. Entitlement usage is computed only for production orgs. It isn't available in sandbox or trial orgs. For more information, see Usage-based Entitlement Fields.

Salesforce monitors event overages based on a calendar month, starting with your contract start date. If you exceed the monthly entitlement, Salesforce contacts you to discuss your event usage needs. The entitlement used for monitoring monthly event overages is the daily allocation multiplied by 30.

Table 2: Example: Entitlement with One Change Data Capture Add-On License

Description	Performance and Unlimited Editions	Enterprise Edition
Maximum number of entities, including standard and custom objects, that you can select for Change Data Capture.	No limit	No limit
Event Delivery: entitlement for delivered event notifications, shared by all CometD clients. (Applies to CometD clients and the `empApi` Lightning component only.) You can exceed this entitlement by a certain amount before receiving an error. Salesforce uses the monthly entitlement for event overage monitoring. The monthly entitlement is returned in the `limits` REST API resource.	Last 24 hours: 150,000 (50 K included with org license + 100 K from add-on license) Monthly entitlement: 4.5 million (1.5 million included with org license + 3 million from add-on license)	Last 24 hours: 125,000 (25 K included with org license + 100 K from add-on license) Monthly entitlement: 3.75 million (0.75 million included with org license + 3 million from add-on license)

To increase the entitlement for event delivery, contact Salesforce to purchase the High-Volume Platform Events add-on.

The maximum event message size that Salesforce can publish is 1 MB. If your entity has hundreds of custom fields or many long text area fields, you could hit this limit. In this case, the change event message isn't delivered and is replaced by a gap event message. For more information, see Gap Events.

Note:

- Change events are based on platform events. The default allocations and usage-based entitlement of delivered events are shared between change events and high-volume platform events.
- Apex change event trigger subscribers don't count against the 24-hour CometD event delivery limit. The number of event messages that an Apex trigger can process depends on how long the processing takes for the trigger. The longer the processing time, the longer it takes for the trigger to reach the tip of the event stream.
- The `empApi` Lightning component is a CometD client. As a result, the event delivery allocation applies to the component and it is per channel per unique browser session.

Monitor Your Event Usage Against Your Allocations

Determine how to check event usage for your org.

Allocation	Org With Add-On License	Org Without Add-On License
Event Delivery: number of delivered event notifications to CometD clients	Check your usage in one of these ways: • With the REST API `limits` resource: usage information is returned in `MonthlyPlatformEventsUsageEntitlement` in API version 48.0 and later. This value is updated once a day. • In the user interface: From Setup, in the Quick Find box, enter *Company Information*, and then select	With the REST API `limits` resource: usage information is returned in `MonthlyPlatformEvents` in API version 47.0 and earlier. This value is updated within a few minutes after event delivery.

Allocation	Org With Add-On License	Org Without Add-On License
	Company Information. The usage is shown under the Usage-based Entitlements related list.	

For more information about the limit usage values that the limits REST resource returns, see Limits and List Organization Limits in the *REST API Developer Guide*.

Monitor 24-Hour and Daily Event Usage with PlatformEventUsageMetric

To get usage data for event publishing and CometD-client delivery, query the PlatformEventUsageMetric object. The usage metrics stored in PlatformEventUsageMetric are separate from the REST API limits values. The REST API limits resource returns the maximum and remaining allocations for platform events and change data capture events. PlatformEventUsageMetric contains actual event usage data broken down by type of event for platform events and change data capture events.

PlatformEventUsageMetric usage data is available for the last 24 hours, ending at the last hour, and for historical daily usage for the last 45 days. Use PlatformEventUsageMetric to get visibility into your usage trends.

For more information, see Monitor Change Event Publishing and Delivery Usage on page 41.

Monitor Hourly Event Delivery Usage with REST API

To monitor your org's event delivery hourly usage, make a REST API call to the limits resource every hour. The difference between the results obtained in the last 2 hours shows how many events were delivered in the last hour.

For example, you make a call at 12:00 PM and see that you have 40,000 events remaining. Then you run the same call at 1:00 PM and see that you have 38,500 events remaining. The returned responses indicate that 1,500 events were delivered to your CometD subscribers between 12:00 PM and 1:00 PM.

These results are examples of the responses that a GET request to the `/services/data/v47.0/limits` URI returns.

```
First call result:
{
...
"MonthlyPlatformEvents" : {
    "Max" : 50000,
    "Remaining" : 40000
  },

...
}

Second call result:
{
...
"MonthlyPlatformEvents" : {
    "Max" : 50000,
    "Remaining" : 38500
  },
```

```
...
}
```

SEE ALSO:

Platform Events Developer Guide

Streaming API Developer Guide

Lightning Component Library: lightning-emp-api Lightning Web Component

Lightning Component Library: lightning:empApi Aura Component

Standard Object Notes

Learn about the characteristics of change events for some standard objects and the fields included in the event messages.

IN THIS SECTION:

Change Events for Tasks and Events

You can receive change events for single and recurring tasks and calendar events, including events with invitees.

Change Events for Person Accounts

Because a person account record combines fields from an account and a contact, changing a person account results in two change events. One change event is for the account record, and the other for the contact record. The two change events are generated for all changes to a person account, including create, update, delete, and undelete operations.

Change Events for Users

The user and email preferences in change events include only the preferences that are enabled (set to true) without their Boolean values. Preferences that are disabled (set to false) are not included in the event payload.

Change Events for Lead Conversion

Converting a lead results in the creation of an account, a contact, and optionally an opportunity, and also a lead update. When converting a lead, the change event for the lead update includes fields specific to the conversion.

Change Events for Tasks and Events

You can receive change events for single and recurring tasks and calendar events, including events with invitees.

IN THIS SECTION:

Recurring Activities

The master activity record in the series is tracked in a single change event. Each occurrence in a recurring activity is tracked by an individual change event.

Event Invitees

Change events are generated for event invitees in addition to the calendar event record. When a Salesforce user is invited to a calendar event, a child calendar event record is created for the invitee. A child calendar event is an Event record with the IsChild field set to true and OwnerId set to the invitee's user ID.

Updating Recurring Calendar Events

If a critical change is made to a recurring calendar event, such as changing the recurrence pattern or the recurrence start date, the series is deleted and recreated.

Shared Activities and Parent Records for Tasks and Events

If Shared Activities is enabled, the relationships between a task and its parent records (for example, contacts and lead), which correspond to TaskRelation objects, are tracked through change events.

Recurring Activities

The master activity record in the series is tracked in a single change event. Each occurrence in a recurring activity is tracked by an individual change event.

Example: These two change events are generated when creating a recurring calendar event. The first change event is for the master event record with GroupEventType set to 3. The second change event is for the first occurrence. The other occurrences are omitted in this example.

```
// Change event generated for the master series.
{
  "schema": "FNhgkGvYpb6gdvt_vSkpBw",
  "payload": {
    "LastModifiedDate": "2019-08-05T18:44:33.000Z",
    "Description": "Let's meet to discuss product requirements.",
    "ActivityDate": "2019-08-12",
    "IsRecurrence": false,
    "Recurrence2PatternVersion": "1",
    "GroupEventType": "3",
    "Subject": "Product Planning",
    "ActivityDateTime": "2019-08-12T19:00:00.000Z",
    "DurationInMinutes": 60,
    "OwnerId": "005RM000001iKYtYAM",
    "CreatedById": "005RM000001iKYtYAM",
    "IsAllDayEvent": false,
    "ChangeEventHeader": {
      "commitNumber": 59038718735,
      "commitUser": "005RM000001iKYtYAM",
      "sequenceNumber": 1,
      "entityName": "Event",
      "changeType": "CREATE",
      "changedFields": [],
      "changeOrigin": "com/salesforce/api/soap/47.0;client=SfdcInternalAPI/",
      "transactionKey": "0001856c-ee5d-c295-dc83-75575ad684a6",
      "commitTimestamp": 1565030673000,
      "recordIds": [
        "00URM000001W22b2AC"
      ]
    },
    "IsChild": false,
    "CreatedDate": "2019-08-05T18:44:33.000Z",
    "IsReminderSet": false,
    "ActivityRecurrence2Id": "828RM000000010WYAQ",
    "IsPrivate": false,
    "LastModifiedById": "005RM000001iKYtYAM",
    "IsRecurrence2Exclusion": false,
    "Recurrence2PatternText": "RRULE:FREQ=WEEKLY;INTERVAL=1;BYDAY=MO;COUNT=13",
    "Location": "Orcas Room",
    "ShowAs": "Busy",
    "IsGroupEvent": false
```

```
  },
  "event": {
    "replayId": 25
  }
}

// Change event generated for the first occurrence.
{
  "schema": "FNhgkGvYpb6gdvt_vSkpBw",
  "payload": {
    "LastModifiedDate": "2019-08-05T18:44:37.000Z",
    "Description": "Let's meet to discuss product requirements.",
    "ActivityDate": "2019-08-12",
    "IsRecurrence": false,
    "Recurrence2PatternVersion": "1",
    "GroupEventType": "0",
    "Subject": "Product Planning",
    "ActivityDateTime": "2019-08-12T19:00:00.000Z",
    "DurationInMinutes": 60,
    "OwnerId": "005RM000001iKYtYAM",
    "CreatedById": "005RM000001iKYtYAM",
    "IsAllDayEvent": false,
    "ChangeEventHeader": {
      "commitNumber": 59038718735,
      "commitUser": "005RM000001iKYtYAM",
      "sequenceNumber": 2,
      "entityName": "Event",
      "changeType": "CREATE",
      "changedFields": [],
      "changeOrigin": "com/salesforce/api/soap/47.0;client=SfdcInternalAPI/",
      "transactionKey": "0001856c-ee5d-c295-dc83-75575ad684a6",
      "commitTimestamp": 1565030677000,
      "recordIds": [
        "00URM000001W22c2AC"
      ]
    },
    "IsChild": false,
    "CreatedDate": "2019-08-05T18:44:37.000Z",
    "IsReminderSet": false,
    "ActivityRecurrence2Id": "828RM000000010WYAQ",
    "IsPrivate": false,
    "LastModifiedById": "005RM000001iKYtYAM",
    "IsRecurrence2Exclusion": false,
    "Recurrence2PatternText": "RRULE:FREQ=WEEKLY;INTERVAL=1;BYDAY=MO;COUNT=13",
    "Location": "Orcas Room",
    "ShowAs": "Busy",
    "IsGroupEvent": false
  },
  "event": {
    "replayId": 26
  }
}
```

Event Invitees

Change events are generated for event invitees in addition to the calendar event record. When a Salesforce user is invited to a calendar event, a child calendar event record is created for the invitee. A child calendar event is an Event record with the IsChild field set to true and OwnerId set to the invitee's user ID.

A child calendar event isn't created for an invitee who isn't a Salesforce user, such as a contact, a lead, or a resource. For each invitee added, an EventRelation record is created to represent the relationship to the calendar event. In a recurring series, child calendar events are created for invitees in each occurrence.

For example, if a calendar event is created with two invitees, three calendar event records are created in Salesforce: one record for the calendar event, and two records for the invitees. The three records result in three change events being generated on the channel for the Event standard object. In addition, two EventRelation records are created and generate two change events on the channel for EventRelation.

Example: These events are generated when creating a calendar event and inviting one user. The first change event is for the calendar event. The second change event is the child calendar event for the invitee, with IsChild set to true and OwnerID set to the invitee's user ID. The third change event is for the EventRelation record representing the relationship between the invitee and the calendar event.

```
// Change event generated for the calendar event record.
{
  "schema": "FNhgkGvYpb6gdvt_vSkpBw",
  "payload": {
    "LastModifiedDate": "2019-08-05T19:25:49.000Z",
    "Description": "Let's meet to discuss product requirements.",
    "ActivityDate": "2019-08-06",
    "IsRecurrence": false,
    "GroupEventType": "1",
    "Subject": "Product Requirements",
    "ActivityDateTime": "2019-08-06T20:00:00.000Z",
    "DurationInMinutes": 60,
    "OwnerId": "005RM000001iKYtYAM",
    "CreatedById": "005RM000001iKYtYAM",
    "IsAllDayEvent": false,
    "ChangeEventHeader": {
      "commitNumber": 59039605527,
      "commitUser": "005RM000001iKYtYAM",
      "sequenceNumber": 1,
      "entityName": "Event",
      "changeType": "CREATE",
      "changedFields": [],
      "changeOrigin": "com/salesforce/api/soap/47.0;client=SfdcInternalAPI/",
      "transactionKey": "000187ad-89e5-5035-c22e-d2521c94e3ee",
      "commitTimestamp": 1565033149000,
      "recordIds": [
        "00URM000001W22s2AC"
      ]
    },
    "IsChild": false,
    "CreatedDate": "2019-08-05T19:25:49.000Z",
    "IsReminderSet": false,
    "IsPrivate": false,
    "LastModifiedById": "005RM000001iKYtYAM",
    "IsRecurrence2Exclusion": false,
```

```
    "Location": "Orcas Room",
    "ShowAs": "Busy",
    "IsGroupEvent": true
  },
  "event": {
    "replayId": 20
  }
}

// Change event generated for the child calendar event record for the invitee.
{
  "schema": "FNhgkGvYpb6gdvt_vSkpBw",
  "payload": {
    "LastModifiedDate": "2019-08-05T19:25:49.000Z",
    "Description": "Let's meet to discuss product requirements.",
    "ActivityDate": "2019-08-06",
    "IsRecurrence": false,
    "GroupEventType": "1",
    "Subject": "Product Requirements",
    "ActivityDateTime": "2019-08-06T20:00:00.000Z",
    "DurationInMinutes": 60,
    "OwnerId": "005RM000001vSg0YAE",
    "CreatedById": "005RM000001iKYtYAM",
    "IsAllDayEvent": false,
    "ChangeEventHeader": {
      "commitNumber": 59039605527,
      "commitUser": "005RM000001iKYtYAM",
      "sequenceNumber": 2,
      "entityName": "Event",
      "changeType": "CREATE",
      "changedFields": [],
      "changeOrigin": "com/salesforce/api/soap/47.0;client=SfdcInternalAPI/",
      "transactionKey": "000187ad-89e5-5035-c22e-d2521c94e3ee",
      "commitTimestamp": 1565033149000,
      "recordIds": [
        "00URM000001W22t2AC"
      ]
    },
    "IsChild": true,
    "CreatedDate": "2019-08-05T19:25:49.000Z",
    "IsReminderSet": false,
    "IsPrivate": false,
    "LastModifiedById": "005RM000001iKYtYAM",
    "IsRecurrence2Exclusion": false,
    "Location": "Orcas Room",
    "ShowAs": "Busy",
    "IsGroupEvent": true
  },
  "event": {
    "replayId": 21
  }
}

// Change event generated for the EventRelation record representing the
```

```
// relationship between the invitee and the calendar event.
{
  "schema": "gtElhMZ7qP8-ewNsr2GXNA",
  "payload": {
    "Status": "New",
    "IsWhat": false,
    "LastModifiedDate": "2019-08-05T19:25:50.000Z",
    "CreatedById": "005RM000001iKYtYAM",
    "IsInvitee": true,
    "IsParent": false,
    "ChangeEventHeader": {
      "commitNumber": 59039605527,
      "commitUser": "005RM000001iKYtYAM",
      "sequenceNumber": 3,
      "entityName": "EventRelation",
      "changeType": "CREATE",
      "changedFields": [],
      "changeOrigin": "com/salesforce/api/soap/47.0;client=SfdcInternalAPI/",
      "transactionKey": "000187ad-89e5-5035-c22e-d2521c94e3ee",
      "commitTimestamp": 1565033150000,
      "recordIds": [
        "0RERM0000004eJI4AY"
      ]
    },
    "RelationId": "005RM000001vSg0YAE",
    "CreatedDate": "2019-08-05T19:25:50.000Z",
    "EventId": "00URM000001W22s2AC",
    "LastModifiedById": "005RM000001iKYtYAM"
  },
  "event": {
    "replayId": 22
  }
}
```

SEE ALSO:

Object Reference for Salesforce and Force.com: EventRelation

Updating Recurring Calendar Events

If a critical change is made to a recurring calendar event, such as changing the recurrence pattern or the recurrence start date, the series is deleted and recreated.

If a recurring calendar event contains many invitees and many occurrences, a critical change can lead to many change events. For example, updating the recurrence start date of a calendar event with 100 occurrences and 100 invitees results in deleting and recreating 10,000 child Event records (100 records x 100 occurrences) and 10,000 EventRelation records. A high volume of changes in a single transaction could generate overflow events. For more information, see Overflow Events.

SEE ALSO:

Object Reference for Salesforce and Force.com: EventRelation

Shared Activities and Parent Records for Tasks and Events

If Shared Activities is enabled, the relationships between a task and its parent records (for example, contacts and lead), which correspond to TaskRelation objects, are tracked through change events.

Similarly, the relationships between a calendar event and its parent records, which correspond to EventRelation objects, are tracked. You can receive change events for task relationships on the `/data/TaskRelationChangeEvent` channel, and for event relationships on the `/data/EventRelationChangeEvent` channel.

When Shared Activities is not enabled, EventRelation objects associate a calendar event with invitees only.

SEE ALSO:

Salesforce Help: Enable Shared Activities

Object Reference for Salesforce and Force.com: EventRelation

Object Reference for Salesforce and Force.com: TaskRelation

Change Events for Person Accounts

Because a person account record combines fields from an account and a contact, changing a person account results in two change events. One change event is for the account record, and the other for the contact record. The two change events are generated for all changes to a person account, including create, update, delete, and undelete operations.

IN THIS SECTION:

Creating and Undeleting a Person Account

When creating or undeleting a person account, the account change event contains both account and contact fields that are not null. It contains all non-null fields from the account record and some fields from the contact record. The contact fields that the account change event includes are all custom contact fields and some standard contact fields, which start with the Person prefix.

Updating a Person Account

When updating a person account, two change events are generated, one for the account and one for the contact, regardless which fields changed. Salesforce always updates the LastModifiedDate system field in both the account and contact even if the field updated is only in one of the underlying records.

Converting an Account

If a person account is converted to a business account through the API by modifying the record type ID, a change event for the account is generated. This change event contains the new record type ID of the account.

Deleting a Person Account

When deleting a person account, two change events are generated: one for the deleted account and one for the deleted contact. The change events don't contain record fields. They contain only event header fields.

SEE ALSO:

Salesforce Help: Person Accounts

Creating and Undeleting a Person Account

When creating or undeleting a person account, the account change event contains both account and contact fields that are not null. It contains all non-null fields from the account record and some fields from the contact record. The contact fields that the account change event includes are all custom contact fields and some standard contact fields, which start with the Person prefix.

The contact change event contains all contact standard and custom fields. The contact change event doesn't contain the account fields of the person account.

The Name field is included in both the account and contact change events for a new or undeleted person account.

Example: These two change events are generated when a person account is created. The first event is for the Account standard object. The account change event contains the Name field, which is the person account name. This field is also included in the contact change event. The second event is for the Contact standard object. The contact has a custom field named CustomContactField__c, which is part of both the contact and account change events. In the account change event, the contact custom field ends with the __pc prefix.

```
// Change event for Account
{
  "schema": "_XIQSYpQxN_6CRHk2UO_dg",
  "payload": {
    "LastModifiedDate": "2019-08-05T20:49:10.000Z",
    "CreatedById": "005RM000001vI4mYAE",
    "OwnerId": "005RM000001vI4mYAE",
    "PersonContactId": "003RM000006IG7WYAW",
    "Phone": "(415) 555-1212",
    "ChangeEventHeader": {
      "commitNumber": 59041562583,
      "commitUser": "005RM000001vI4mYAE",
      "sequenceNumber": 1,
      "entityName": "Account",
      "changeType": "CREATE",
      "changedFields": [],
      "changeOrigin": "com/salesforce/api/soap/47.0;client=SfdcInternalAPI/",
      "transactionKey": "00018d42-5322-5ccb-9fd6-d7221a2f9467",
      "commitTimestamp": 1565038150000,
      "recordIds": [
        "001RM000004OZA8YAO"
      ]
    },
    "CustomContactField__pc": "001",
    "CreatedDate": "2019-08-05T20:49:10.000Z",
    "RecordTypeId": "012RM000000FngOYAS",
    "LastModifiedById": "005RM000001vI4mYAE",
    "Name": {
      "FirstName": "Jane",
      "LastName": "Smith"
    }
  },
  "event": {
    "replayId": 14
  }
}

// Change event for Contact
{
  "schema": "42pKC5JiBJqKHPog_CT94A",
  "payload": {
    "LastModifiedDate": "2019-08-05T20:49:10.000Z",
    "AccountId": "001RM000004OZA8YAO",
    "OwnerId": "005RM000001vI4mYAE",
```

```
    "CreatedById": "005RM000001vI4mYAE",
    "IsPersonAccount": true,
    "Phone": "(415) 555-1212",
    "ChangeEventHeader": {
      "commitNumber": 59041562583,
      "commitUser": "005RM000001vI4mYAE",
      "sequenceNumber": 2,
      "entityName": "Contact",
      "changeType": "CREATE",
      "changedFields": [],
      "changeOrigin": "com/salesforce/api/soap/47.0;client=SfdcInternalAPI/",
      "transactionKey": "00018d42-5322-5ccb-9fd6-d7221a2f9467",
      "commitTimestamp": 1565038151000,
      "recordIds": [
        "003RM000006IG7WYAW"
      ]
    },
    "CreatedDate": "2019-08-05T20:49:10.000Z",
    "CustomContactField__c": "001",
    "LastModifiedById": "005RM000001vI4mYAE",
    "Name": {
      "FirstName": "Jane",
      "LastName": "Smith"
    }
  },
  "event": {
    "replayId": 15
  }
}
```


SEE ALSO:

Salesforce Help: Person Account Fields (by Label Names)

Updating a Person Account

When updating a person account, two change events are generated, one for the account and one for the contact, regardless which fields changed. Salesforce always updates the LastModifiedDate system field in both the account and contact even if the field updated is only in one of the underlying records.

Because a person account corresponds to one account and one contact, the timestamp fields of the account and contact records must match. If an account-only field is updated, such as the Industry field, the account change event contains the changed field and the LastModifiedDate field. The contact change event contains only the LastModifiedDate field. If the updated field stems from a contact, or is a custom contact field, both change events contain all changed fields and the LastModifiedDate field. In particular, if a person account's first name or last name is modified, the corresponding field is included in both change events.

Converting an Account

If a person account is converted to a business account through the API by modifying the record type ID, a change event for the account is generated. This change event contains the new record type ID of the account.

Conversely, if a business account is converted to a person account, a change event is generated for the account with the new record type ID.

Deleting a Person Account

When deleting a person account, two change events are generated: one for the deleted account and one for the deleted contact. The change events don't contain record fields. They contain only event header fields.

Change Events for Users

The user and email preferences in change events include only the preferences that are enabled (set to true) without their Boolean values. Preferences that are disabled (set to false) are not included in the event payload.

For a list of user and email preferences, see the User Object in the SOAP API Developer Guide.

Note: Preferences are stored in a 32-bit integer internal field in the database. When a preference is changed, a change is detected in the corresponding 32-bit integer field, and all enabled preferences that are represented by that internal integer field are published, whether or not they were changed.

Example: This change event is generated when a User record is created. Preferences are included under EmailPreferences and UserPreferences.


```
{
  "schema": "93etK5psDNYN7fkR6LeDVw",
  "payload": {
  "ChangeEventHeader": {
    "commitNumber": 59039569143,
    "commitUser": "005RM000001iKYtYAM",
    "sequenceNumber": 1,
    "entityName": "User",
    "changeType": "CREATE",
    "changedFields": [],
    "changeOrigin": "",
    "transactionKey": "00018898-30ff-31da-204e-aaf5973aaf25",
    "commitTimestamp": 1565033021000,
    "recordIds": [
      "005RM000001vSg0YAE"
    ]
  },
  "ProfileId": "00eRM000000zej4YAA",
  "EmailPreferences": [
    "AutoBcc",
    "StayInTouchReminder"
  ],
  "UserPreferences": [
      "ActivityRemindersPopup",
      "EventRemindersCheckboxDefault",
      "TaskRemindersCheckboxDefault",
      "DisableLikeEmail",
      "SortFeedByComment",
      "ShowTitleToExternalUsers",
      "LightningExperiencePreferred",
      "HideSfxWelcomeMat"
    ],
    "LastModifiedDate": "2019-08-05T19:23:41.000Z",
    "ReceivesAdminInfoEmails": true,
    "Email": "jane.smith@example.com",
```

```
    "LanguageLocaleKey": "en_US",
    "TimeZoneSidKey": "America/Los_Angeles",
    "IsActive": true,
    "ForecastEnabled": false,
    "DigestFrequency": "D",
    "Name": {
      "FirstName": "Jane",
      "LastName": "Smith"
    },
    "IsProfilePhotoActive": false,
    "CommunityNickname": "jane.smith",
    "CreatedById": "005RM000001iKYtYAM",
    "UserPermissions": [],
    "LocaleSidKey": "en_US",
    "IsSystemControlled": false,
    "EmailEncodingKey": "ISO-8859-1",
    "ReceivesInfoEmails": true,
    "Username": "jane.smith@example.com",
    "Alias": "janes",
    "DefaultGroupNotificationFrequency": "N",
    "CreatedDate": "2019-08-05T19:23:41.000Z",
    "UserType": "Standard",
    "LastModifiedById": "005RM000001iKYtYAM"
  },
  "event": {
    "replayId": 27
  }
}
```


Change Events for Lead Conversion

Converting a lead results in the creation of an account, a contact, and optionally an opportunity, and also a lead update. When converting a lead, the change event for the lead update includes fields specific to the conversion.

These fields are included in the lead update change event for a lead conversion.

Field	Description
`Status`	The lead conversion status. Possible status values are in the LeadStatus standard object.
`IsConverted`	Indicates whether the lead was converted (`true`).
`ConvertedDate`	The date of the lead conversion. `ConvertedDate` doesn't include the time.
`ConvertedAccountId`	The ID of the account created in the lead conversion.
`ConvertedContactId`	The ID of the contact created in the lead conversion.
`ConvertedOpportunityId`	The ID of the opportunity created in the lead conversion.

The change event for the lead update doesn't include the `LastModifiedDate` field.

For an example lead update change event for a lead conversion, see Lead Update Change Event in the Example section.

Example: These example change events are generated when converting a lead. The order of the change events corresponds to the sequence of operations: the creation of an account, contact, opportunity, and the lead update. The `sequenceNumber` field in each change event denotes the sequence of the operations in the same transaction.

Account Create Change Event


```
{
  "data": {
    "schema": "t9m-QIHrQ-DjEtyYLAowhA",
    "payload": {
      "LastModifiedDate": "2021-05-03T23:30:14.000Z",
      "Name": "Cadinal Inc.",
      "CreatedById": "00550000001N45jAAC",
      "AccountSource": "Web",
      "CreatedDate": "2021-05-03T23:30:14.000Z",
      "BillingAddress": {
        "State": "IL",
        "Country": "USA"
      },
      "OwnerId": "00550000001N45jAAC",
      "Phone": "(847) 262-5000",
      "ChangeEventHeader": {
        "commitNumber": 10969942389321,
        "commitUser": "00550000001N45jAAC",
        "sequenceNumber": 1,
        "entityName": "Account",
        "changeType": "CREATE",
        "changedFields": [
        ],
        "changeOrigin": "com/salesforce/api/soap/51.0;client=SfdcInternalAPI/",
        "transactionKey": "0003dc15-ae1e-8042-f9a1-cc4ed6f1bc51",
        "commitTimestamp": 1620084614000,
        "recordIds": [
          "0012J00002SrYmBQAV"
        ]
      },
      "LastModifiedById": "00550000001N45jAAC"
    },
    "event": {
      "replayId": 7635322
    }
  },
  "channel": "/data/ChangeEvents"
}
```


Contact Create Change Event


```
{
  "data": {
    "schema": "zvY0n7RbB6-4Gy3s5TfJCw",
    "payload": {
      "LastModifiedDate": "2021-05-03T23:30:14.000Z",
      "AccountId": "0012J00002SrYmBQAV",
      "Email": "brenda@cardinal.net",
      "Name": {
```

```
        "FirstName": "Brenda",
        "LastName": "Mcclure"
      },
      "OwnerId": "00550000001N45jAAC",
      "CreatedById": "00550000001N45jAAC",
      "Phone": "(847) 262-5000",
      "Title": "CFO",
      "MailingAddress": {
        "State": "IL",
        "Country": "USA"
      },
      "LeadSource": "Web",
      "ChangeEventHeader": {
        "commitNumber": 10969942389321,
        "commitUser": "00550000001N45jAAC",
        "sequenceNumber": 2,
        "entityName": "Contact",
        "changeType": "CREATE",
        "changedFields": [
        ],
        "changeOrigin": "com/salesforce/api/soap/51.0;client=SfdcInternalAPI/",
        "transactionKey": "0003dc15-ae1e-8042-f9a1-cc4ed6f1bc51",
        "commitTimestamp": 1620084614000,
        "recordIds": [
          "0032J00003f7EtfQAE"
        ]
      },
      "CreatedDate": "2021-05-03T23:30:14.000Z",
      "LastModifiedById": "00550000001N45jAAC"
    },
    "event": {
      "replayId": 7635323
    }
  },
  "channel": "/data/ChangeEvents"
}
```


Opportunity Create Change Event


```
{
  "data": {
    "schema": "ahbw80yzNnEGGuB-KtyjvQ",
    "payload": {
      "LastModifiedDate": "2021-05-03T23:30:14.000Z",
      "AccountId": "0012J00002SrYmBQAV",
      "HasOpportunityLineItem": false,
      "ForecastCategory": "Closed",
      "IsClosed": true,
      "CloseDate": "2021-06-30",
      "Name": "Cadinal Inc.-",
      "OwnerId": "00550000001N45jAAC",
      "CreatedById": "00550000001N45jAAC",
      "IsWon": true,
      "StageName": "Prospecting",
```

```
      "Probability": 10,
      "LeadSource": "Web",
      "ChangeEventHeader": {
        "commitNumber": 10969942389321,
        "commitUser": "00550000001N45jAAC",
        "sequenceNumber": 3,
        "entityName": "Opportunity",
        "changeType": "CREATE",
        "changedFields": [
        ],
        "changeOrigin": "com/salesforce/api/soap/51.0;client=SfdcInternalAPI/",
        "transactionKey": "0003dc15-ae1e-8042-f9a1-cc4ed6f1bc51",
        "commitTimestamp": 1620084614000,
        "recordIds": [
          "0062J00000r4X4QQAU"
        ]
      },
      "CreatedDate": "2021-05-03T23:30:14.000Z",
      "IsPrivate": false,
      "ForecastCategoryName": "Closed",
      "LastModifiedById": "00550000001N45jAAC"
    },
    "event": {
      "replayId": 7635324
    }
  },
  "channel": "/data/ChangeEvents"
}
```

Lead Update Change Event

```
{
  "data": {
    "schema": "XBX78x3GtAi1BfOW4Yxzqg",
    "payload": {
      "ConvertedOpportunityId": "0062J00000r4X4QQAU",
      "Status": "Closed - Converted",
      "ConvertedAccountId": "0012J00002SrYmBQAV",
      "IsConverted": true,
      "ConvertedDate": "2021-05-03",
      "ConvertedContactId": "0032J00003f7EtfQAE",
      "ChangeEventHeader": {
        "commitNumber": 10969942389321,
        "commitUser": "00550000001N45jAAC",
        "sequenceNumber": 4,
        "entityName": "Lead",
        "changeType": "UPDATE",
        "changedFields": [
          "Status",
          "IsConverted",
          "ConvertedDate",
          "ConvertedAccountId",
          "ConvertedContactId",
          "ConvertedOpportunityId"
```

```
        ],
        "changeOrigin": "com/salesforce/api/soap/51.0;client=SfdcInternalAPI/",
        "transactionKey": "0003dc15-ae1e-8042-f9a1-cc4ed6f1bc51",
        "commitTimestamp": 1620084615000,
        "recordIds": [
          "00Q5000000TZYWOEA5"
        ]
      }
    },
    "event": {
      "replayId": 7635325
    }
  },
  "channel": "/data/ChangeEvents"
}
```


Change Events for Fields

Learn about the change event characteristics for fields.

IN THIS SECTION:

Sending Data Differences for Fields of Updated Records

To reduce the event payload size and improve performance, Salesforce sometimes sends data differences of updated text values. For large text fields, such as Description or Long Text Area fields that contain at least 1,000 characters, only the data differences might be sent. Data differences use the unified diff format.

Change Events for Custom Field Type Conversions

When you change the type of a custom field, a change event or gap event is generated for data changes for some conversions. Other conversions, such as those that preserve or truncate field values, don't generate events.

Sending Data Differences for Fields of Updated Records

To reduce the event payload size and improve performance, Salesforce sometimes sends data differences of updated text values. For large text fields, such as Description or Long Text Area fields that contain at least 1,000 characters, only the data differences might be sent. Data differences use the unified diff format.

Differences are computed for each line in the text value. The diff algorithm breaks the field value into lines by using the line breaks found in the value.

If sending the diff for updates of large text fields does not reduce the field size, the entire value is sent. The diff value is **not** sent for the following conditions.

- The length of the field value is less than 1,000 characters.
- The difference between the old and new values is greater than 50% in length.
- More than 25% of the lines of the total of number of lines in the old and new values are changed.
- The diff's length is greater than the length of the new value.

For more information about the unified diff format and the diff utility, see the Diff Utility Wikipedia article.

The diff value includes an SHA-256 hash value that is computed on the entire updated value. Use the hash value to verify that the reconstructed value matches the original value before it was converted to a diff. To do so, compute the SHA-256 hash after expanding

the diff value. Then compare the two hash values to ensure that they're equal. If the reconstructed content is different from the original content, the hash value is different. To compute an SHA-256 hash value, you can use a utility such as the UNIX sha256sum command or the DigestUtils class from the Apache Commons library.

IN THIS SECTION:

Data Differences in Event Fields
When the updated text field value is sent as a diff, the field includes the diff subfield which contains the the SHA-256 hash value and data differences in the unified diff format. The structure of the diff field is as follows.

How to Reconstruct a Field from Its Diff Value
The value of a diff field is in the unified diff format. Use a diff utility to obtain the full field value from the diff.

Considerations for Newline Characters and Computing the SHA-256 Hash
The content that Salesforce uses to generate the SHA-256 hash might have newline characters transformed by the browser. Many browsers transform newline characters to `\r\n` in record field values before records are stored in Salesforce. Also, Salesforce trims leading and trailing white spaces in field values.

Data Differences in Event Fields

When the updated text field value is sent as a diff, the field includes the diff subfield which contains the the SHA-256 hash value and data differences in the unified diff format. The structure of the diff field is as follows.

```
"<Field_Name>": {
      "diff": "--- \n+++ <hash_value>\n
                (Changes)"
}
```

Example: This change event is sent after the Description field with more than 1,000 characters is updated for an account. The Description field contains the hash value after the `+++` prefix followed by the data differences.

```
{
  "schema": "f_i77-QWWmMztk9VVhxbQg",
  "payload": {
    "LastModifiedDate": "2019-08-05T21:11:41.000Z",
    "Description": {
      "diff": "--- \n+++
682b8747ccdb93b546e7bbe479b27d26ec7c38ccabb76cdd8308c6595492bffc\n@@ -2,1 +2,1
@@\n-Business applications are moving to the cloud. It's not just a fad—the shift from
 traditional software models to the Internet has steadily gained momentum over the
last 10 years.\n+Business apps are moving to the cloud. It's not just a fad—the shift
 from traditional software models to the Internet has steadily gained momentum over
the last 10 years.\n@@ -7,1 +7,1 @@\n-As cloud computing grows in popularity, thousands
 of companies are simply rebranding their non-cloud products and services as "cloud
computing." Always dig deeper when evaluating cloud offerings.\n+As cloud computing
grows in popularity, thousands of companies are simply rebranding their non-cloud
products and services as "cloud computing." Always dig deeper when evaluating cloud
offerings. And keep in mind that if you have to buy and manage hardware and software,
 what you're looking at isn't really cloud computing but a false cloud."
    },
    "ChangeEventHeader": {
      "commitNumber": 59042231215,
      "commitUser": "005RM000001iKYtYAM",
      "sequenceNumber": 1,
```

```
      "entityName": "Account",
      "changeType": "UPDATE",
      "changedFields": [
        "Description",
        "LastModifiedDate"
      ],
      "changeOrigin": "com/salesforce/api/soap/47.0;client=SfdcInternalAPI/",
      "transactionKey": "00018d74-98a7-3096-438d-2c171e95a47a",
      "commitTimestamp": 1565039501000,
      "recordIds": [
        "001RM000004OZB2YAO"
      ]
    }
  },
  "event": {
    "replayId": 2
  }
}
```


The updates made to the diff field are the following.

- In the paragraph starting with "Business applications," the word "applications" was replaced with "apps."
- In the paragraph starting with "As cloud computing grows,", one sentence was appended at the end of the paragraph: "And keep in mind that if you have to buy and manage hardware and software, what you're looking at isn't really cloud computing but a false cloud."

The following event for the account creation shows the original and full values in the Description field before it was updated. If you generate the SHA-256 hash on the full value, you get the same value sent in the account update event (`682b8747ccdb93b546e7bbe479b27d26ec7c38ccabb76cdd8308c6595492bffc`).


```
{
  "schema": "f_i77-QWWmMztk9VVhxbQg",
  "payload": {
    "LastModifiedDate": "2019-08-05T21:09:45.000Z",
    "Description": "Everyone is talking about "the cloud." But what does it
mean?\r\nBusiness applications are moving to the cloud. It's not just a fad—the shift
 from traditional software models to the Internet has steadily gained momentum over
the last 10 years.\r\nCloud computing: a better way\r\nWith cloud computing, you
eliminate headaches because you're not managing hardware and software—that's the
responsibility of an experienced vendor like Salesforce. The shared infrastructure
means it works like a utility: you only pay for what you need, upgrades are automatic,
 and scaling up or down is easy.\r\nCloud-based apps can be up and running in days or
 weeks, and they cost less. With a cloud app, you just open a browser, log in, customize
 the app, and start using it.\r\nBusinesses are running all kinds of apps in the cloud,
 like customer relationship management (CRM), HR, accounting, and much more. Some of
the world's largest companies moved their applications to the cloud with Salesforce
after rigorously testing the security and reliability of our infrastructure.\r\nAs
cloud computing grows in popularity, thousands of companies are simply rebranding their
 non-cloud products and services as "cloud computing." Always dig deeper when evaluating
 cloud offerings.",
    "OwnerId": "005RM000001iKYtYAM",
    "CreatedById": "005RM000001iKYtYAM",
    "ChangeEventHeader": {
      "commitNumber": 59042182355,
      "commitUser": "005RM000001iKYtYAM",
```

```
      "sequenceNumber": 1,
      "entityName": "Account",
      "changeType": "CREATE",
      "changedFields": [],
      "changeOrigin": "com/salesforce/api/soap/47.0;client=SfdcInternalAPI/",
      "transactionKey": "00018d59-a5fc-38be-8632-d1ca8cdc182a",
      "commitTimestamp": 1565039385000,
      "recordIds": [
        "001RM000004OZB2YAO"
      ]
    },
    "CreatedDate": "2019-08-05T21:09:45.000Z",
    "LastModifiedById": "005RM000001iKYtYAM",
    "Name": "Acme"
  },
  "event": {
    "replayId": 1
  }
}
```

How to Reconstruct a Field from Its Diff Value

The value of a diff field is in the unified diff format. Use a diff utility to obtain the full field value from the diff.

For example, you can use the Java Diff Utilities library.

The following code sample shows the order of operations and the library tools used. The toLines() method, which you implement, splits the diff value into a list of lines. The BufferedReader Java object determines how the newline character is represented, so you don't need to pass in the newLine value.

Next, patches are obtained from the diff lines through the Java diff utility method DiffUtils.parseUnifiedDiff(). The patches are the changes applied to the content. The `toLines()` method is called again to split the original content into lines. The patches are then applied to the original lines using the `DiffUtils.patch()` method.

You implement the `combineLines()` method to combine the updated lines into one string variable. The `newLine` variable is passed to `combineLines()` to reintroduce the original line breaks in the text. Set the `newLine` variable to the newline character sequence that was in the original content (`\r\n` or `\n`). For more information, see Considerations for Newline Characters and Computing the SHA-256 Hash. The revised string variable is the reconstructed value from the diff that contains the updates.

The final step is to generate a SHA-256 hash value to validate that the original updated value matches the reconstructed value. To generate a hash, use the Apache Common DigestUtils library. After the hash is generated, compare it to the one sent in the event and ensure that both hash values are equal.

```
public void BuildOriginalValueFromDiff(String original, String diff, String newLine) {
    // Split diff value into lines and get patches.
    List<String> diffLines = toLines(diff);
    Patch<String> patch = DiffUtils.parseUnifiedDiff(diffLines);

    // Split original text into lines.
    List<String> originalLines = toLines(original);

    // Apply patches to original lines, then combined lines.
    List<String> revisedLines = DiffUtils.patch(originalLines, patch);
    String revised = combineLines(revisedLines, newLine);
```

```
    // Generate SHA-256 hash on reconstructed value.
    String checkSum = DigestUtils.sha256Hex(revised);

    // Extract hash from the event diff field.

    // Compare extracted hash with generated hash and verify they are equal.
}
```

The following are examples of what to implement to split lines and combine lines.

```
private List<String> toLines(String s) {
    BufferedReader rd = new BufferedReader(new StringReader(s));
    return rd.lines().collect(Collectors.toList());
}

private String combineLines(List<String> lines, String newLine) {
    StringBuilder sb = new StringBuilder();
    lines.forEach(l -> sb.append(l).append(newLine));
    sb.deleteCharAt(sb.length()-newLine.length());  // remove last newline added
    return sb.toString();
}
```

Considerations for Newline Characters and Computing the SHA-256 Hash

The content that Salesforce uses to generate the SHA-256 hash might have newline characters transformed by the browser. Many browsers transform newline characters to `\r\n` in record field values before records are stored in Salesforce. Also, Salesforce trims leading and trailing white spaces in field values.

Before you generate the SHA-256 hash value, ensure that the reconstructed content from the diff contains the same newline characters as the original content and that no new leading or trailing white spaces are added. For example, when you save the content in a file, the operating system can add a trailing white space character.

Note: If you used the API to create or update field values, the newline characters supplied by the application are honored and stored in Salesforce without further transformations.

Windows systems represent the newline character as a carriage return and line-feed character sequence (`\r\n`). UNIX and UNIX-based systems, like macOS and Linux, represent the newline character as a line-feed character (`\n`).

Change Events for Custom Field Type Conversions

When you change the type of a custom field, a change event or gap event is generated for data changes for some conversions. Other conversions, such as those that preserve or truncate field values, don't generate events.

IN THIS SECTION:

Conversions That Generate a Change Event

When converting a custom field type to another type that is not compatible, field data is lost and is set to null in records corresponding to the object. One change event is generated for all the affected records, and the event message contains no record fields.

Conversions That Generate a Gap Event

A gap event is generated for all the affected records for some field conversions from Picklist. The change event header of the gap event message contains information about the records, including the record IDs and a change type of `GAP_UPDATE`.

Conversions That Don't Generate Events

No change or gap events are generated for custom field type conversions that preserve or truncate field data, and for conversions between Picklist and Text fields.

SEE ALSO:

Salesforce Help: Custom Field Types

Salesforce Help: Notes on Changing Custom Field Types

Salesforce Help: Custom Field Types

Conversions That Generate a Change Event

When converting a custom field type to another type that is not compatible, field data is lost and is set to null in records corresponding to the object. One change event is generated for all the affected records, and the event message contains no record fields.

Examples of incompatible field changes are:

- Changing a Date or Date/Time field to any other field type, and vice versa
- Changing a Checkbox field to any other field type
- Changing a Picklist (Multi-Select) field to any other field type

For a complete list of conversions that result in data loss, see Notes on Changing Custom Field Types.

Because a field type conversion can affect many records, the `recordIds` header field value in the event message contains a wildcard value instead of a record ID array. The value starts with the three-character object ID prefix, followed by the wildcard character `*`. For example, if you make an incompatible field type change for an Account custom field, the `recordIds` field looks similar to the following.

```
"ChangeEventHeader": {
        "entityName": "Account",
        "recordIds": [
          "001*"
        ],
        ...
}
```

SEE ALSO:

Change Event Header Fields

Conversions That Generate a Gap Event

A gap event is generated for all the affected records for some field conversions from Picklist. The change event header of the gap event message contains information about the records, including the record IDs and a change type of `GAP_UPDATE`.

These field type conversions generate a gap event.

- Changing a Picklist field to Checkbox
- Changing a Picklist field to Picklist (Multi-Select)

SEE ALSO:

Gap Events

Conversions That Don’t Generate Events

No change or gap events are generated for custom field type conversions that preserve or truncate field data, and for conversions between Picklist and Text fields.

Compatible Field Types with No Data Change

When converting a field type to another type that is compatible, field data is unchanged, and no event is generated. For example, these conversions are compatible.

- Changing a Text Area, Email, Url, Phone, Autonumber, Number, Percent, or Currency field to a Text field
- Changing a Text field to a Text Area, Text Area (Long), Email, Url, Phone, or Autonumber field

Other Field Type Conversions

These field type conversions also don't generate events.

- Changing a Picklist field to a Text field
- Changing a Text field to a Picklist field
- Conversions that result in truncated data because the target field type has a smaller size, such as changing a Text Area (Long) field to a Text, Text Area, Email, Url, or Phone field