**Components of Event-Driven Systems**

Before we go any further, let’s define some terms.

**Event**  
A change in state that is meaningful in a business process. For example, placement of a purchase order is a meaningful event, because the order fulfillment center expects to receive a notification before processing an order.  
  
**Event message**  
A message that contains data about the event. Also known as an event notification. For example, an event message can be a notification about an order placement containing information about the order.  
  
**Event producer**  
The publisher of an event message. For example, an order placement app.  
  
**Event channel**  
A stream of events on which an event producer sends event messages and event consumers read those messages. For platform events, the channel is for one platform event and groups all event messages for that platform event.  
  
**Event consumer**  
A subscriber to a channel that receives messages from the channel. For example, an order fulfillment app that is notified of new orders.

**Event bus**

A multitenant, multicloud event storage and delivery service based on a publish-subscribe model. The event bus enables the retrieval of stored event messages at any time during the retention window. The event bus is based on a time-ordered event log, which ensures that event messages are stored and delivered in the order that they’re received by Salesforce.

Types of events:

<https://developer.salesforce.com/docs/atlas.en-us.api_streaming.meta/api_streaming/event_comparison.htm>

[https://developer.salesforce.com/docs/atlas.en-us. https://developer.salesforce.com/docs/atlas.en-us.api\_streaming.meta/api\_streaming/\_events.meta/platform\_events/platform\_events\_intro.htm](https://developer.salesforce.com/docs/atlas.en-us.platform_events.meta/platform_events/platform_events_intro.htm)

<https://developer.salesforce.com/docs/atlas.en-us.api_streaming.meta/api_streaming/using_streaming_api_durability.htm>

<https://developer.salesforce.com/blogs/2018/07/which-streaming-event-do-i-use.html>

Define a platform event named Cloud News:

1. From Setup, enter Platform Events in the Quick Find box, then select **Platform Events**.
2. On the Platform Events page, click **New Platform Event**.
3. For Label, enter Cloud News.
4. For Plural Label, enter Cloud News.
5. For Description, enter Cloud news events deliver news at your fingertips.
6. For Publish Behavior, keep the default of **Publish After Commit**.
7. Click **Save**.
8. In the Custom Fields & Relationships related list, click **New**.
9. Select **Text**, and click **Next**.
10. For Field Label/Name, type Location.
11. For Length, type 100. Keep the defaults for the other fields and leave the Description field empty. Click **Save**.

Apex call to publish an event:

// Create an instance of the event and store it in the newsEvent variable

Cloud\_News\_\_e newsEvent = new Cloud\_News\_\_e(

Location\_\_c='Mountain City',

Urgent\_\_c=true,

News\_Content\_\_c='Lake Road is closed due to mudslides.');

// Call method to publish events

Database.SaveResult sr = EventBus.publish(newsEvent);

// Inspect publishing result

if (sr.isSuccess()) {

System.debug('Successfully published event.');

} else {

for(Database.Error err : sr.getErrors()) {

System.debug('Error returned: ' +

err.getStatusCode() +

' - ' +

err.getMessage());

}

}

Publish a list of Events:

// List to hold event objects to be published.

List<Cloud\_News\_\_e> newsEventList = new List<Cloud\_News\_\_e>();

// Create event objects.

Cloud\_News\_\_e newsEvent1 = new Cloud\_News\_\_e(

Location\_\_c='Mountain City',

Urgent\_\_c=true,

News\_Content\_\_c='Lake Road is closed due to mudslides.');

Cloud\_News\_\_e newsEvent2 = new Cloud\_News\_\_e(

Location\_\_c='Mountain City',

Urgent\_\_c=false,

News\_Content\_\_c='Small incident on Goat Lane causing traffic.');

// Add event objects to the list.

newsEventList.add(newsEvent1);

newsEventList.add(newsEvent2);

// Call method to publish events.

List<Database.SaveResult> results = EventBus.publish(newsEventList);

// Inspect publishing result for each event

for (Database.SaveResult sr : results) {

if (sr.isSuccess()) {

System.debug('Successfully published event.');

} else {

for(Database.Error err : sr.getErrors()) {

System.debug('Error returned: ' +

err.getStatusCode() +

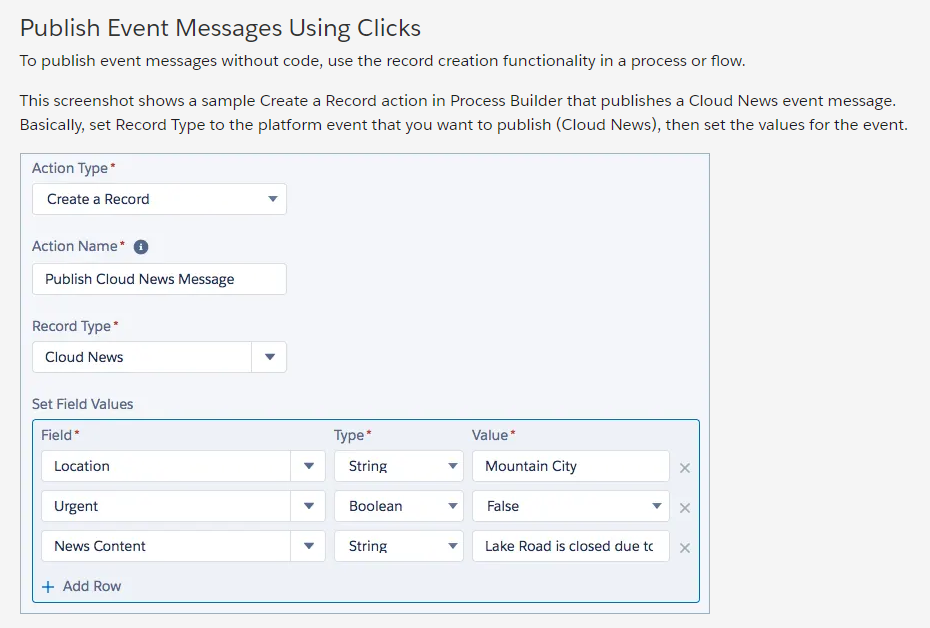
' - ' +

err.getMessage());

}

}

}



Publish via API call:

Define a platform event named Cloud News:

1. From Setup, enter Platform Events in the Quick Find box, then select **Platform Events**.
2. On the Platform Events page, click **New Platform Event**.
3. For Label, enter Cloud News.
4. For Plural Label, enter Cloud News.
5. For Description, enter Cloud news events deliver news at your fingertips.
6. For Publish Behavior, keep the default of **Publish After Commit**.
7. Click **Save**.
8. In the Custom Fields & Relationships related list, click **New**.
9. Select **Text**, and click **Next**.
10. For Field Label/Name, type Location.
11. For Length, type 100. Keep the defaults for the other fields and leave the Description field empty. Click **Save**.

* Text Area (Long)

**Event Retention and ReplayId System Field**

Salesforce stores high-volume platform events for 72 hours in the event bus. Standard-volume events that were defined before Spring ’19 are stored for 24 hours in the event bus.

Note

Newly defined events are high volume by default. Standard-volume events are the predecessors of high-volume events, and you can no longer define such events.

You can retrieve stored events from the event bus using API CometD clients. You can retrieve all stored events, or you can specify the replay ID of an event as the baseline for the retrieved portion of events. With Apex triggers, you can resume a suspended trigger and pick up the earliest retained unprocessed events. You learn later in this module how to manage Apex trigger subscriptions.

Even though Salesforce retains event messages temporarily, you can’t query them through SOQL or SOSL. Similarly, you can’t use event messages in the user interface in reports, list views, and search. You can retrieve past events only when subscribing in CometD and using a ReplayId option. We show you how to subscribe to events in the next unit.

Each event message is assigned an opaque ID contained in the ReplayId field. The ReplayId field value, which is populated by the system when the event is delivered to subscribers, refers to the position of the event in the event stream. Replay ID values are not guaranteed to be contiguous for consecutive events. For example, the event following the event with ID 999 can have an ID of 1,025. A subscriber can store a replay ID value and use it on resubscription to retrieve events that are within the retention window. For example, a subscriber can retrieve missed events after a connection failure. Subscribers must not compute new replay IDs based on a stored replay ID to refer to other events in the stream.

**API Name Suffix**

When you create a platform event, the system appends the \_\_e suffix to create the API name of the event. For example, for the Cloud News event, the API name is Cloud\_News\_\_e. Use the API name whenever you refer to the event programmatically, for example, in Apex, REST API, and Enterprise API.

**Platform Events and Transactions**

Platform events that are defined with the **Publish After Commit** publish behavior are transactional and can be rolled back, but those defined with the **Publish Immediately** publish behavior are not and cannot be rolled back. Note the following differences in behavior.

**Publish After Commit Publish Behavior**

A platform event defined with the **Publish After Commit** behavior is published only after a transaction commits successfully. Define an event with this option if subscribers rely on data that the publishing transaction commits. For example, a process publishes an event message and creates a task record. A second process that is subscribed to the event is fired and expects to find the task record. Another reason for choosing this behavior is when you don't want the event message to be published if the transaction fails.

**Publish Immediately Publish Behavior**

A platform event defined with the **Publish Immediately** behavior is published when the publish call executes. Select this option if you want the event message to be published regardless of whether the transaction succeeds. Also choose this option if the publisher and subscribers are independent, and subscribers don't rely on data committed by the publisher. For example, the immediate publishing behavior is suitable for an event used for logging purposes. With this option, a subscriber might receive the event message before data is committed by the publisher transaction.

**Publish Events**

If your app is on the Salesforce platform, you can publish events using an Apex method or with declarative tools, such as Process Builder or Flow Builder. If your app is an external app, you can publish events using Salesforce APIs.

**Publish Event Messages Using Apex**

To publish event messages, you create an instance of the event and pass it to the EventBus.publish() method.

The following example creates one event of type Cloud\_News\_\_e, publishes it, and then checks whether the publishing was successful or encountered errors. The EventBus.publish() method returns a Database.SaveResult object, which contains the result of the publishing. If isSuccess() returns true, the publish request is queued in Salesforce and the event message is published asynchronously. If isSuccess() returns false, the event publish operation resulted in errors, which are returned in the Database.Error object. This method doesn’t throw an exception due to an unsuccessful publish operation.

You can execute the Apex code snippet in the Developer Console.

1. From the quick access menu ( Quick access menu), select **Developer Console**.
2. Click **Debug** | **Open Execute Anonymous Window**.
3. In the new window, replace any contents with the code snippet and then click **Execute**

// Create an instance of the event and store it in the newsEvent variable

Cloud\_News\_\_e newsEvent = new Cloud\_News\_\_e(

Location\_\_c='Mountain City',

Urgent\_\_c=true,

News\_Content\_\_c='Lake Road is closed due to mudslides.');

// Call method to publish events

Database.SaveResult sr = EventBus.publish(newsEvent);

// Inspect publishing result

if (sr.isSuccess()) {

System.debug('Successfully published event.');

} else {

for(Database.Error err : sr.getErrors()) {

System.debug('Error returned: ' +

err.getStatusCode() +

' - ' +

err.getMessage());

}

}

To publish more than one event in the same call, add your events to a list of events, and pass the list to the EventBus.publish() method. The output of this method is an array of Database.SaveResult objects: one for each published event. EventBus.publish() can publish some passed-in events, even when other events can’t be published due to errors. The EventBus.publish() method doesn’t throw exceptions caused by an unsuccessful publish operation. It is similar in behavior to the Apex Database.insert() method when called with the partial success option.

// List to hold event objects to be published.

List<Cloud\_News\_\_e> newsEventList = new List<Cloud\_News\_\_e>();

// Create event objects.

Cloud\_News\_\_e newsEvent1 = new Cloud\_News\_\_e(

Location\_\_c='Mountain City',

Urgent\_\_c=true,

News\_Content\_\_c='Lake Road is closed due to mudslides.');

Cloud\_News\_\_e newsEvent2 = new Cloud\_News\_\_e(

Location\_\_c='Mountain City',

Urgent\_\_c=false,

News\_Content\_\_c='Small incident on Goat Lane causing traffic.');

// Add event objects to the list.

newsEventList.add(newsEvent1);

newsEventList.add(newsEvent2);

// Call method to publish events.

List<Database.SaveResult> results = EventBus.publish(newsEventList);

// Inspect publishing result for each event

for (Database.SaveResult sr : results) {

if (sr.isSuccess()) {

System.debug('Successfully published event.');

} else {

for(Database.Error err : sr.getErrors()) {

System.debug('Error returned: ' +

err.getStatusCode() +

' - ' +

err.getMessage());

}

}

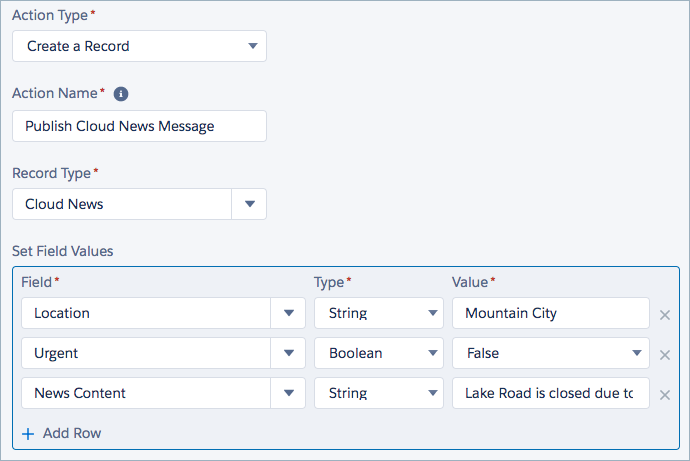
}

The Salesforce platform provides allocations for how many events you can define in your org, and how many events you can publish in an hour. For events configured with the Publish After Commit behavior, each method execution is counted as one DML statement against the Apex DML statement limit. You can check limit usage using the Apex Limits.getDMLStatements() method. For events configured with the Publish Immediately behavior, each method execution is counted against a separate event publishing limit of 150 EventBus.publish() calls. You can check limit usage using the Apex Limits.getPublishImmediateDML() method. For more information see the Resources section.

## Publish Event Messages Using Clicks

To publish event messages without code, use the record creation functionality in a process or flow.

This screenshot shows a sample Create a Record action in Process Builder that publishes a Cloud News event message. Basically, set Record Type to the platform event that you want to publish (Cloud News), then set the values for the event.



Similarly, you can publish a platform event message with a flow. Configure a Create Records element to create an instance of the platform event (Cloud\_News\_\_e), then set the values for the event. To set the value of a Boolean field, such as Urgent\_\_c, use {!$GlobalConstant.True}.

## Publish Event Messages Using Salesforce APIs

External apps use an API to publish platform event messages. You publish events by creating records of your event in the same way that you insert sObjects. You can use Salesforce APIs to create platform event messages, such as SOAP API, REST API, or Bulk API. Like with Apex, the event message is published asynchronously. When you publish an event and get a success status of true, the publish request is queued in Salesforce.

For example, for the Cloud News event, you can publish event notifications by inserting Cloud\_News\_\_e records. The following example creates one event of type Cloud\_News\_\_e in REST API.

sObject REST endpoint:

/services/data/v45.0/sobjects/Cloud\_News\_\_e/

Request body for a POST request:

{

"Location\_\_c" : "Mountain City",

"Urgent\_\_c" : true,

"News\_Content\_\_c" : "Lake Road is closed due to mudslides."

}

After the platform event message is created, the REST response looks like this output. Headers are deleted for brevity.

HTTP/1.1 201 Created

{

"id" : "e00xx000000000B",

"success" : true,

"errors" : [ ],

"warnings" : [ ]

}

You can use any REST API tool or an HTTP client app to make REST API calls. For example, you can use Workbench by following these steps.

1. Log in to your Trailhead DE org.
2. Open a new tab and navigate to Workbench at https://workbench.developerforce.com/login.php
3. For Environment, select **Production**.
4. For API Version, select the highest available number.
5. Select **I agree to the terms of service**.
6. Click **Login with Salesforce**.
7. On the next screen, click **Allow**.
8. In the top menu, select **utilities** | **REST Explorer**.
9. Click **POST**.
10. Replace the URI with:

/services/data/v45.0/sobjects/Cloud\_News\_\_e

For the Request Body, add the following body in JSON format.

{

"Location\_\_c" : "Mountain City",

"Urgent\_\_c" : true,

"News\_Content\_\_c" : "Lake Road is closed due to mudslides."

}

Click **Execute**.The response that Salesforce returns after posting the event looks similar to the following.

{

"id" : "e00xx0000000001AAA",

"success" : true,

"errors" : [ ],

"warnings" : [ ]

}

<https://developer.salesforce.com/docs/atlas.en-us.224.0.api_streaming.meta/api_streaming/using_streaming_api_durability.htm>

<https://developer.salesforce.com/docs/atlas.en-us.224.0.platform_events.meta/platform_events/platform_event_limits.htm>

<https://developer.salesforce.com/docs/atlas.en-us.224.0.apexcode.meta/apexcode/apex_gov_limits.htm>

<https://help.salesforce.com/articleView?id=process_overview.htm&language=en_US>

<https://help.salesforce.com/>

<https://developer.salesforce.com/docs/atlas.en-us.api_rest.meta/api_rest/intro_what_is_rest_api.htm>

<https://developer.salesforce.com/docs/atlas.en-us.api.meta/api/sforce_api_quickstart_intro.htm>

<https://developer.salesforce.com/docs/atlas.en-us.api_asynch.meta/api_asynch/asynch_api_intro.htm>

## Subscribe to Platform Events

Now that you’ve seen how to publish platform events, how do you subscribe to them to be notified of the latest news or of the shipment of a package? On the Salesforce Platform, Apex triggers, processes, flows. The empApi Lightning component and Visualforce apps receive event notifications through CometD. In an external app, you subscribe to events using CometD as well.

## Subscribe to Platform Event Notifications with Apex Triggers

You’ve probably used Apex triggers before, to perform actions based on database events. With platform events, the process is similar. You simply write an after insert Apex trigger on the event object to subscribe to incoming events. Triggers provide an autosubscription mechanism in Apex. No need to explicitly create and listen to a channel. Triggers receive event notifications from various sources—whether they’re published through Apex or APIs.

Platform events support only after insert triggers. The after insert trigger event corresponds to the time after a platform event is published. After an event message is published, the after insert trigger is fired.

To create a platform event trigger, use the Developer Console.

1. Click the Setup icon, select **Developer Console**, and click **File** | **New** | **Apex Trigger**.
2. Provide a name and choose your event for the sObject, and click **Submit**.

The Developer Console automatically adds the after insert event in the trigger template. Also, you can conveniently create a trigger from the event’s definition page in Setup, in the Triggers related list, but you have to specify the after insert keyword.

The following example shows a trigger for the Cloud News event. It iterates through each event and checks whether the news is urgent through the Urgent\_\_c field. If the news is urgent, the trigger creates a case to dispatch a news reporter and adds the event location to the case subject.

Note

Before you run this example, create a queue with a label of Regional Dispatch. To learn how to set up a queue, see [Set Up Queues](https://help.salesforce.com/articleView?id=sf.setting_up_queues.htm&type=5) in Salesforce Help. For more information about the Group object, which represents a queue, see [Group](https://developer.salesforce.com/docs/atlas.en-us.object_reference.meta/object_reference/sforce_api_objects_group.htm) in the Object Reference for Salesforce and Lightning Platform. This example assigns cases to a queue. Queues aren't part of platform events and you don't need to use them to use platform events. Assigning cases to a queue enables distributing cases to a team of support agents that are members of the queue.

// Trigger for listening to Cloud\_News events.

trigger CloudNewsTrigger on Cloud\_News\_\_e (after insert) {

// List to hold all cases to be created.

List<Case> cases = new List<Case>();

// Get queue Id for case owner

Group queue = [SELECT Id FROM Group WHERE Name='Regional Dispatch' AND Type='Queue'];

// Iterate through each notification.

for (Cloud\_News\_\_e event : Trigger.New) {

if (event.Urgent\_\_c == true) {

// Create Case to dispatch new team.

Case cs = new Case();

cs.Priority = 'High';

cs.Subject = 'News team dispatch to ' +

event.Location\_\_c;

cs.OwnerId = queue.Id;

cases.add(cs);

}

}

// Insert all cases corresponding to events received.

insert cases;

}

**Set Up Debug Logging**

Unlike triggers on standard or custom objects, triggers on platform events don’t execute in the same Apex transaction as the one that published the event. The trigger runs in its own process under the Automated Process entity, which is a system user. As a result, debug logs corresponding to the trigger execution are created by the Automated Process entity and aren’t available in the Developer Console. To collect platform event trigger logs, add a trace flag entry for the Automated Process entity in Setup.

1. From Setup, enter Debug Logs in the Quick Find box, then click **Debug Logs**.
2. Click **New**.
3. For Traced Entity Type, select **Automated Process**.
4. Select the start date and expiration date for the logs you want to collect.
5. For Debug Level, enter \* and click **Search**.
6. Select a predefined debug level, such as SFDC\_DevConsole or click **New** to create your own debug level.
7. Click **Save**.

Debug logs for Apex tests are an exception. They include logging for event triggers in the same test execution log.

**Things to Note About Platform Event Triggers**

**Order of Event Processing**  
A trigger processes platform event notifications sequentially in the order they’re received. The order of events is based on the event replay ID. An Apex trigger can receive a batch of events at once. The order of events is preserved within each batch. The events in a batch can originate from one or more publishers.  
  
**Asynchronous Trigger Execution**  
A platform event trigger runs in its own process asynchronously and isn’t part of the transaction that published the event. As a result, there might be a delay between when an event is published and when the trigger processes the event. Don't expect the result of the trigger’s execution to be available immediately after event publishing.  
  
**Automated Process System User**  
Because platform event triggers don’t run under the user who executes them (the running user) but under the Automated Process system user, we set the owner ID field explicitly in our CloudNewsTrigger example. We used the ID of a sample user queue called Regional Dispatch for the trigger example. If you create a Salesforce record with an OwnerId field in the trigger, such as a case or opportunity, explicitly set the owner ID. For cases and leads, you can, alternatively, use assignment rules to set the owner.

Also, system fields of records created or updated in the event trigger, such as CreatedById and LastModifiedById, reference the Automated Process entity. Similarly, the Apex UserInfo.getUserId() statement returns the Automated Process entity.

You can override the running user of a platform event trigger so that the trigger runs under that user instead of Automated Process. Configure the trigger by using PlatformEventSubscriberConfig in Metadata API or Tooling API. For more information, see [Configure the User and Batch Size for Your Platform Event Trigger](https://developer.salesforce.com/docs/atlas.en-us.platform_events.meta/platform_events/platform_events_trigger_config.htm) in the Platform Events Developer Guide.

**Apex Governor Limits**  
Like standard or custom object triggers, platform event triggers are subject to Apex governor limits.  
  
**Apex Trigger Limitations**  
Platform event triggers share many of the same limitations of custom and standard object triggers. For example, you can’t make Apex callouts synchronously from triggers.  
  
**Trigger Batch Size**  
The batch size in a platform event trigger is 2,000 event messages, which is larger than the Salesforce object trigger batch size of 200. The batch size corresponds to the size of the Trigger.New list. You can modify the batch size of a platform event trigger. For more information, see [Configure the User and Batch Size for Your Platform Event Trigger](https://developer.salesforce.com/docs/atlas.en-us.platform_events.meta/platform_events/platform_events_trigger_config.htm) in the Platform Events Developer Guide.

## Subscriptions Related List on the Event Definition Page

You can view the state of all event triggers on the Platform Event Definition Detail page in Setup. Under Subscriptions, each active trigger is listed along with execution information and the state. Information includes the replay ID of the last published and last processed events. The state indicates whether the trigger is running or is disconnected from the subscription because of unrecoverable errors or insufficient permissions. The Error state is reached only when a trigger has been retried the maximum number of times. The following screenshot shows the Subscriptions related list on the Cloud News event detail page.



Note

* The Subscriptions related list also lists flows and processes that are subscribed to the event.
* The Subscriptions related list doesn’t include subscribers that use CometD or the empApi Lightning component. You learn about the other types of subscribers later in this unit.
* For high-volume platform events, the Last Published Id value is not available and is always shown as Not Available.

## Manage an Event’s Apex Trigger Subscribers

Resume a suspended subscription where it left off, starting from the earliest event message that is available in the event bus. If you want to bypass event messages that are causing errors or are no longer needed, you can resume the subscription from the tip, starting from new event messages.

To manage a trigger subscription, in the Subscriptions related list, click **Manage** next to the Apex trigger.

In the subscription detail page, choose the appropriate action.

* To suspend a running subscription, click **Suspend**.  
  
* To resume a suspended subscription, starting from the earliest event message that is available in the event bus, click **Resume**.
* To resume a suspended subscription, starting from new event messages, click **Resume from Tip**.  
  

You can’t manage subscriptions for flows and processes through the Subscriptions related list.

Note

When you save a trigger, the trigger subscription resumes automatically. For more information, see [View and Manage an Event’s Subscribers on the Platform Event’s Detail Page](https://developer.salesforce.com/docs/atlas.en-us.platform_events.meta/platform_events/platform_events_get_subscribers_apex.htm) in the Platform Events Developer Guide.

## Test Platform Event Triggers

Ensure that your platform event trigger is working properly by adding an Apex test. Before you can package or deploy any Apex code (including triggers) to production, your Apex code must have tests. To publish platform events in an Apex test, enclose the publish statements within Test.startTest and Test.stopTest statements.

// Create test events

Test.startTest();

// Publish events

Test.stopTest();

// Perform validation here

In a test context, the publish method call queues up the publish operation. The Test.stopTest() statement causes the event publishing to be carried out. After Test.stopTest(), perform your validations.

Here is an example of a test class for our Cloud\_News event and its associated trigger. Publishing the event causes the associated trigger to fire. After Test.stopTest(), the test verifies that the publishing was successful by inspecting the value returned by isSuccess() in Database.SaveResult. Also, the test queries the case that the trigger created. If the case record is found, the trigger executed successfully, and the test passes.

@isTest

public class PlatformEventTest {

@isTest static void test1() {

// Create test event instance

Cloud\_News\_\_e newsEvent = new Cloud\_News\_\_e(

Location\_\_c='Mountain City',

Urgent\_\_c=true,

News\_Content\_\_c='Test message.');

Test.startTest();

// Call method to publish events

Database.SaveResult sr = EventBus.publish(newsEvent);

Test.stopTest();

// Perform validation here

// Verify that the publish was successful

System.assertEquals(true, sr.isSuccess());

// Check that the case that the trigger created is present.

List<Case> cases = [SELECT Id FROM Case];

// Validate that this case was found.

// There is only one test case in test context.

System.assertEquals(1, cases.size());

}

}

## Subscribe to Platform Event Notifications with a Lightning Component

Lightning apps can use the empApi Lightning Web or Aura component to subscribe to events in the app.

## Subscribe in a Lightning Web Component

To use the empApi methods in your Lightning web component, import the methods from the lightning/empApi module as follows.

import { subscribe, unsubscribe, onError, setDebugFlag, isEmpEnabled }

from 'lightning/empApi';

Then call the imported methods in your JavaScript code.

For an example of how to use the lightning/empApi module and a complete reference, see the [lightning-emp-api documentation](https://developer.salesforce.com/docs/component-library/bundle/lightning-emp-api/documentation) in the Lightning Component Library.

## Subscribe in an Aura Component

To use the empApi methods in your Aura component, add the lightning:empApi component inside your custom component and assign an aura:id attribute to it.

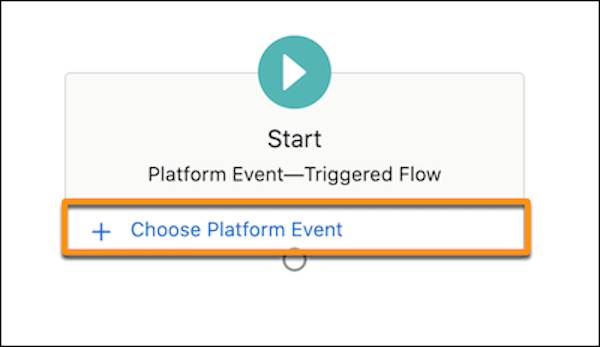
<lightning:empApi aura:id="empApi"/>

Then in the client-side controller, add functions to call the component methods.

For an example of how to use the lightning:empApi component and a complete reference, see the [lightning:empApi documentation](https://developer.salesforce.com/docs/component-library/bundle/lightning:empApi/documentation" \t "_blank) in the Lightning Component Library.

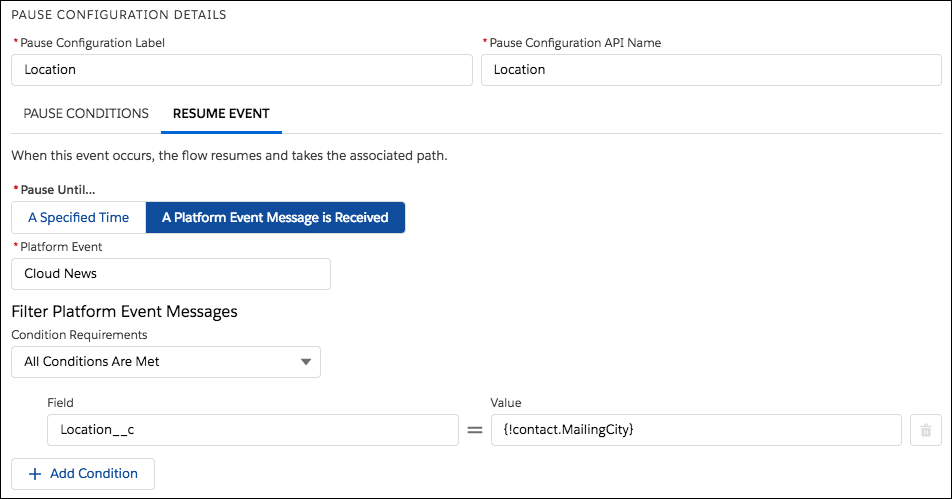
## Subscribe to Platform Event Notifications Using Clicks

To start a flow when a platform event message is received, create a platform event–triggered flow. From the Start element, choose a platform event whose event messages trigger the flow to run.



As you build the flow, you can use the field values from the platform event message by referencing the $Record global variable.

Alternatively, you can subscribe to a platform event in flows by using a Pause element. Instead of starting a flow when a platform event message is received, that event message causes a paused flow interview to resume. For example, here’s a Pause element that pauses the flow until Salesforce receives a Cloud News event message. The flow resumes only if the event’s location matches {!contact.MailingCity}. The {!contact} record variable stores values for a contact record.



## Subscribe to Platform Event Notifications with CometD

External apps subscribe to platform events with CometD and perform long polling. The empApi Lightning component and Visualforce pages, which run on the platform, can use CometD as well and are considered CometD clients. CometD is a scalable HTTP-based event routing bus that uses an AJAX push technology pattern known as Comet. It implements the Bayeux protocol. Long polling, also called Comet programming, allows emulation of an information push from a server to a client. Similar to a normal poll, the client connects and requests information from the server. However, instead of sending an empty response if information isn't available, the server holds the request and waits until information is available (an event occurs).

Salesforce provides a Java library, EMP Connector, which implements all the details of connecting to CometD and listening on a channel. You can use EMP Connector to subscribe easily to platform events. EMP Connector hides the complexity of subscribing to events. For more information about EMP Connector, check out the Java client example in the Streaming API Developer Guide.

The process of subscribing to platform event notifications through CometD is similar to subscribing to PushTopic events or generic events. The only difference is the channel name. The platform event channel name is case-sensitive and is in the following format.

/event/<EventName>\_\_e

For example, if you have a platform event named Cloud News, provide this channel name when subscribing.

/event/Cloud\_News\_\_e

Specify the API version at the end of the CometD URL, as follows.

// Connect to the CometD endpoint

cometd.configure({

url: 'https://<Salesforce\_URL>/cometd/48.0/',

requestHeaders: { Authorization: 'OAuth <Session\_ID>'}

});

## Platform Event Message in JSON Format

The message of a delivered platform event looks similar to the following example for a Cloud News event.

{

"data": {

"schema": "\_2DBiqh-utQNAjUH78FdbQ",

"payload": {

"CreatedDate": "2017-04-27T16:50:40Z",

"CreatedById": "005D0000001cSZs",

"Location\_\_c": "San Francisco",

"Urgent\_\_c": true,

"News\_Content\_\_c": "Large highway is closed due to asteroid collision."

},

"event": {

"replayId": 2

}

},

"channel": "/event/Cloud\_News\_\_e"

}

The schema field in the event message contains the ID of the platform event schema (in this example, it is "schema": "\_2DBiqh-utQNAjUH78FdbQ"). The schema is versioned—when the schema changes, the schema ID changes as well.

To determine if the schema of an event has changed, retrieve the schema through REST API. Use the schema ID by performing a GET request to this REST API resource: /vXX.X/event/eventSchema/Schema\_ID. Alternatively, you can retrieve the event schema by supplying the event name to this endpoint: /vXX.X/sobjects/Platform\_Event\_Name\_\_e/eventSchema. For more information, see the REST API Developer Guide.

Note

Unlike PushTopic and generic events, platform events don’t support using filtered subscriptions. For example, subscribing to /event/Cloud\_News\_\_e?Location\_\_c='San Francisco' to filter by location isn’t supported.

Now that you’ve seen how to use Platform Events on the Salesforce platform and in external apps, the possibilities are endless! Use Platform Events for any number of applications and integrations, such as processing business transactions or engaging in proactive customer service. With Platform Events, you adopt an event-based programming model and enjoy the benefits of event-based software architecture.

<https://trailhead.salesforce.com/en/content/learn/projects/workshop-platform-events>

<https://developer.salesforce.com/docs/atlas.en-us.api_streaming.meta/api_streaming/>

<https://developer.salesforce.com/docs/atlas.en-us.platform_events.meta/platform_events/code_sample_java_client_intro.htm>

<https://docs.cometd.org/>

<https://developer.salesforce.com/docs/atlas.en-us.platform_events.meta/platform_events/platform_event_limits.htm>

<https://developer.salesforce.com/docs/atlas.en-us.platform_events.meta/platform_events/platform_events_intro.htm>

<https://developer.salesforce.com/docs/atlas.en-us.platform_events.meta/platform_events/platform_events_subscribe_apex_refire.htm>

<http://developer.force.com/cookbook/recipe/running-case-assignment-rules-from-apex>

<https://developer.salesforce.com/docs/atlas.en-us.api_rest.meta/api_rest/resources_event_eventschema.htm>

<https://developer.salesforce.com/docs/atlas.en-us.api_rest.meta/api_rest/resources_sobject_eventschema.htm>