Webhook2Flow Administrator's Guide

Use Webhook2flow to connect Salesforce Flows to external system's webhooks
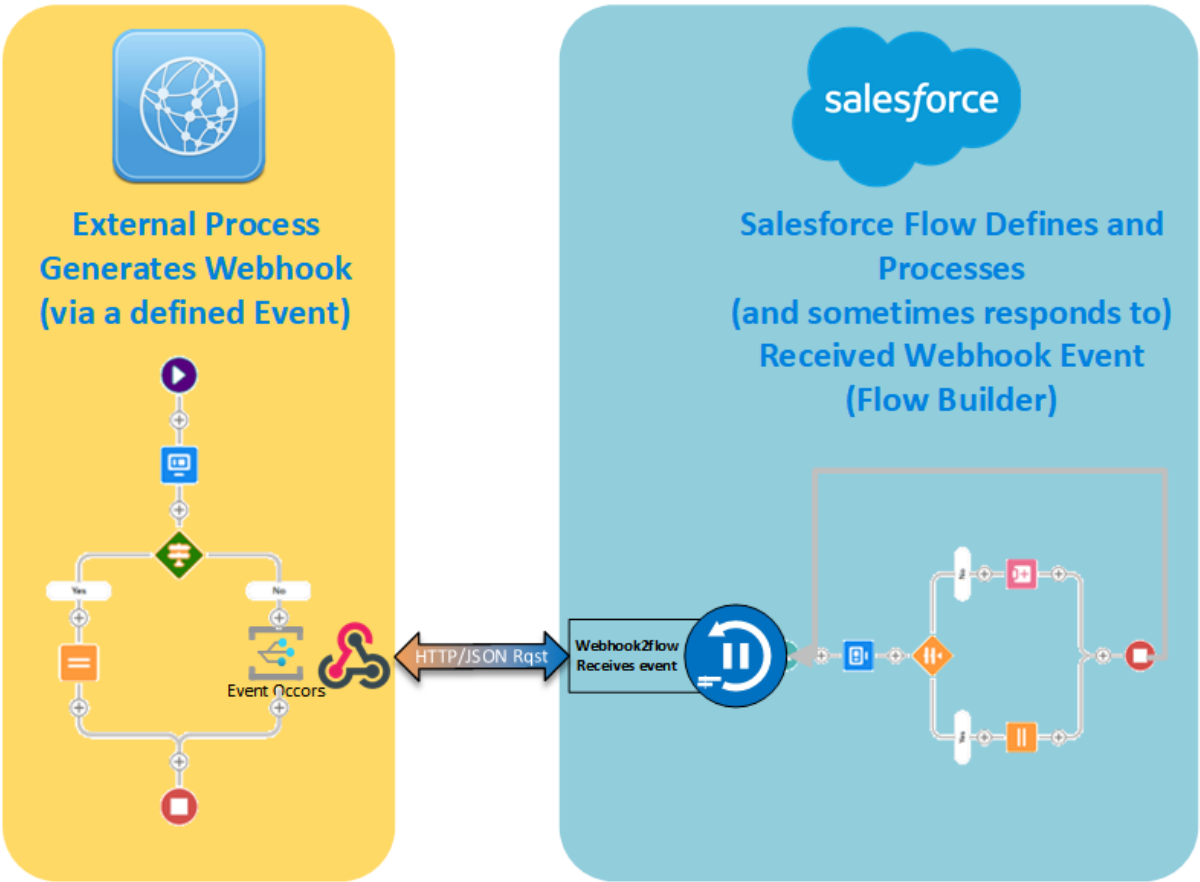
Webhook2flow facilitates webhook input processing by creating the interface and processing in Flow builder:

Webhook Payload	Define Interface in Flow

url_verification

```
{
    "token": "Jhj5dZrVaK7ZwHHjRyZWjbDl",
    "challenge":
"3eZbrw1aBm2rZgRNFdxV2595E9CY3gmdALWMmHkvFXO7tYXAY
M8P",
    "type": "url_verification"
}
```

RESOURCES

∨ Decision Outcomes (1)

⊙⊙ Response_with_Challenge ⟩

∨ Variables (3) ⟸

Aₐ challenge ⟩
Aₐ token ⟩
Aₐ type ⟩

ELEMENTS

∨ Decisions (1)

url_verification ⟩

challenge ✏

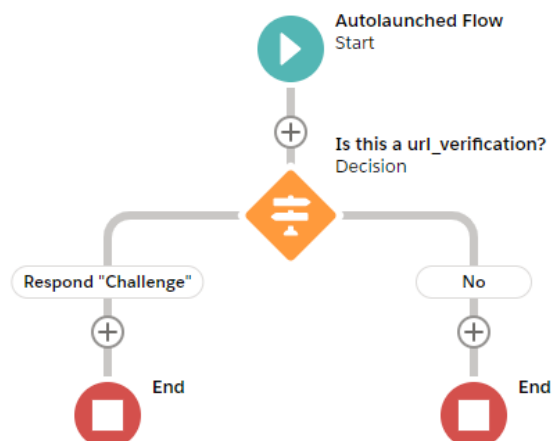* Data Type ⓘ

Text

Default Value

Enter value or search resources...

Availability Outside the Flow

✔ Available for input
✔ Available for output

And use Flow Builder Logic to provide the service for that webhook:

Autolaunched Flow
Start

Is this a url_verification?
Decision

Respond "Challenge"   No

End   End

Webhook2Flow automates the creation and logic of webhook requests generated by services external to Salesforce - almost completely in Flow Builder and without coding.

Webhooks facilitate real-time integration of information and systems through a web (HTTP) request to another system when an event occurs. For example, if a record is created. changed, deleted, . . ., it triggers a event that uses a webhook to communicate that change to your Salesforce system.

Webhooks are usually part of the standard interface extension of most systems that, out of the box, invoke behavior another in system when an event on the originating system occurs.

The way webhooks generally work, an event occurs and the source system makes an HTTP request to the URL configured for the webhook. However, webhooks can be initiated on many systems as a standard part of their workflow or add-ons/extension capabilities. Webhooks2flow can be used with just about any RESTful HTTP Request (see OpenAPI for more information).

OVERVIEW

If you are familiar with the concept of Webhooks and just want to implement using webhooks2flow, just skip to **here**.

Most modern systems have the inherent capability of extending their capabilities through "webhooks", a common API interface using standard Internet protocols and authentications. For example, if I wanted to add a note on a contact every time that contact messaged in slack, I could use the Slack existing system hook "message.im" that posts to a designated Salesforce end point . This is a standard hook for Slack (among dozens of others) which requires no more setup than creating the connection - using a recipient web end point to interpret and act on that request.

Most (if not all) state-of-the-art systems and services allow and utilize webhooks to automate transactions between applications, systems, and services and they are growing exponentially in popularity. . A webhook can be a request for information or a request for an action. Webhooks are increasingly available as extensions to system events and facilitate interaction with one or more external services when these events occur.

In today's world, customers expect a seamless customer experience - no matter if that experience consists of behind the scenes business solutions and services that reside on a single platform or across multiple off-platform hosts. Webhook2Flow is the inverse of Salesforce External Services. External Services smooth the way for this exchange by letting you declaratively (no coding!) integrate with externally hosted services that perform a variety of business actions or computations for use in your Salesforce org. Webhook2Flow allows you to let Other systems integrate with your Salesforce services using existing or created webhooks.

External Services	Webhook2Flow Services
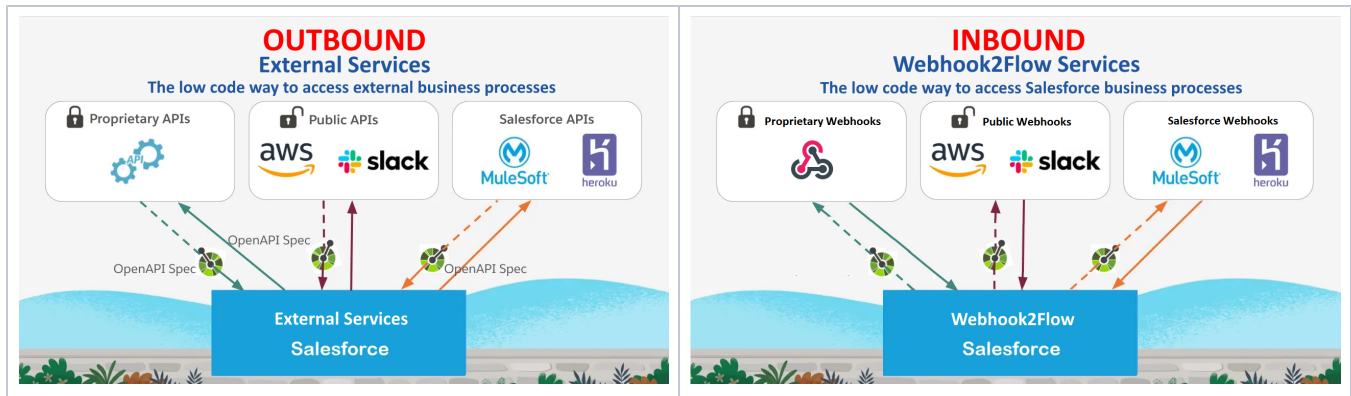
Webhook2Flow exposes a webhook endpoint servicer (as a RESTful Web Service using JSON) entirely through flows and using any of the most common HTTP request interfaces used by those webhooks (POST, but also GET, PUT, PATCH, DELETE).

How Does This Work (The Basics)

Here are the basic steps to create a webhook2flow webhook receptor/servicer on a Salesforce instance.

1. Find (or create) the webhook you want to use from the external system. On most state-of-the art systems, a reference guide exists of all the available webhooks and the payloads/response definitions needed to use them.
2. Create the Webhook2flow Data Interface (in Flow Builder)
   a. Request parameters are defined by marking variables as "Available for Input"
   b. Response parameters are defined by marking variables as "Available for Output"
3. Use Flow Builder to Service the request and build any needed response.
4. Create the Salesforce endpoint to *receive* the webhook request.
   a. Create the Endpoint (Site or Connected App)
   b. Create a Webhook2Flow Endpoint Definition Custom Metadata Record to Define the webhook endpoint instance. The SalesforceWebhook2Flow Endpoint Definition (Custom Metadata Type) record defines the endpoint and makes sure it is only used by the source system you authorize.
5. Use Flow Builder to Provide detailed and supportive Error conditions and helpful diagnostic error messages (if needed).
6. Define custom objects in the webhook definition using apex-defined classes (Advanced feature - use only if required)

When a system event occurs (examples; add a record, perform a query, update a record, a logic or time state change), by using a webhook, that system makes a request to your defined Webhook2Flow instance. This utility allows your Salesforce system to service the requests of external systems directly in Flows without having to go through the hassle of setting up the "web service" (at least on the salesforce side). It does so by making your flow the service handler.

For this example, let's use Slack. Slack has a fairly mature set of webhooks available from a Slack AppCreate a new app-->Event SubscriptionsEnable EventsSubscribe to bot eventsmessage.channels and follow the above cookbook.

We're going to use it to post a note on the senders' contact.

About Authentication and Authorization

It is well beyond the scope of this implementation guide to explore the various mechanisms allowed/required in Salesforce for authentication and authorization of services. In the hated words of our college textbooks, It is "an exercise left for the reader". If you are unfamiliar with it, Enable OAuth Settings for API Integration and Authorization Through Connected Apps and OAuth 2.0 and understanding how to set up public (unauthenticated) access via might be a good place to start.

However, for developers, a quick way to start is to use any of the OpenAPI support tools (e.g., Postman) and create an OATH 2.0 Password Credential and use that to authenticate for development.

Example Implementation - Step by Step for Slack message.channels (with url_verification)

In this example, we will use a very basic POST service provided by Slack. This webhook2flow example will ingest the information sent by the message.channels webhook event via an HTTP POST on the event of a mention (@ Your Salesforce Instance) in Slack, and processed on Salesforce using a flow (Built in Flow Builder) to post a note on the mentioning Contact(User).

Development Tips for this Example:

1. Develop and test the flow first. Information about how the flow failed is difficult to find and use.
2. For debugging purposes, you may want to make the Parameter Variables both Input and Output. If you make them output, the information in them will appear in the body of the HTTP Response - and can help with debugging (if the originating source tracks all http requests and responses)
3. Add a custom field to Contact (SlackUserId__c) that associates the Contact with a Slack User ID, and add a contact with that association

Steps Illustrated in Slack message.channels Example

1. Find (or create) the webhook you want to use from the external system. On most state-of-the art systems, a reference guide exists of all the available webhooks and the payloads/response definitions needed to use them.
2. Create the Webhook2flow Data Interface (in Flow Builder)
    a. Request parameters are defined by marking variables as "Available for Input"
    b. Response parameters are defined by marking variables as "Available for Output"
3. Use Flow Builder to Service the request and build any needed response.
4. Create the Salesforce endpoint to *receive* the webhook request.
5. Create a Webhook2Flow Endpoint Definition Custom Metadata Record to Define the webhook instance. The SalesforceWebhook2Flow Endpoint Definition (Custom Metadata Type) record defines the endpoint and makes sure it is only used by the source system you authorize.

Find (or create) the webhook you want to use from the external system

The webhook for this example we'll use two Slack webooks - url_verification which verifies ownership of an Events API Request URL. This webhook validates the requested webhook endpoint is actually owned by the requestor by sending a **challenge** to that endpoint and expecting it to be returned as a response when the webhook posts the verification request.

The second part of this example will show a more advanced webhook - specifically message.channels which posts a notification that a bot you've set up in Slack as been direct messaged. But more on this later

---

url_verification event

```
{
    "token": "Jhj5dZrVaK7ZwHHjRyZWjbDl",
    "challenge": "3eZbrw1aBm2rZgRNFdxV2595E9CY3gmdALWMmHkvFXO7tYXAYM8P",
    "type": "url_verification"
}
```

This app event allows your app to subscribe to `message` events that directly mention your bot user. It has 6 fields every time it does a request (Via HTTP POST).

Note also in this selection, that event webhook uses **Signing Secret** for security and verification (See Verifying requests from Slack), but more on that later.

Create the Webhook2flow Data Interface (in Flow Builder)

Webhooks usually communicate in JSON data structures (often called payloads). These data structures are defined completely in Flow Builder by defining the variables and designating them as Available for Input for the request payload and Available for Output for the response payload.

For this example, you will actually need to implement two flow paths, he Slack URL VERIFICATION HANDSHAKE (to authorize this endpoint as a valid webhook destination), and the Slack message.channels webhook flow path.

The Slack Payload for URL VERIFICATION HANDSHAKE webhook is:

URL VERIFICATION HANDSHAKE

```
{
    "token": "Jhj5dZrVaK7ZwHHjRyZWjbDl",
    "challenge": "3eZbrw1aBm2rZgRNFdxV2595E9CY3gmdALWMmHkvFXO7tYXAYM8P",
    "type": "url_verification"
}
```

---

and the Slack message.channels, request payload is defined as - but the documentation is wrong. More on this later

---

JSON Request

```
{
    "type": "message.channels",
    "user": "U061F7AUR",
    "text": "<@U0LAN0Z89> is it everything a river should be?",
    "ts": "1515449522.000016",
    "channel": "C0LAN2Q65",
    "event_ts": "1515449522000016"
}
```

Use Flow Builder to Service the request and build any needed response

The input fields will be populated from the request. Use those variables as you would in any flow to fulfill the request For example, in this instance, you may want to create a note on a contact every time a user @ references the Salesforce System in a channel. In this example we actually need two service flows; one to return the challenge necessary to establish the URL as a valid endpoint for Slack, then the second to process the actual message .channels.

REMEMBER: *The Flow must be* ***ACTIVATED*** *before it can be used by webhook2flow.*

Implement the Slack URL VERIFICATION HANDSHAKE flow path

The Slack URL VERIFICATION HANDSHAKE is actually it's own little webhook, and as simple as it gets for logic. This handshake is a post to the URL defined for the webhook event, and the flow logic is as simple as it gets; if the "type" == "url_verification", return the "challenge" to the requestor.

The steps are:

1. Create two new Resources - ResourceType = Variable, DataType=Text, with API names "challenge" and "type". Both of these are selected as "Available for Input" and "challenge" is also "Available for Output".
2. Add a decision checking if the "type" == "url_verification", and if so, done. (You don't have to set the "challenge", because you've already designated it as "Available for Output", so it will be what it was when passed in as "Availabe for Input".)
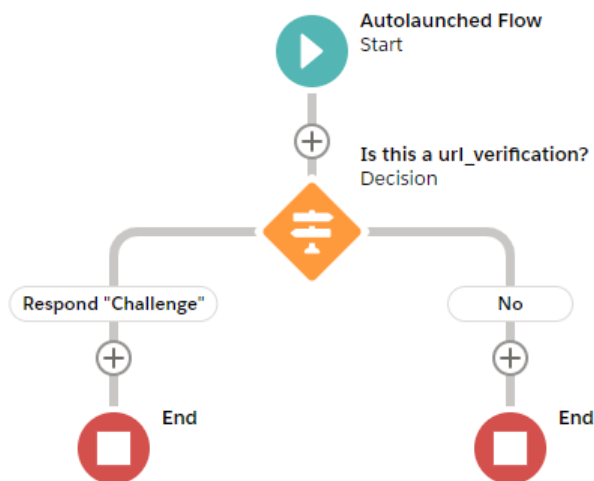
Voilà - you're done with the url_verification flow. If Slack posts to the defined endpoint with the above "url_verification" request, the flow returns the "challenge" value.

Create the Salesforce endpoint to *receive* the webhook request

You will need to create a Custom Metadata WebhookInboundEvent that describes your endpoint and a Site for public access.

The Slack webhook requires a public endpoint with security enforced by:

1. url_verification - Slack send HTTP POST verification request to the URL you give it when events occur. As soon as you enter a URL, it sends a request with a `challenge` parameter, and your endpoint must respond with the challenge value. The easiest way to do this is to create a flow that takes challenge and type as input variables (Available to receive) and responds with challenge (Available as Output). Slack url_verification technical instructions.
2. Secret Signature Message Validation. When you verify the URL, Slack makes available a "Secret" token which you then copy into the Webho okInboundEvent

Steps:

1. Create a public site (using Site). Make sure the Public Access settings are appropriate to allow access to flows, access to the appropriate data, and access to the apex. There are two custom permissionsets included that can be used to assist this:
   a. webhook2workflowEnable
   b. LogServiceEnable
2. Copy the endpoint into Slack (For this example as defined below):

```
https://sandbox-webhook2flowdemo-developer-edition.cs43.force.com/webhook2flowListener/services
/apexrest/v1/WebHookListener/Webhook2Flow/webhook2flow_slack_demo_20210519
```

3. Create an Endpoint Definition Metadata set WebhookInboundEvent. This is an example for Slack. You will have to shut off "Signing Algorithm" until after you've done url_verification - and then add the new Secret once you have verified.

| Inbound Webhook | webhook2flow_slack_demo_20210519 | | Protected Compone | |

Name					nt	
Actor	webhook2flow_slack_demo_20210519				Namespace Prefix	
Host	sandbox-webhook2flowdemo-developer-edition.cs43.force.com				Agent (ApexClass)	Webhook2Flow. WebHook2FlowHandler
Unique Site URL	webhook2flowListener					
Description	Temporary Endpoint Definition to demonstrate webhook2flow using a Slack Mention webhook. This Flow implements the Slack app_mention event webhook event, including the URL VERIFICATION HANDSHAKE. The mention is associated with a contact, then added as a note to that contact.					
Label	webhook2flow_slack_demo_20210519					
Save Logs						

Message Validation

Signing Algorithm	hmacSHA256	Secret	Secret From Slack
Header Signature Parameter	X-Slack-Signature	Signature Prefix	v0=
Payload Concatenation	"v0:";X-Slack-Request-Timestamp;":"		

(Advanced) Implement the Slack message.channels payload interface

First some issues:

1. The documented payload is incorrect - this was discovered by examining the payload actually sent using the logger. The REAL payload follows.
2. This payload actually sent includes Complex Object types (*event, blocks, elements,* and *authorizations*). We're only going to use *event,* but we'll need to create this using an invocable class (the only option currently available)

What we'll be doing is matching the UserID to a created contact

The steps are:

1. Create the resources needed for this webhook (team_id,api_app_id,type, event_id, is_ext_shared_channel,event_context) All of these are selected as "Available for Input".
2. Create the *event* Complex Object as an Invocable Class (see below). After you create it, it will be visible as an Apex-Defined Object. The key fields needed in that object will be **text,** and **user,** and add a note to the Contact identified with the custom field SlackUserId__c.
3. Add the event as a resource
4. Create the logic to associate the message.channels webhook event with an existing contact and add the message to that contact as a Note.

---

app_mention payload

```
{
        "token": "HMomSTnOLtuEpR2hrcUa993H",
        "team_id": "T07GC5A2Z",
        "api_app_id": "A01UY6TE6F8",
        "event": {
                "client_msg_id": "9268eff7-6a56-46f6-b02d-1af33eebd2fa",
                "type": "message.channels",
                "text": "Test 24",
                "user": "U07GBPGJJ",
                "ts": "1619718245.003800",
                "team": "T07GC5A2Z",
                "blocks": [
                        {
                                "type": "rich_text",
                                "block_id": "bFJ",
                                "elements": [
                                        {
                                                "type": "rich_text_section",
                                                "elements": [
                                                        {
                                                                "type": "user",
                                                                "user_id": "U0204UN581F"
                                                        },
                                                        {
```

```
                                              "type": "text",
                                              "text": "Test 24"
                                        }
                                  ]
                            }
                      ]
                }
          ],
          "channel": "C01VD6U6UG3",
          "event_ts": "1619718245.003800"
    },
    "type": "event_callback",
    "event_id": "Ev020FV4A19Q",
    "event_time": 1619718245,
    "authorizations": [
          {
                      "enterprise_id": null,
                      "team_id": "T07GC5A2Z",
                      "user_id": "U0204UN581F",
                      "is_bot": true,
                      "is_enterprise_install": false
          }
    ],
    "is_ext_shared_channel": false,
    "event_context": "1-message.channels-T07GC5A2Z-C01VD6U6UG3"
}
```

Couple of notes on the "REAL" payload for the Slack message.channels webhook event:

1. The above definition is completely wrong.
2. The "flat" level parameters are:
   a. token
   b. team_id
   c. api_app_id
   d. type
   e. event_id
   f. event_time
   g. is_ext_shared_channel
   h. event_context
3. There are 4 complex objects in this payload
   a. event - YOU HAVE TO HAVE THIS ONE. It contains all the real information you are looking for
   b. blocks(collection/array)
   c. elements(collection/array)
   d. authorizations(collection/array)
4. Don't worry about creating variables or invocable variables for any of the ones you don't want/need - if you don't define them, they will just be ignored.

Create (or Extend) the Webhook Payload Interface (incoming)

Sooooo, for this example, we HAVE to create a complex object (event).

We looked at three different ways to do this, but for now, the available option is to create an Apex-Defined Object using an invocable class. The class can be called anything (in this example, it is called slack_event), but many webhooks use the same names for complex objects (example: event, user, . . . appear in webhooks exposed for Slack, Github, Gitlab, Jira). As a best practice, you may want to use the event publisher as a prefix for the complex object. Example: slack_event, github_event, jira_event, . . .

After you have created the invocable class (see Below), it will appear in the drop-down when you've specified the object type as Apex-Defined:

event invocable class

```apex
public inherited sharing class slack_event{

        @invocableVariable(label='type' )
    @AuraEnabled
        public String type;

        @invocableVariable(label='channel' )
    @AuraEnabled
        public String channel;

    @invocableVariable(label='user' )
    @AuraEnabled
        public String user;

    @invocableVariable(label='text' )
    @AuraEnabled
        public String text;

    @invocableVariable(label='ts' )
    @AuraEnabled
        public String ts;

    @invocableVariable(label='team' )
    @AuraEnabled
        public String team;
}
```
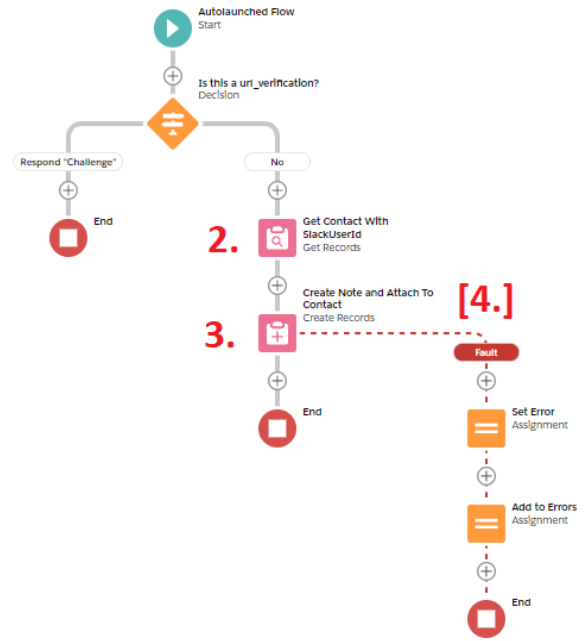
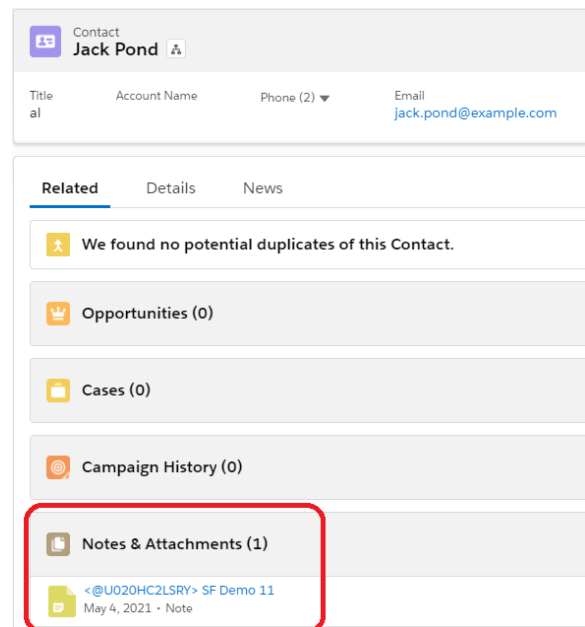## Implement the Slack message.channels servicing logic

Add the logic to service the webhook payload request when received. Illustrated here is:

1. Enable the endpoint to use Secret Signing Validation Inbound Webhook Custom Object instance settings:
   a. Signing Algorithm - hmacSHA256
   b. Header Signature Parameter   X-Slack-Signature
   c. Payload Concatenation   "v0:";X-Slack-Request-Timestamp;":" (this is the prefix definition concatenated with the body to generate a valid signature which will match the contents of header parameter X-Slack-Signature)
      i. "v0:" - a constant defined by Slack (as defined by the double quotes)
      ii. X-Slack-Request-Timestamp - a parameter passed in the request header as specified by Slack
      iii. ":" - a constant (in this case delimiter) before concatenating body (as defined by the double quotes)
   d. Secret   - XXXXXXXXXXXXXXXXXXXXXXX (the 32 character signing secret provided by slack for this webhook event in the Slack API Basic Information App Credentials Signing Secret
2. Lookup the User in Contacts to determine which Contact to apply the mention to as a note
3. Create the note and attach it to the Contact
4. Error processing (optional)
5. Validate that note was added

▼ Message Validation

Signing Algorithm hmacSHA256
Header Signature Parameter X-Slack-Signature   Secret
Payload Concatenation "v0:",X-Slack-Request-Timestamp,":"   Signature Prefix   v0=

Autolaunched Flow
Start

Is this a uri_verification?
Decision

Respond "Challenge"   No

End

2. Get Contact With SlackUserId
Get Records

Create Note and Attach To Contact   **[4.]**
Create Records

3.   Fault

End

Set Error
Assignment

Add to Errors
Assignment

End

And when executed, will attach to Contact

jack 3:10 PM
@SFListener SF Demo 10

jack 3:17 PM
@SFListener SF Demo 11

Message 🔒 test-salesforce

Contact
Jack Pond

Title   Account Name   Phone (2) ▼   Email
al   jack.pond@example.com

Related   Details   News

👤 We found no potential duplicates of this Contact.

👑 Opportunities (0)

🗂 Cases (0)

◎ Campaign History (0)

📄 Notes & Attachments (1)
<@U020HC2LSRY> SF Demo 11
May 4, 2021 · Note

Voilà - you're done with the message.channels flow path. If the Slack message.channels webhook event posts to the defined endpoint with the above message.channels request, the flow adds it to the contact as a note.

Advanced Technical Information and Capabilities

DataType Capabilities and Restrictions

1. Available
   a. Collections (including collections of objects)
   b. sObject (including custom objects)
   c. String
   d. Number
   e. Boolean
   f. DateTime
   g. Date
2. **NOT Available (Yet)**
   a. Null as string (problem with JSON parser)
   b. Blob
   c. Integer

Complex Object Types in Webhook Payloads

As Webhooks have become the de facto standard for systems integration, the complexity of the information has grown, and Complex Objects (Encoding Object) are becoming pervasive. At this time, the only support for this is to define the complex object as an Invocable Class (Apex-Defined). Once defined, variables can be created of the type Apex-Defined in flows. Other alternatives are under exploration and will hopefully be available soon.

Example of Complex Object Definition

```
/**
 * @description      : An invocable class to be used as Apex-Defined for testing webhook2flow (is actually
a Slack 'Event")
 * @author           : Jack D. Pond
 * @group            :
 * @last modified on  : 05-11-2021
 * @last modified by  : Jack D. Pond
 * Modifications Log
 * Ver   Date        Author        Modification
 * 1.0   05-11-2021  Jack D. Pond  Initial Version
**/
public inherited sharing class webhook2flow_apexclass_test{

        @invocableVariable(label='type' )
    @AuraEnabled
        public String type;

        @invocableVariable(label='channel' )
    @AuraEnabled
        public String channel;

    @invocableVariable(label='user' )
    @AuraEnabled
        public String user;

    @invocableVariable(label='text' )
    @AuraEnabled
        public String text;

    @invocableVariable(label='ts' )
    @AuraEnabled
        public String ts;

    @invocableVariable(label='team' )
    @AuraEnabled
        public String team;

}
```

And can be used a Slack Event Object (don't forget to make "Available):

Specifying the HTTP Response StatusCode

By default, Webhook2Flow returns a StatusCode of 200 if successful, or 400 if not. You can override this using a custom parameter (DataType Number) **Webhook2Flow_RestResponse_statusCode** (and making it "Available for output") from your flow,   If defined in the flow and null, it will use the defaults. If set, it will be used as the response.statusCode. The status code you return should be one acceptable to the webhook, but you can check here for a list of common usage.

Exceptions and Error Processing

OK, if you are one of those extraordinarily rare developers who provides meaningful errorsand error messages - even if those users are on different systems (yes, both of you, you know who you are), you can return this information from your flow. This is done by creating a collection of FlowExecutionErrorEvent objects (and making it "Available for output") and adding each error you wish to return. If the collection has no members, it is assumed no error occurred.

Flow Error Example

In this example, adding a record caused an error. This error is returned in the body of the response.

Steps:

1. Create a FlowExecutionErrorEvent record collection and make that collection "Available for output". In this example it was named *errorCollection*
2. Create a FlowExecutionErrorEvent record variable to be used in assignments
3. Introduce Flow logic that detects the error condition (in this example a fault condition from a record insert)
4. Define the error (two fields in the FlowExecutionErrorEvent record variable- ErrorMessage and ErrorId)
5. Add the defined FlowExecutionErrorEvent record variable to the FlowExecutionErrorEvent record collection
6. TEST

Create Records

Fault

End

Set Error
Assignment

Add Error to Error Collection
Assignment

End
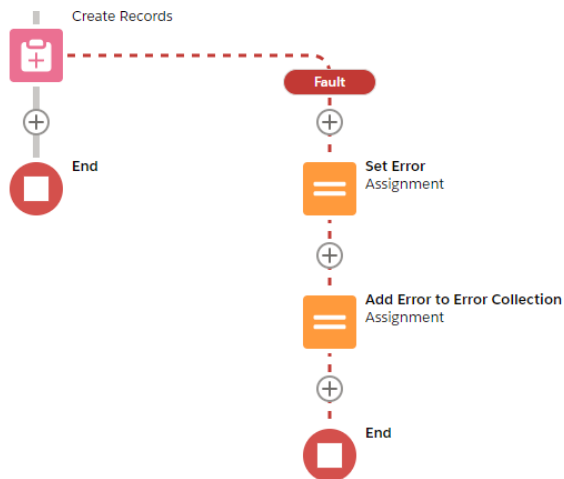
Edit Assignment

Set Error (Set_Error) ✏️

Set the error record

Set Variable Values

Each variable is modified by the operator and value combination.

Variable	Operator	Value	
Aa errorInstance > Flow Error Message ✕	Equals ▼	Aa Fault Message ✕	🗑
Aa errorInstance > Flow Error ID ✕	Equals ▼	CANNOT_INSERT_UPDATE_ACTIVATE_E	🗑

Edit Assignment

Add Error to Error Collection (Add_Error_to_Error_Collection) ✏️

Set Variable Values

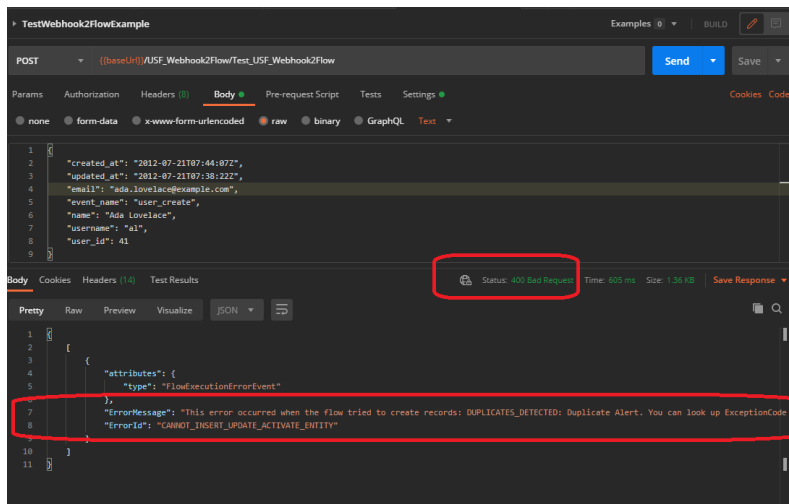Each variable is modified by the operator and value combination.

Variable	Operator	Value
(x) errorCollection ✕	Add ▼	(x) errorInstance ✕

If you're looking for an ErrorId to return, you may want to look at those already defined in your instance at: https://[yourinstance].my.salesforce.com/services/wsdl/tooling under <xsd:simpleType name="StatusCode">.

Endpoint Definition

The endpoint definition serves three functions:

1. Authentication
2. Location (webhook2flow receptor URL)
3. Validation
4. Debug and Auditability

Authentication

An endpoint can be one of two types on Salesforce, Public or Connected app. Public allows for anonymous access while Connected App provides the whole host of authentication options, including the most commonly used secure authentication/authorization OATH2 in its various flows. Most out of the box webhooks use anonymous posts with Signed Secret Verification, but many are more functional AND more secure.

Two options are:

1. **Site** (**Secured by *Signed Secret Verification or Unsecured***). Salesforce sites enables you to create public websites and applications that are directly integrated with your Salesforce.com organization—without requiring users to log in with a username and password. You can publicly expose any information stored in your organization through pages that match the look and feel of your company's brand. Use sites to create public community sites to gather customer feedback, branded login and registration pages for your portals, Web forms for capturing leads, and so on.
2. **Connected App** (**Secured by *SAML, OAuth, and OpenID***). A connected app is a framework that enables an external application to integrate with Salesforce using APIs and standard protocols, such as SAML, OAuth, and OpenID Connect.

Location

The endpoint URL will look like this:

https://**[Your Host][(Optional)Site URL)]**/services/apexrest/**v1/WebHookListener/[your_endpoint_definition_mdt]?parameter1=this+param+value&parameter2=that+param2+value...**

1. **[Your Host] - Of course, you have to know how to get to your instance**
2. **[(Optional)Site URL)] - If you are using a Salesforce Site, it will introduce the Unique Site URL after the host and before the Resource URI.**
3. **v1/WebHookListener/ - This is how you are accessing the utility. It is the defined "@RestResource" Url Mapping that invokes the utility.**
4. **[your_endpoint_definition_mdt] - This is the API name of your flow you wish to execute when the Webhook comes calling.**
5. **[parameters] = e.g., parameter1=this+param+value&parameter2=that+param2+value.... NOTE: YOU SHOULDN'T USE THESE UNLESS YOU HAVE TO!!! We know that some existing webhooks use parameters to define the URI, so they are supported. Body parameters will ALWAYS override URL parameters. This doesn't mean you should use them, unless you have to. URL parameters are not only insecure (the URL is passed open/unencrypted across the internet), they are also subject to man-in-the-middle attacks.**

Example:

Web endpoint URL

```
https://sandbox-webhook2flowdemo-developer-edition.cs43.force.com/webhook2flowListener/services/apexrest/v1/WebHookListener/Webhook2Flow/webhook2flow_slack_demo_20210519
```

This URL is for a publicly accessible Flow endpoint:

1. **https://sandbox-webhook2flowdemo-developer-edition.cs43.force.com/** (host from Site - note ***https***, cut and paste will give you *http* (bad))
2. **webhook2flowListener/** (name of the site, also from Site - may not be needed or used if using Connected App)
3. **services/apexrest/** (standard Salesforce http request prefix)
4. **v1/WebHookListener/** (webhook2flow listener/director)
5. **webhook2flow_slack_demo_20210519** (Developer name of the flow) (host from Site - note https:)

The accessing webhook can use this exact same URL for every HTTP request type (e.g., DELETE, GET, PATCH, **POST**, PUSH). The most commonly used is POST, but this supports them all. For the taxonomy-oriented developers, you can have a single "category" for all of these (eg. [default_flow_APIName]). If you support multiple request types for this function, this utility will automatically look for existing flows of the appended request type (e.g., default_flow_APIName_delete, default_flow_APIName_get, default_flow_APIName_patch, . . .), or you could specify each type specifically through the URL with different names.

NOTE: If you have granted access to https://[Your Host][(Optional)Site URL)]/services/apexrest/**Webhook2Flow**/**[your_flow_API_Name])**, and reference a non-existent **[your_flow_API_Name],** webhook2flow will always return 200 (success) without doing anything, or even telling you that your Flow is not accessible. This is a calculated security decision.

To illustrate the Slack webhook, we'll use the simplest (and most common) access by site. With a Site endpoint, anyone can send a request to the endpoint, but usually there is a way of validating the request - most often using ***Signing Secret*** to create a hash digest with a signature which can be checked by someone only with access to same signature. This is also sometimes referred to as ***HMAC*** validation.

Note: The Site URL will say http: here. ALWAYS change that to **https:** when using it as a reference link. If you do not, Salesforce will automatically change and limit some of your http request types (e.g., POST to GET) and may modify the request body too.

Create a Webhook2Flow Endpoint Definition Custom Metadata Record to Define the webhook instance on Salesforce

Webhook2flow uses Custom Metadata Type ***Inbound Webhook*** to configure and secure the endpoint. Usually you want to know who is making the requests and that the requests are valid and unaltered when you receive them. You also want to make sure only the flow(s) you want to be are accessible as endpoints - and each flow is only accessible by the endpoint you designate.

First, select the ***Inbound Webhook Definition*** Custom Metadata Type, then the ***Manage Inbound Webhook Definitions*** button and ***New***.

Slack has a two-stage authorization of endpoints and validation of requests

1. A challenge to the endpoint to make sure the Slack App developer actually controls/provides that endpoint
2. Signed Secret Verification (HMAC validation) for each message.

For this example, we need two steps in the this definition - a minimum to allow the challenge, then the full Message Validation information once the challenge is accepted.

Inbound Webhook

« Back to List: Custom Object Definitions

Inbound Webhook Detail

Inbound Webhook Name	webhook2flow_slack_demo_20210422	**Protected Component**	☐
Actor	webhook2flow_slack_demo_20210422	**Namespace Prefix**	
Host	sandbox-webhook2flowdemo-developer-edition.cs97.force.com	**Agent (ApexClass)**	Webhook2Flow.WebHook2FlowHandler
Unique Site URL	webhook2flowListener		
Description	Temporary Endpoint Definition to demonstrate webhook2flow using a Slack Mention webhook. This Flow implements the Slack app_mention event webhook event, including the URL VERIFICATION HANDSHAKE. The mention is associated with a contact, then added as a note to that contact.		
Label	webhook2flow_slack_demo_20210422		
Save Logs	✓		

▼ Message Validation

Signing Algorithm	hmacSHA256	**Secret**	████████████
Header Signature Parameter	X-Slack-Signature	**Signature Prefix**	v0=
Payload Concatenation	"v0:";X-Slack-Request-Timestamp;":"		

For the Slack example, the essential fields are:

1. **Inbound Webhook Name** - the unique name for this particular inbound webhook service
2. **Actor** - The name of the servicer (usually the DeveloperName of the Flow) that will handle the inbound webhook requests
3. **Host** - This is the site base address which can be obtained from the base Default Web Address (no HTTP or HTTPS)s. (example: yourdomain.my.salesforce.com)

Default Web Address   http: **sandbox-webhook2flowdemo-developer-edition.cs97.force.com/** **webhook2flowListener** ⓘ
Host Portion   **Unique Site URL**

4. **Unique Site URL** - The unique URL for this site. Salesforce.com provides the first part of the URL; you create the suffix using only alphanumeric characters in the site Default Web Address. This is addition security when multiple webhooks are attached to the same site - if you use it, it will query the site and make sure the Client accessing is the default user for that site. If you create multiple Site endpoints, you should use this parameter for each endpoint.
5. **Description** - always a good administrative practice
6. **Agent(Apex Class)** - The agent that is actually the engine for webhook2flow. Right now this defaults to **Webhook2Flow. WebHook2FlowHandler** and is reserved for future use for more complex handlers (if needed).
7. **Label** - This is a required fiel by Salesforce, but not used for Webhook2flow.
8. **Save Logs** - Webhook2flow has a built in logging capability. This is useful for debugging - and sometimes to discover what the actual payload is (versus what is published). Like any logging tool, it consumes space and presents certain security risks.

   For the Slack example, slack also requires the Request URL to be verified using challenge flow. Slack sends an HTTP POST request with a challenge parameter to the URL you've specified, and your endpoint must respond with the challenge value. Learn more. This means your flow needs a challenge path which returns the challenge parameter. After the challenge has been passed, Slack provides additional information (The Secret) you will use to set up this endpoint.

9. **Signing Algorithm** - The algorithm used to create the hash digest signature (Message Authentication code or MAC) used to validate the message. Currently supported are: hmacMD5, hmacSHA1,hmacSHA256,and hmacSHA512
10. **Header Signature Parameter** - The name of the parameter passed in the request header that contains the Signature (Hashed Digest) as calculated by the sender to be matched with the one calculated by the receiver. For Slack, the parameter is: "X-Slack-Signature"
11. **Secret** - Confirm each request by verifying its unique signature. This is copied directly from the initiating webhook configuration
12. **Signature Prefix** - some webhooks pass the Signature with a prefix. While not used in the Slack example, GitHub uses "sha256=" for digests encrypted as hmacSHA256 and "sha1=" for those using hmacSHA1(GitHub - Securing your webhooks)

    There are other parameters that may be necessary to verify the unique signature. For the Slack example, see Verifying requests from Slack, They include:

13. **Payload Concatenation** - Often used to prevent replay, the payload can be concatenated with both constant and header/parameter information before it is signed. For Slack this is:

   a. **Payload Concatenation**

   ```
   "v0:";X-Slack-Request-Timestamp;":"
   ```

   webhook2flow parses this request as 3 different parameters separated by a semicolon (;). Parameters enclosed in quotes (") are taken as constant values, and there are two of those, "v0:" and ":". Those not enclosed are assumed to be header parameters and are merged as such and there is one of these, X-Slack-Request-Timestamp. The body is always assumed final component. So for the Slack example, this would be a string:

   ```
   V0:[whatever is in the X-Slack-Request-Timestamp header parameter]:[Request Body]
   ```

14. There are additional advanced parameters which will be discussed in the reference section - but are not needed for this example implementation.

Debugging and Auditability (Logs)

webhook2flow has a built-in logging capability to provide both auditability (if required) and a tool for debugging the webhook interfaces - because we all know things never go perfectly the first time. This built-in capability is through a custom Object (LogService__c) which records every request if activated in the WebhookInboundEvent metadata.

> ⓘ **LogService Documentation**
>
> This space reserved for future use - LogService needs to be documented

References

- [Enabling Webhooks to Launch Flows](#)
- [Connected Apps](#)
- [Connected App Use Cases](#)
- [Create a Connected App](#)
- [Identity Basics Trailhead Module](#)
- [Connected AppBasics](#)
- [Build a Connected App for API Integration](#)
- [OAUTH 2.0 Authentication](#)
- [OAuth Authorization Flows](#)
- [OAuth 2.0 Web Server Flow for Web App Integration](#)
- [Create a Connected App for Your Dev Hub Org](#)
- [Defining Connected Apps](#)
- [Digging Deeper into OAuth 2.0 in Salesforce](#)