First-Round Interview Questions:

1. How many Relationship are there in a Salesforce ?

Ans. In Salesforce, there are two main types of relationships:

Master-Detail Relationship: In a master-detail relationship, the detail record belongs to a specific master record. When the master record is deleted, all the detail records associated with it are also deleted.

For example, consider the relationship between the Account and Contact objects. An Account can have many Contacts, but each Contact belongs to a specific Account. If we create a master-detail relationship between Account and Contact, and delete an Account, all the Contacts associated with that Account will also be deleted.

Lookup Relationship: In a lookup relationship, a record in one object is linked to a record in another object, but the linked record is not owned by the master record. When the master record is deleted, the linked record is not deleted.

For example, consider the relationship between the Opportunity and Product objects. An Opportunity can have many Products, but each Product can be associated with multiple Opportunities. If we create a lookup relationship between Opportunity and Product, and delete an Opportunity, the Products associated with that Opportunity will not be deleted.

In addition to these main relationship types, there are also hierarchical relationships, external lookup relationships, and many-to-many relationships. However, these are less commonly used and have more specific use cases.

Hierarchical Relationship: A hierarchical relationship is a special type of relationship used to create a parent-child relationship within a single object. This is commonly used in organizations that have a hierarchical structure, such as in a sales organization with a manager-employee hierarchy.

For example, consider a custom object called "Employee" with a hierarchical relationship field called "Manager" that links each employee to their manager. In this case, each Employee record will have a Manager field that points to the Employee record of their manager. This can be useful for creating reports or workflows that follow the hierarchy.

External Lookup Relationship: An external lookup relationship is a relationship that links a record in a Salesforce object to a record in an external system. This is useful when you want to integrate data from an external system into Salesforce.

For example, consider a custom object called "Order" that has an external lookup relationship with a record in an external system's "Inventory" table. In this case, each Order record will have a field that points to a record in the Inventory table, which can be used to pull in additional data about the product being ordered.

Many-to-Many Relationship: A many-to-many relationship is a relationship between two objects where each object can have multiple records associated with the other object. This requires a junction object to link the two objects together.

For example, consider the relationship between the Account and Contact objects, where a Contact can be associated with multiple Accounts, and an Account can have multiple Contacts. To create a many-to-many relationship between these objects, we would create a junction object called "Account Contact" that has a master-detail relationship with both the Account and Contact objects. Each record in the junction object would link one Contact record with one Account record. This can be useful for creating reports or workflows that involve both objects.

2. Difference between Schedule-Trigger flow and Record-Trigger-Flow ?

Ans. Let's say you have a scenario where you want to automatically update the status of an Opportunity record in Salesforce based on certain criteria. Here's how you might use each type of flow to accomplish this:

Schedule-Triggered Flow:

You might use a Schedule-Triggered Flow if you want to update the status of all Opportunity records that meet certain criteria on a regular basis, such as every day or every week. For example, you might set up a Schedule-Triggered Flow to run every night and update the status of all Opportunity records that have a Close Date that is more than 30 days in the past.

Record-Triggered Flow:

You might use a Record-Triggered Flow if you want to update the status of an Opportunity record as soon as it meets certain criteria, such as when it is first created or when a certain field is updated. For example, you might set up a Record-Triggered Flow that updates the status of an Opportunity record to "Closed - Won" as soon as the Amount field exceeds a certain threshold.

So, the main difference between the two types of flows is the trigger that initiates them. Schedule-Triggered Flows are initiated on a scheduled basis, while Record-Triggered Flows are initiated based on changes to specific records.

3. Tell me one significant difference that Master-Detail Relationship has which is not there in Look-Up relationship?

Ans. One significant difference between Master-Detail Relationship and Look-Up Relationship in Salesforce is that a Master-Detail Relationship has a parent-child relationship between the two objects, while a Look-Up Relationship does not.

In a Master-Detail Relationship, the child object is completely dependent on the parent object, meaning that if the parent record is deleted, all of its related child records will also be deleted. Additionally, the child record cannot exist without a corresponding parent record.

On the other hand, in a Look-Up Relationship, the child object is not dependent on the parent object, and can exist without a corresponding parent record. Additionally, if the parent record is deleted, its related child records will remain in the system and will not be deleted.

Here's an example to help illustrate the difference:

Let's say you have two custom objects in Salesforce: "Invoice" and "Invoice Line Item". Each Invoice record can have multiple related Invoice Line Item records.

If you set up a Master-Detail Relationship between the Invoice and Invoice Line Item objects, each Invoice Line Item record would be completely dependent on its parent Invoice record. This means that if an Invoice record is deleted, all of its related Invoice Line Item records would also be deleted. Additionally, an Invoice Line Item record cannot exist without a corresponding Invoice record.

On the other hand, if you set up a Look-Up Relationship between the Invoice and Invoice Line Item objects, each Invoice Line Item record would not be dependent on its parent Invoice record. This means that if an Invoice record is deleted, its related Invoice Line Item records would remain in the system and would not be deleted. Additionally, an Invoice Line Item record could exist without a corresponding Invoice record.

4. If there are two master-detail relationships associated with a single object, which one would you consider as a original parent object?

Ans. In Salesforce, if an object has two Master-Detail Relationships associated with it, one of the relationships must be designated as the "Master" and the other as the "Detail". The "Master" relationship is the primary relationship, and the "Detail" relationship is the secondary relationship.

The "Master" relationship is used to determine the ownership and sharing behavior of the child records, while the "Detail" relationship is used to establish a parent-child relationship between the objects.

In determining which relationship is the "Master" and which is the "Detail", you would need to consider the business requirements and the relationships between the objects.

Here's an example to help illustrate this:

Let's say you have two custom objects in Salesforce: "Opportunity" and "Quote". Both objects have a Master-Detail Relationship with the Account object.

In this scenario, you would need to consider the business requirements to determine which object is the "Master" and which is the "Detail". If the relationship between Opportunity and Account is more important and plays a more central role in the business process, then you might designate Opportunity as the "Master" and Quote as the "Detail". This would mean that the ownership and sharing behavior of Quote records would be determined by their associated Opportunity record.

On the other hand, if the relationship between Quote and Account is more important and plays a more central role in the business process, then you might designate Quote as the "Master" and Opportunity as the "Detail". This would mean that the ownership and sharing behavior of Opportunity records would be determined by their associated Quote record.

Ultimately, the designation of which relationship is the "Master" and which is the "Detail" depends on the business requirements and the relationships between the objects.

5. On which relationship, rollup summary is not possible? If it is lookup relationship, then how would you create rollup summary field?

Ans. Roll-up summary fields are not possible on Lookup Relationships in Salesforce. This is because Lookup Relationships do not have a hierarchical relationship between the parent and child records. Instead, they have a one-to-many relationship where multiple child records can be related to a single parent record, but the child records do not depend on the parent record for their existence.

However, there are third-party tools and custom development options available that can be used to create roll-up summary functionality for Lookup Relationships in Salesforce. One such option is the "Declarative Lookup Rollup Summaries" (DLRS) package, which is a free, open-source package that can be installed in Salesforce to provide roll-up summary functionality for Lookup Relationships.

Here's an example to help illustrate how DLRS can be used to create a roll-up summary field for a Lookup Relationship:

Let's say you have two custom objects in Salesforce: "Account" and "Opportunity". The Opportunity object has a Lookup Relationship with the Account object, meaning that each Opportunity record is related to a single Account record.

If you want to create a roll-up summary field on the Account object to calculate the total value of all related Opportunity records, you can use the DLRS package to do so. First, you would install the DLRS package in Salesforce and configure it to create a roll-up summary field on the Account object that sums the value of all related Opportunity records.

Once the DLRS package is configured, the roll-up summary field will be automatically updated whenever a new Opportunity record is added, updated, or deleted. This allows you to track the total value of all related Opportunity records on the Account record, even though the relationship between Account and Opportunity is a Lookup Relationship and not a Master-Detail Relationship.

6. What type of events are used to send data from parent to child component in aura? And which type of events are not supported for the same?

Ans. In Aura framework, the events used to send data from parent to child components are called "Component Events" and "Application Events".

Component Events are used for communication between parent and child components that are in the same namespace. They allow the child component to pass data back to the parent component for processing. An example of using Component Events would be to update a value in a parent component based on a user action in a child component.

Application Events, on the other hand, are used for communication between components in different namespaces. They can be used to pass data between unrelated components and are often used for global actions such as logging or triggering a notification. An example of using Application Events would be to trigger an update in multiple components when a specific user action occurs.

Both Component Events and Application Events can be used to send data from parent to child components in Aura framework.

However, the events that are not supported for sending data from parent to child components are "Handling Events". Handling Events are used to notify a component when an event occurs, but they do not pass any data back to the parent component. An example of using Handling Events would be to trigger a notification when a specific user action occurs, but the notification itself does not contain any data that needs to be passed back to the parent component.

In summary, Component Events and Application Events can be used to send data from parent to child components in Aura framework, while Handling Events are not supported for this purpose.

7. An apex class written for inserting 1000 records through a list is called inside a trigger with system.debug(trigger.new.size). What should be in the log?

Assuming that the trigger is set up to fire on the insert of a specific object, and that the Apex class is called within the trigger, the log should show the number "1000" as the output of the System.debug statement.

Here's an example scenario to help illustrate how this works:

Let's say you have a custom object called "My\_Object\_\_c", and you have a trigger set up to fire on the insert of My\_Object\_\_c records. Within the trigger, you call an Apex class that inserts 1000 My\_Object\_\_c records into Salesforce using a list.

Here's what the code for the trigger might look like:

trigger My\_Object\_Trigger on My\_Object\_\_c (after insert) {

MyApexClass.insertRecords(trigger.new);

}

And here's what the code for the Apex class might look like:

public class MyApexClass {

public static void insertRecords(List<My\_Object\_\_c> recordsToInsert) {

List<My\_Object\_\_c> recordsList = new List<My\_Object\_\_c>();

for (Integer i=0; i<1000; i++) {

My\_Object\_\_c record = new My\_Object\_\_c();

record.Name = 'Record ' + i;

recordsList.add(record);

}

insert recordsList;

System.debug(recordsList.size()); // should output 1000

}

}

In this example, when you insert a single My\_Object\_\_c record, the trigger fires and calls the MyApexClass.insertRecords method. This method creates a list of 1000 My\_Object\_\_c records, sets their Names, and inserts them into Salesforce using the "insert" DML operation. Finally, the System.debug statement outputs the size of the recordsList list, which should be 1000.

Therefore, when you check the log, you should see a single debug statement outputting the number 1000. This confirms that the Apex class successfully inserted 1000 records into Salesforce using a list.

8. How will you call an Apex method into LWC file?

Ans. To call an Apex method from a Lightning Web Component (LWC), you need to use the "@wire" decorator to create a wire adapter that calls the Apex method, and then use the "wire" service to retrieve the data from the Apex method and use it in your LWC.

Here's an example of how to call an Apex method from an LWC:

First, create your Apex class with the method you want to call. For example, let's say you have an Apex class called "MyApexClass" with a method called "getAccountList":

public class MyApexClass {

@AuraEnabled(cacheable=true)

public static List<Account> getAccountList() {

return [SELECT Id, Name, Type FROM Account LIMIT 10];

}

}

In your LWC JavaScript file, use the "@wire" decorator to create a wire adapter that calls the Apex method:

import { LightningElement, wire } from 'lwc';

import getAccountList from '@salesforce/apex/MyApexClass.getAccountList';

export default class MyLWC extends LightningElement {

@wire(getAccountList) accountList;

}

In your LWC HTML template, use the "template if:true" directive to display the data returned by the Apex method:

<template>

<template if:true={accountList.data}>

<ul>

<template for:each={accountList.data} for:item="acct">

<li key={acct.Id}>{acct.Name} - {acct.Type}</li>

</template>

</ul>

</template>

</template>

In this example, the "@wire" decorator creates a wire adapter that calls the "getAccountList" method in the "MyApexClass" Apex class. The "wire" service retrieves the data returned by the method and stores it in the "accountList" variable in the LWC. The "template if:true" directive in the HTML template checks if the "accountList" data exists, and if it does, it loops through the list and displays the Account Name and Type in a list item.

By following this pattern, you can easily call Apex methods from your LWCs and use the data in your components.

9. Explain the different Governor Limits Salesforce has?

Ans. Salesforce has several types of governor limits that help maintain the health and performance of the platform. These limits apply to various resources and actions such as Apex code, SOQL queries, workflow rules, and more. The different governor limits in Salesforce are:

Apex Limits: These limits govern the Apex code execution and include CPU time, heap size, number of SOQL queries, number of DML statements, and more.

SOQL Limits: These limits govern the SOQL query execution and include the number of queries executed, the number of records retrieved per query, the number of child-to-parent relationships traversed per query, and more.

DML Limits: These limits govern the DML statement execution and include the number of records processed per transaction, the number of fields updated per record, and more.

Email Limits: These limits govern the email sending and include the number of emails that can be sent per day, the number of email messages that can be sent per day, and more.

API Limits: These limits govern the API requests made to Salesforce and include the number of API calls per 24-hour period, the number of concurrent API requests, and more.

Streaming API Limits: These limits govern the Streaming API requests and include the number of events that can be received per 24-hour period, the number of events that can be processed per minute, and more.

Platform Events Limits: These limits govern the Platform Events and include the number of events that can be published per 24-hour period, the number of events that can be consumed per 24-hour period, and more.

Lightning Platform Limits: These limits govern the use of various Lightning Platform features such as Workflow Rules, Process Builder, and more.

It's important to be aware of these governor limits and design your Salesforce implementation accordingly to avoid hitting these limits, which can cause performance issues or errors in your application. Additionally, there are ways to optimize your code to work within these limits, such as using batch processing, reducing the number of SOQL queries, and more.

10. Future Method in Apex.

In Salesforce Apex, a future method is an asynchronous Apex method that runs in a separate thread, allowing the calling code to continue executing without waiting for the future method to complete. Future methods are useful when you need to perform a long-running operation such as callouts to external systems or processing large volumes of data.

Here is an example of a future method in Apex:

public class MyFutureClass {

@future

public static void doLongRunningOperation() {

// perform a long-running operation, such as a callout or data processing

// this code will run in a separate thread

}

}

In this example, we have a class called MyFutureClass that contains a future method called doLongRunningOperation(). The @future annotation tells Salesforce that this method should be executed asynchronously in a separate thread.

To call this future method from another part of your code, you simply need to invoke it as follows:

MyFutureClass.doLongRunningOperation();

When this line of code is executed, the doLongRunningOperation() method will be queued for execution in a separate thread, allowing the calling code to continue executing without waiting for the future method to complete.

It's important to note that there are some limitations to future methods, such as the inability to access some types of data, and the fact that future methods cannot be called from within a transaction that has already performed a DML operation. Additionally, there are restrictions on the number of future methods that can be executed in a given time period,

Second-Round Interview Questions:

1. How Many Master-Detail Relationships and Look-Up Relationships can an Object have?

Ans. In Salesforce, an object can have multiple master-detail and lookup relationships with other objects, but there are specific limits that apply to the number of each type of relationship that can be created.

A single object can have up to two master-detail relationships and up to 25 lookup relationships. This means that you can define up to two parent-child relationships on an object using the master-detail relationship, and up to 25 relationships where records in the current object can reference records in another object using the lookup relationship.

Here is an example of how this could work in practice:

Let's say we have two custom objects in Salesforce: Account and Opportunity. Account is the parent object and Opportunity is the child object. We can create a master-detail relationship between Account and Opportunity to allow us to link an Opportunity record to a specific Account record. This relationship is known as the "Opportunity - Account" master-detail relationship.

We can also create a lookup relationship between Opportunity and another custom object, such as Product. This lookup relationship allows us to reference a specific Product record when creating an Opportunity. This relationship is known as the "Opportunity - Product" lookup relationship.

In this example, we have one master-detail relationship and one lookup relationship associated with the Opportunity object. We can create up to one more master-detail relationship on Opportunity, as well as up to 24 more lookup relationships, before reaching the limit of two master-detail relationships and 25 lookup relationships per object.

2. Can I able to create a master-detail Relationship on a child object which already have some records?

Ans. Yes, it is possible to create a master-detail relationship on a child object even if it already has records. However, there are some limitations and considerations to keep in mind when doing so.

Firstly, any existing records on the child object will need to have a value for the new master-detail field. If the field is required, you may need to update all existing records to have a value for the field before creating the new relationship.

Additionally, there may be data consistency issues to consider if the child records have different values for the new master-detail field. It's important to ensure that the data is consistent and accurate before creating the new relationship.

Here's an example to illustrate this scenario:

Let's say we have a custom object called "Invoice Line Item" that contains records of individual line items on an invoice. This object currently does not have any relationships with other objects in Salesforce.

Now, we want to create a master-detail relationship between "Invoice Line Item" and another custom object called "Product". This relationship will allow us to link each line item to a specific product in Salesforce.

However, we already have some records on the "Invoice Line Item" object, and we need to ensure that each record has a value for the new master-detail field. We can update the existing records manually or programmatically to ensure they have the correct product values before creating the new relationship.

Once we've updated the existing records, we can create the new master-detail relationship between "Invoice Line Item" and "Product". This will allow us to link each line item to a specific product, and take advantage of the benefits of master-detail relationships such as roll-up summary fields and cascading deletes.

3. Lightning Message Service.

Ans. Lightning Message Service is a feature in Salesforce that enables communication between Lightning web components, Aura components, and Visualforce pages. This feature allows developers to build flexible and reusable components that can be used across different Salesforce applications.

Here's an example of how Lightning Message Service can be used:

Let's say we have two Lightning web components: "Order Form" and "Shopping Cart". The "Order Form" component allows users to input their order details, while the "Shopping Cart" component displays a list of items that the user has added to their cart.

We want to enable communication between these two components so that when a user adds an item to their cart in the "Order Form" component, the "Shopping Cart" component is updated to display the new item.

To accomplish this, we can use Lightning Message Service to send a message from the "Order Form" component to the "Shopping Cart" component whenever a new item is added to the cart.

In the "Order Form" component, we would define the message channel that we want to use to communicate with the "Shopping Cart" component:

import { LightningElement } from 'lwc';

import { publish, MessageContext } from 'lightning/messageService';

import ADD\_ITEM\_CHANNEL from '@salesforce/messageChannel/Add\_Item\_\_c';

export default class OrderForm extends LightningElement {

@wire(MessageContext)

messageContext;

handleAddItem() {

// code to add item to cart goes here

const message = {

item: newItem

};

publish(this.messageContext, ADD\_ITEM\_CHANNEL, message);

}

}

In the "Shopping Cart" component, we would define a message handler that receives the message sent by the "Order Form" component:

import { LightningElement, wire } from 'lwc';

import { subscribe, MessageContext } from 'lightning/messageService';

import ADD\_ITEM\_CHANNEL from '@salesforce/messageChannel/Add\_Item\_\_c';

export default class ShoppingCart extends LightningElement {

@wire(MessageContext)

messageContext;

connectedCallback() {

this.subscription = subscribe(

this.messageContext,

ADD\_ITEM\_CHANNEL,

(message) => this.handleAddItem(message)

);

}

handleAddItem(message) {

// code to add item to cart goes here

}

disconnectedCallback() {

unsubscribe(this.subscription);

this.subscription = null;

}

}

With this code, whenever a user adds an item to their cart in the "Order Form" component, a message will be sent through the "Add\_Item\_\_c" message channel, which the "Shopping Cart" component will receive and use to update the cart display.

Lightning Message Service provides a powerful and flexible way to communicate between components, and can be used in a wide range of use cases across Salesforce applications.

4. How will you communicate between two LWCs components?

There are several ways to communicate between two LWC components, depending on the relationship between the components and the type of data being communicated. Here are a few examples:

Using @api properties and events:

If the two components have a parent-child relationship, the parent component can pass data to the child component using @api properties. The child component can communicate back to the parent component using events. Here's an example:

Parent Component:

<template>

<c-child-component message={message} onchildmessage={handleChildMessage}></c-child-component>

</template>

import { LightningElement } from 'lwc';

export default class ParentComponent extends LightningElement {

message = 'Hello from parent';

handleChildMessage(event) {

console.log('Message from child: ' + event.detail);

}

}

Child Component:

<template>

<h1>{message}</h1>

<button onclick={handleClick}>Send Message to Parent</button>

</template>

import { LightningElement, api } from 'lwc';

export default class ChildComponent extends LightningElement {

@api message;

handleClick() {

const messageEvent = new CustomEvent('childmessage', { detail: 'Hello from child' });

this.dispatchEvent(messageEvent);

}

}

In this example, the parent component passes the "message" property to the child component, which displays it in the template. When the user clicks the button in the child component, the child component dispatches a "childmessage" event with the message "Hello from child". The parent component handles this event with the "handleChildMessage" method, which logs the message to the console.

Using Lightning Message Service:

If the two components are not directly related or have a more complex relationship, Lightning Message Service can be used to facilitate communication. Here's an example:

Component A:

import { LightningElement } from 'lwc';

import { publish, MessageContext } from 'lightning/messageService';

import MY\_MESSAGE\_CHANNEL from '@salesforce/messageChannel/MyMessageChannel\_\_c';

export default class ComponentA extends LightningElement {

handleClick() {

const message = {

value: 'Hello from Component A'

};

publish(this.messageContext, MY\_MESSAGE\_CHANNEL, message);

}

}

Component B:

import { LightningElement, wire } from 'lwc';

import { subscribe, MessageContext } from 'lightning/messageService';

import MY\_MESSAGE\_CHANNEL from '@salesforce/messageChannel/MyMessageChannel\_\_c';

export default class ComponentB extends LightningElement {

@wire(MessageContext)

messageContext;

connectedCallback() {

this.subscription = subscribe(

this.messageContext,

MY\_MESSAGE\_CHANNEL,

(message) => this.handleMessage(message)

);

}

handleMessage(message) {

console.log('Message received: ' + message.value);

}

disconnectedCallback() {

unsubscribe(this.subscription);

this.subscription = null;

}

}

In this example, Component A publishes a message on the "MyMessageChannel\_\_c" message channel when the user clicks a button. The message contains a "value" property with the string "Hello from Component A". Component B subscribes to this message channel in its "connectedCallback" method, and handles incoming messages with the "handleMessage" method, which logs the message value to the console.

These are just a few examples of how LWC components can communicate with each other. There are many other techniques and patterns for data sharing and communication in LWC, depending on the specific requirements of your application.

5. With which Process Automation tool you will go with WorkFlow Rules or Process Builder or Flow Builder?

Ans. Salesforce offers multiple process automation tools to automate business processes. The choice of tool depends on the complexity of the process and the business requirements.

Workflow Rules:

Workflow rules are simple and straightforward to use. They are best suited for simple processes that require basic actions like updating a field or sending an email. They are very easy to set up and require no coding skills. For example, you can create a workflow rule that updates the status of an opportunity to "Closed Won" when the opportunity stage is updated to "Closed Won."

Process Builder:

Process Builder is more advanced than Workflow Rules and allows for more complex processes to be built. They allow you to automate processes that require multiple steps or involve multiple objects. For example, you can create a process that creates a follow-up task when an opportunity is closed won and assigns it to the sales representative responsible for the opportunity.

Flow Builder:

Flow Builder is the most powerful process automation tool available in Salesforce. It allows you to create custom, guided user experiences that automate complex business processes. Flow Builder offers the most flexibility when it comes to automating business processes. For example, you can create a flow that guides a user through a multi-step process for creating a new account, including validation and approval steps.

In summary, Workflow Rules are best for simple processes with basic actions, Process Builder is best for automating more complex processes involving multiple steps and objects, and Flow Builder is best for creating custom, guided user experiences for complex business processes. The choice of tool depends on the complexity of the process and the business requirements.

6. How will you make an Apex method accessible to Flow, Process Builder ?

Ans. To make an Apex method accessible to Flow or Process Builder, we need to create a global Apex class and define the method as a global static method. Here's an example:

global class AccountUtils {

@InvocableMethod

global static List<Account> getAccounts(List<Id> accountIds) {

return [SELECT Id, Name, Industry FROM Account WHERE Id IN :accountIds];

}

}

In the above example, we have created a global Apex class called AccountUtils and defined a method called getAccounts. This method takes a list of Account Ids as input and returns a list of Accounts.

We have also used the @InvocableMethod annotation to make this method available to Flow and Process Builder. This annotation is used to mark a method as invocable by Flow or Process Builder.

Once we have created this class, we can use this method in Flow or Process Builder by creating an Apex Action. Here's an example of how to use this method in Flow:

Create a new Flow and add a Get Records element to retrieve a list of Account Ids.

Add an Apex Action element to the flow.

Select the AccountUtils class and the getAccounts method.

Map the Account Ids to the method parameter.

Add a new variable to store the output of the method.

Use the output variable in the flow as needed.

By using this global Apex class and making the method available to Flow and Process Builder, we can leverage the power of Apex code in our process automation without needing to write complex Apex code within Flow or Process Builder.