# Einstein Bots Developer Cookbook

Version 54.0, Spring '22


@salesforcedocs

Last updated: April 8, 2022

# CONTENTS

# CHAPTER 1 Einstein Bots Developer Cookbook

With Einstein Bots taking over the simple tasks during a chat or messaging session, your agents can focus more on complex issues or value-added activities. This cookbook covers some common use cases when building a bot that works for you and your customers.



## Get Started with Bots

Use these recipes to get started with your bot.



### Set Up Your First Einstein Bot

Let's set up a basic bot that we can use throughout this cookbook.

1. Enable Chat. Before you can use Einstein Bots, you must enable Chat in your org. From Setup, enter **Chat** in the Quick Find box, then select **Chat Settings**. Check **Enable Chat**.

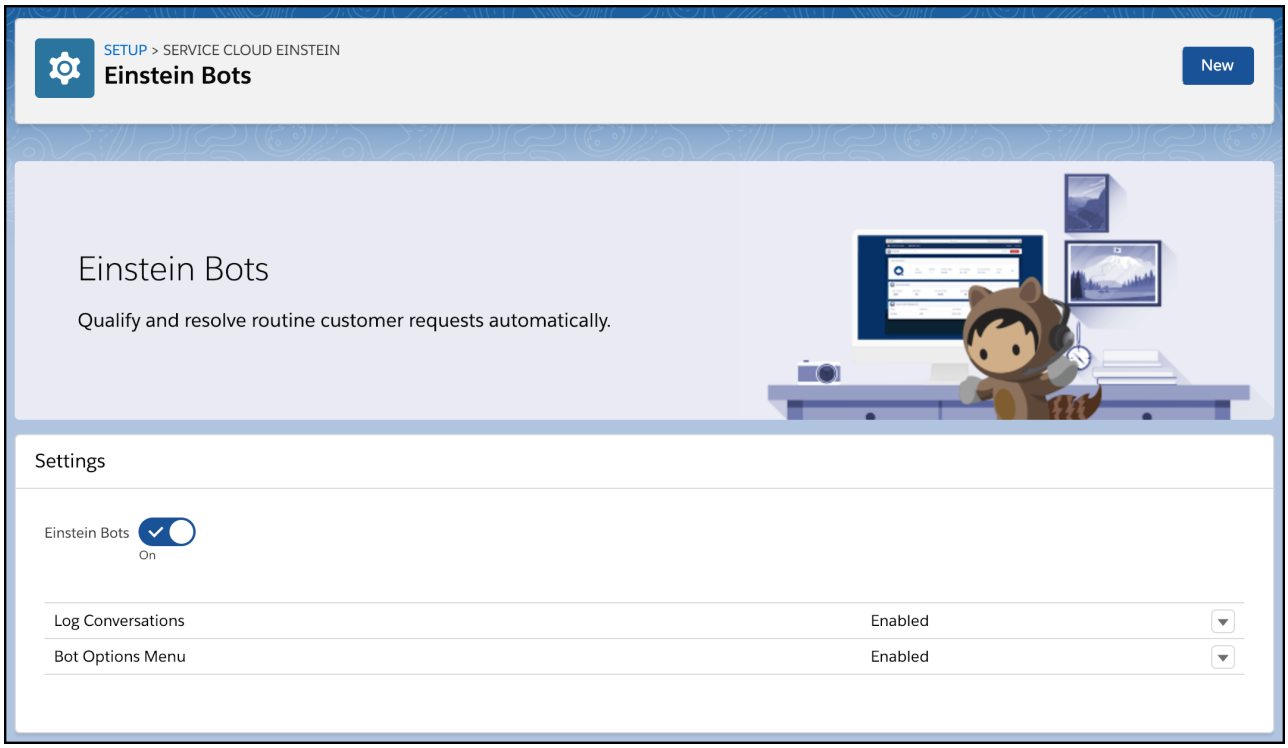
   Note: Even if you don't plan on using Chat, you still need to enable it in order to use Einstein Bots.

2. Implement Chat and/or Messaging. To use your bot, you must connect it to a Chat implementation, a Messaging channel, or both!

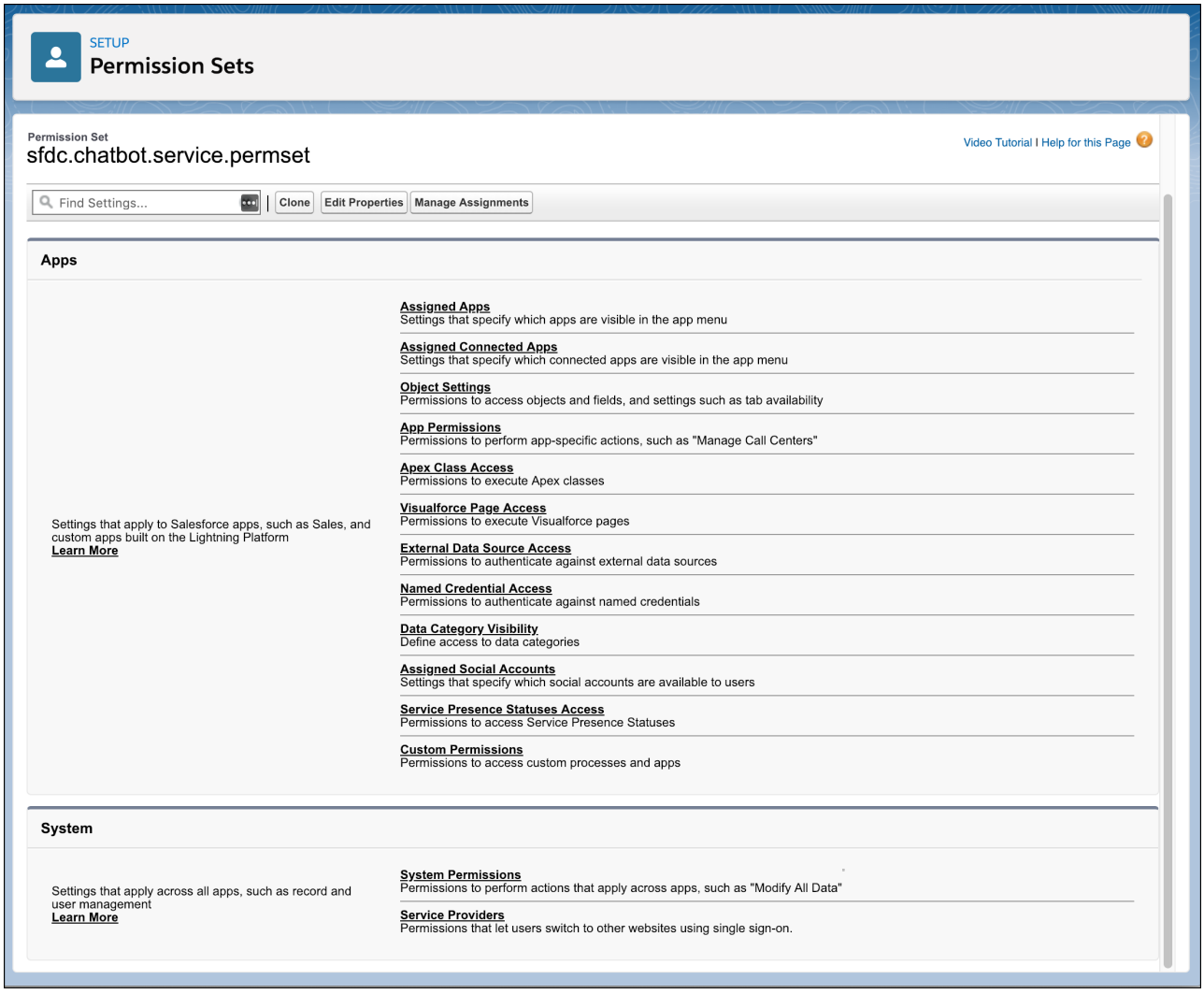
- For Lightning Chat, use the "Get Started with Web Chat" unit in the Trailhead module Web Chat Basics.
- For Classic Chat, follow the instructions in Create a Basic Chat Implementation.
- For Messaging, go through our guided setup flow. To learn more, see Set Up Messaging with a Guided Setup Flow. Alternatively, you can manually Set Up a Messaging Channel.

3. From Setup, use the Quick Find box to find **Einstein Bots**.
4. Enable Einstein Bots.

   Set the **Einstein Bots** switch to **On**. If you're enabling the bot for the first time, you are asked to accept the terms.




As part of setup, the permission set **sfdc.chatbot.service.permset** is added to your org. This permission set controls which objects and Apex classes bots can access. To build a simple bot, you don't have to change anything in the permission set. As you add more features to the bot, edit the permission set to expand access. For instance, you can grant access to Apex classes from **Apex Class Access**; you can grant access to Salesforce objects through **Object Settings**; you can enable access to flows from the **Run Flows** checkbox in the **System Permissions** section.

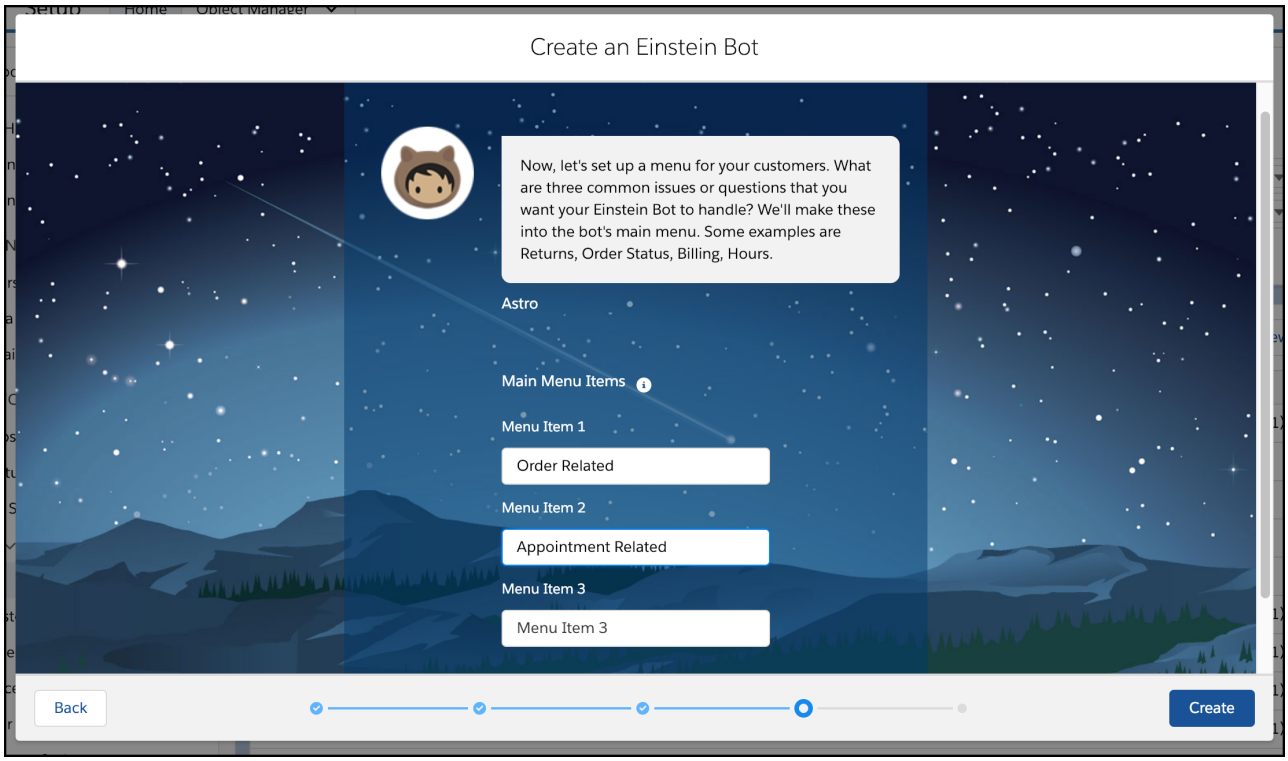



5. To create a bot, click **New** on the Einstein Bots setup page.

   When prompted, provide basic information about the bot, including a name, welcome message, and main menu options.

   Speaking of menus, it's a good practice to provide your customers with a clear picture of what the bot can and can't do. We recommend that you focus on the top inquiry types. If you provide these options in a menu, customers don't have to guess what the bot supports or how to ask for help. Also, provide a transfer to agent option. If customers don't see what they need, they can get help quickly.

   For now, let's add two main menu options: `Order Related` and `Appointment Related`. Then click **Create** and **Finish**.



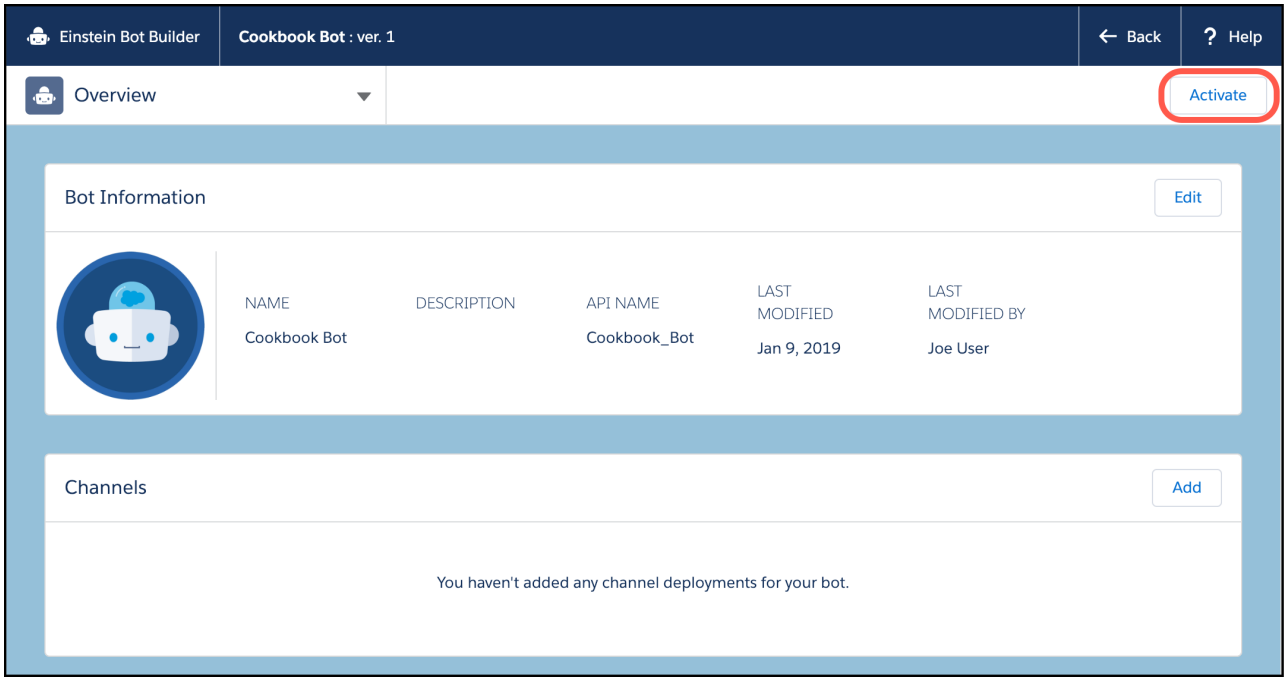

When your new bot is ready, you're taken to the Bot Builder **Overview** page. To get oriented, explore the **Overview**, **Dialogs**, **Entities**, **Variables**, and **Performance** pages.

To learn more about each page within the Bot Builder, use the following Help articles:

- Overview: Create a Basic Bot
- Dialogs: Understand Einstein Bot Dialogs
- Entities: What's an Entity?
- Variables: What's a Variable?
- Performance: Monitor, Analyze, and Refine Bot Activity

6. Activate your bot.

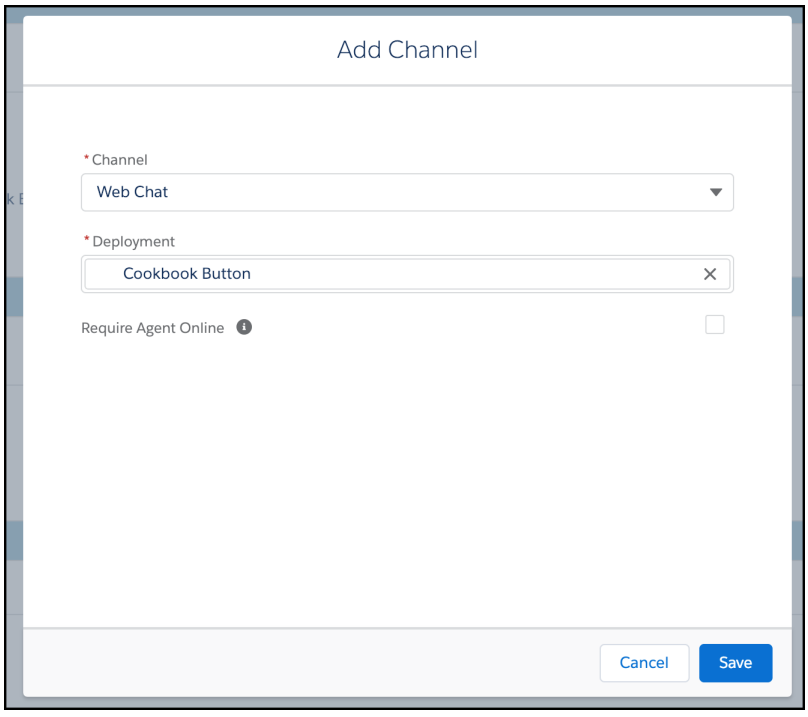
   To activate your bot, click **Activate**.




7. Add a channel for your bot.

   From the **Overview** page of the Bot Builder, in the **Channels** section, click **Add**. If you want to connect your bot to your chat implementation, select **Web Chat** for the channel and select the chat deployment you created for your bot. If you want to connect your bot to your Messaging implementation, select the messaging channel type (for example, Text), and then select the channel name as the deployment.

   Now, instead of going directly to an agent, users are first connected to your bot.



If you always want agents ready for transfer when the bot can't answer an inquiry or the customer wants help from an agent, select **Require Agent Online**. If your agents work limited hours and you want the bot working 24x7, don't select this option.

For now, leave this setting deselected so that we can test the bot without setting up the agent side.

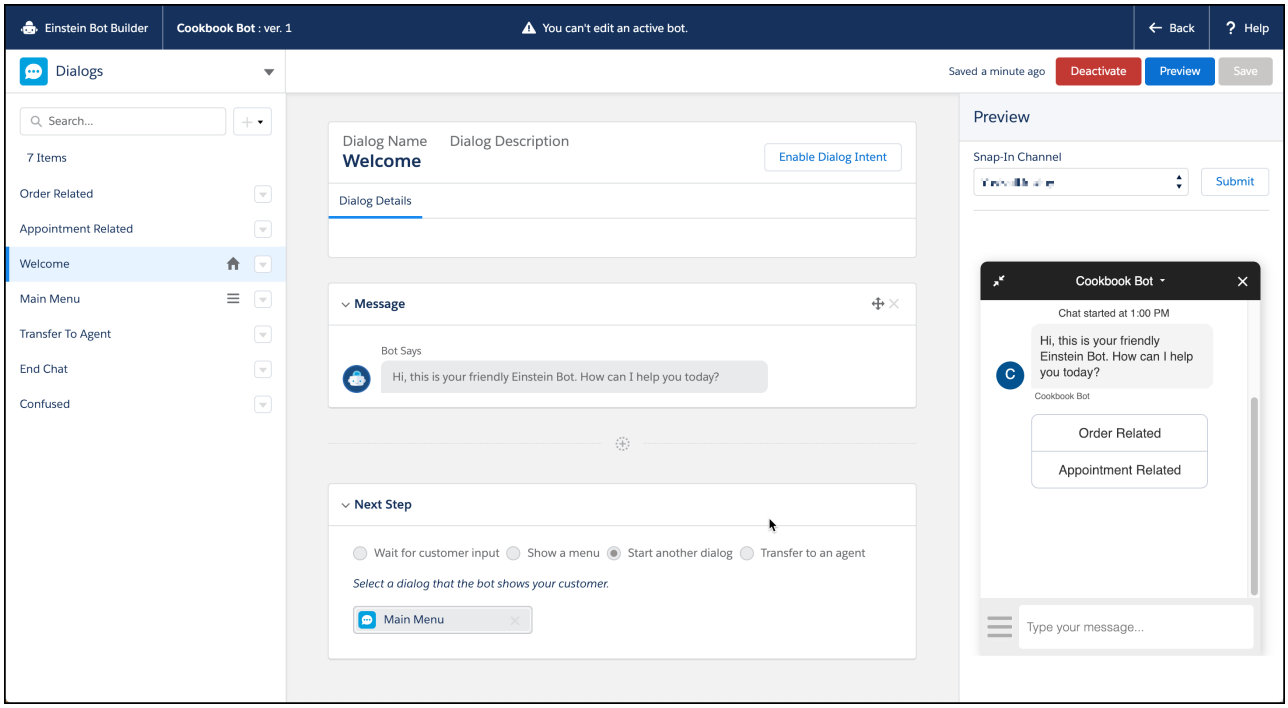
8. To preview the bot from within the Bot Builder, add an Embedded Chat deployment. .

   For preview, Einstein Bots uses an Embedded Chat component. Even if you use Classic Live Agent for your customer chat interface, create a chat deployment.

   Note: Enable Lightning Experience to set up Embedded Service.

9. Preview your bot. If you created a chat deployment in the previous step, you can preview your bot by going to the Dialog page of Bot Builder and clicking **Preview**. Alternatively, you can test your bot simply by using one of the channels you assigned to your bot.

   .




Your Einstein Bot welcomes you with a greeting message and your two menu options. If you set up the bot using Embedded Chat, you may also see a pre-chat form before the bot sends its first message.

Note: To preview your bot, you must first activate it.

After you're able to test your bot, you can take on the other recipes in this cookbook.

# Greet the Customer

In this recipe, we learn how to use information we know about a customer to add more smarts to the bot welcome message.

For the basic bot, we added a welcome message. But wouldn't it be nice for the bot to send a personal greeting with the customer's name? If we gather some basic information before the conversation starts, we can use it to personalize the session.

## Greet the Customer with Embedded Service Chat

If you're using Embedded Service Chat, follow these steps to greet the customer.

**Prerequisites**

- Set Up Your First Einstein Bot on page 1

**In this recipe you learn how to:**

- Use a pre-chat form to get data from the customer.
- Upload JavaScript to static resources to map fields in the **Contact** object with fields in the **LiveChatTranscript** object.
- Inspect fields in the **LiveChatTranscript** object.
- Access context variables with the Einstein Bot Builder.
- Use Apex code or the Metadata API to get data into a new variable.

1. Create custom fields on the `LiveChatTranscript` object to give the bot access to customer information.

   In the Service Cloud Console, agents can view all chat fields on the Chat Details tab. Admins can use the Console Integration Toolkit API to give agents even more information. Because bots don't have a view of an agent's console, the data must be committed to a platform object. Commitment is necessary for giving the bot access to information programmatically. Bots can use the `LiveChatTranscript` object to access pre-chat variables.

   To capture the first name, last name, and email address data from the pre-chat form, create three fields on the `LiveChatTranscript` object.

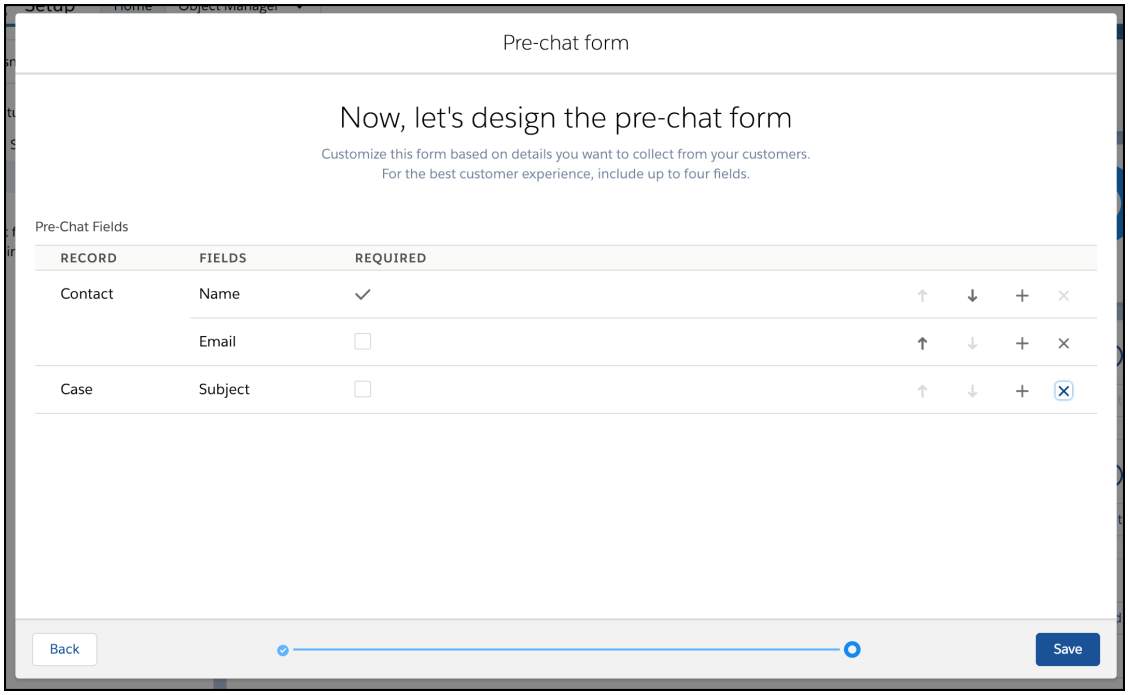
| | | |
|---|---|---|
| Email | Email__c | Email |
| End Time | EndTime | Date/Time |
| Ended By | EndedBy | Picklist |
| FirstName | FirstName__c | Text(80) |
| Last Modified By | LastModifiedById | Lookup(User) |
| Last Modified Date | LastModifiedDate | Date/Time |
| LastName | LastName__c | Text(80) |

2. Set Up Omni-Channel Routing

   In chatbot implementations, Omni-Channel routing is preferred because the bot requires access to the `LiveChatTranscript` record, which is created first in Omni-Channel. To learn more about setting up Omni-Channel, visit our help topics about Omni-Channel. After you've set up Omni-Channel, change the routing type for your chat button. From Setup, use the Quick Find box to find **Chat Buttons & Invitations**. Update the **Routing Information** section.

3. Enable the pre-chat page in your chat configuration.

From Setup, use the Quick Find box to find **Embedded Service**. Click **View** to access your Embedded Service deployment, then click **Edit** next to **Chat Settings**. Enable the **Pre-chat page** if it has not been done already. By default, customer name and email are included on the pre-chat form. If you have a Webmail field on the pre-chat form, click the **X** icon to remove it. and click **Save**.




4. Populate the `LiveChatTranscript` record with pre-chat data.

   With the pre-chat configuration, admins can select fields from the two objects on the Pre-chat form: contact and case. However, the built-in Bot Builder variables give you access only to the `LiveChatTranscript` record. To bridge this gap, you have a couple of approaches. You can write code to get data from the contact and case records, or you can populate custom fields on the `LiveChatTranscript` record and then turn those fields into bot variables. Let's go with the second approach.

   The process for populating pre-chat data in the `LiveChatTranscript` record is slightly different depending on whether you're using a customer website or an Experience Cloud site.

   a. (Option 1 of 2) Add Embedded Chat to an Experience Cloud site.

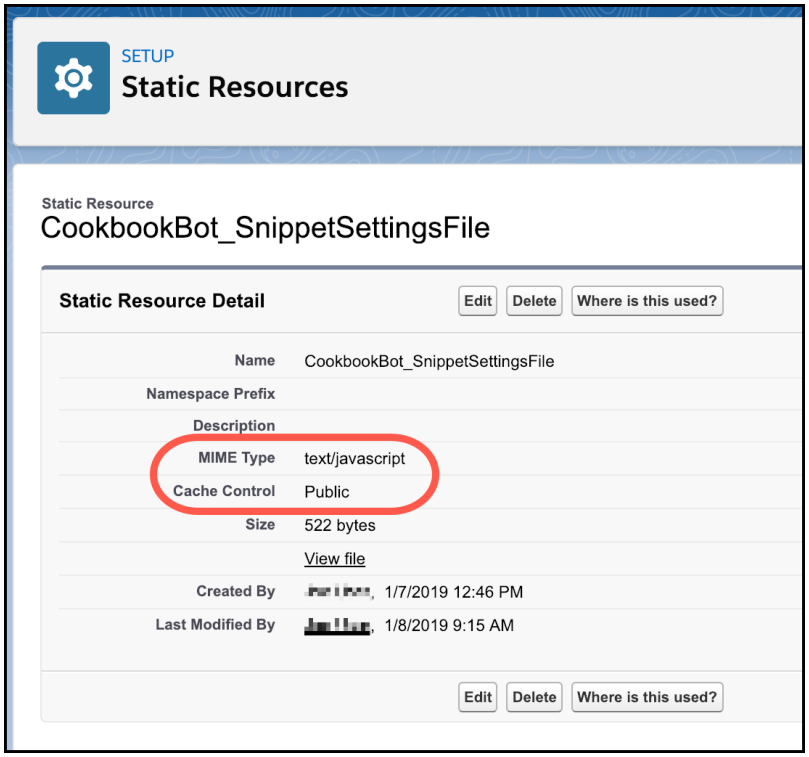
      Implementing Embedded Chat in an Experience Cloud site is easy because you use a standard Embedded Chat Lightning component. Drag the component onto the site page or template from Experience Builder.

      Create a JavaScript file locally named *`CookbookBot_SnippetSettingsFile.js`*. Notice that the `extraPreChatFormDetails` array is almost identical to what we use for an external site.

```
window._snapinsSnippetSettingsFile = (function() {

  // Logs that the snippet settings file was loaded successfully
  //console.log("Snippet settings file loaded.");

  embedded_svc.snippetSettingsFile.extraPrechatFormDetails = [{
    "label":"First Name",
    "transcriptFields": ["FirstName__c"]
  },{
    "label":"Last Name",
    "transcriptFields": ["LastName__c"]
  },{
```

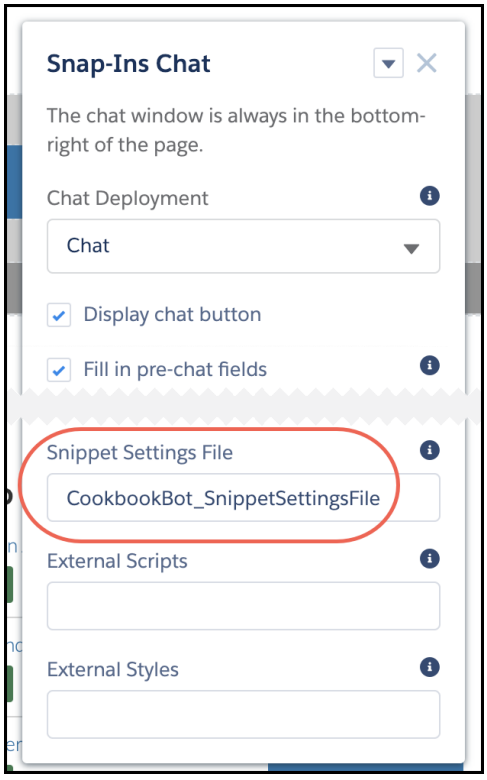
```
    "label":"Email",
    "transcriptFields": ["Email__c"]
  }];
})();
```

Upload the JavaScript file to the org as a static resource. From Setup, use the Quick Find box to find **Static Resources**. Name the resource *`CookbookBot_SnippetSettingsFile`*. Verify that the MIME Type is set to `text/javascript` and that **Cache Control** is **Public**. Some text applications can add extra text which can cause problems down the road. To ensure that your file matches the code sample, click **View file**.




Now, in Experience Builder, you can set the `Snippet Settings File` field in your chat window to your static resource.

In Experience Builder, add your static resource to `Snippet Settings File`.




**Important:** Enter the static resource name (not the filename) in the **Snippet Settings File** field. For example, if your JavaScript file is called `CookbookBot_SnippetSettingsFile.js` and you named it `CookbookBot_SnippetSettingsFile` in your static resources, enter *`CookbookBot_SnippetSettingsFile`* in the field.

Publish your Experience Cloud site after this change. If you run into other errors or messages from the site while publishing, follow the instructions inside the site to resolve.

**b.** (Option 2 of 2) Add Embedded Chat to a customer website.

This method involves embedding the code snippet generated from your Embedded deployment in your website.

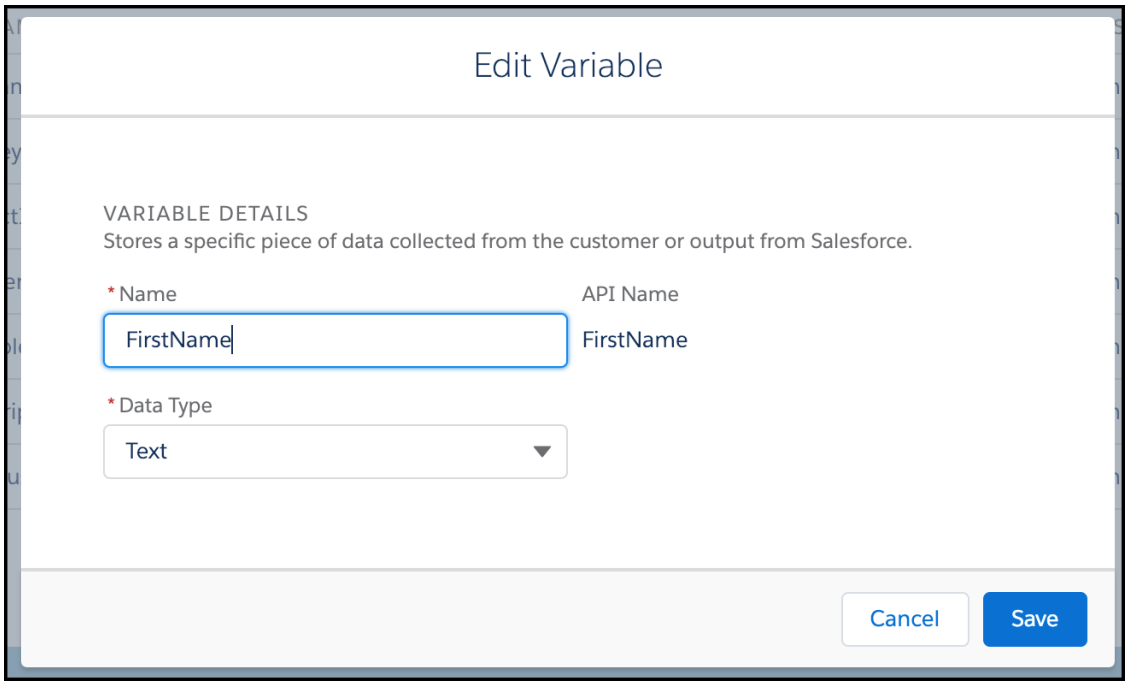
Add the following code to your code snippet before the `embedded_svc.init()` method. Then embed the code snippet in your website

```
embedded_svc.settings.extraPrechatFormDetails = [{
  "label":"First Name",
  "transcriptFields": ["FirstName__c"]
},{
  "label":"Last Name",
  "transcriptFields": ["LastName__c"]
},{
  "label":"Email",
  "transcriptFields": ["Email__c"]
}];
```

`embedded_svc.settings.extraPrechatFormDetails` sets the default value and adds information to the chat transcript. The `transcriptFields` attribute specifies the field API name on the `LiveChatTranscript` object where the data is saved.

5. Deactivate the bot and create a variable for first name.

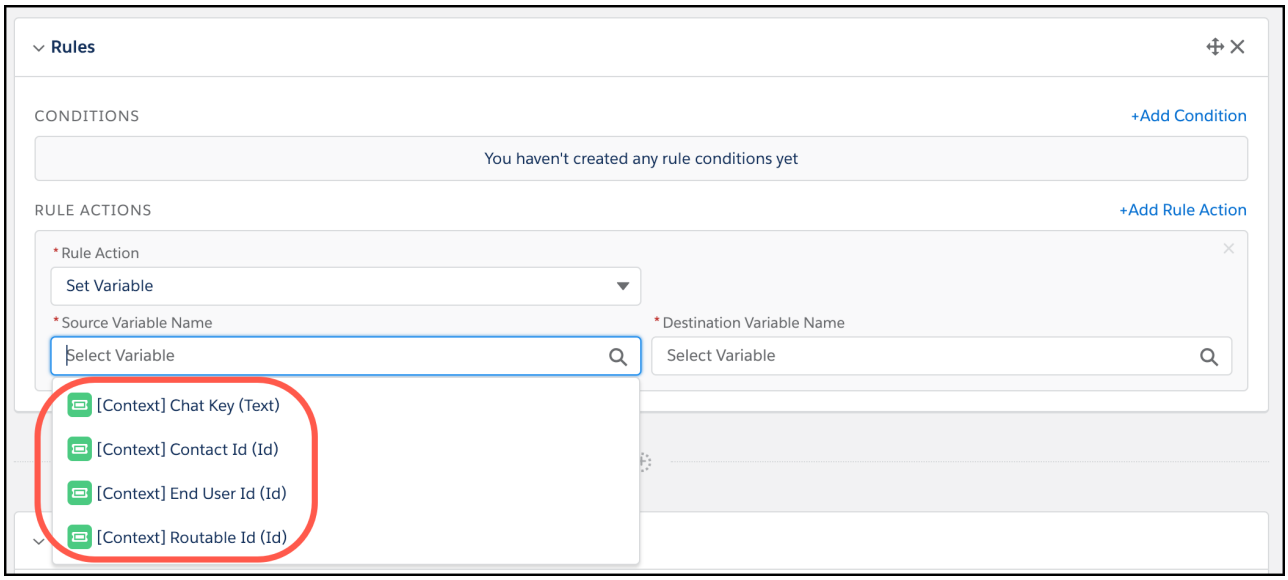
   A bot variable stores and passes data between different steps in a bot session. In Bot Builder, on the **Variables** tab, create a variable with the API name `FirstName` and with a data type of **Text**.




   We can pass the pre-chat data to a variable, and use the variable to merge the customer name into the greeting message.

6. We now want to extract the value of the `FirstName__c` field from the `LiveChatTranscript` object, and get it into the `FirstName` variable. You can get the `FirstName__c` value in two different ways, but first it's important to understand context variables.

   Let's say you create a **Rule** in a bot flow and specify a **Set Variable** rule action. You can choose from a set of built-in variables to access objects and fields related to the chat session.




   The values of these context variables depend on how you set up your bot.

**Table 1: Context Variable Definitions**

| Context Variable | Chat Using Omni-Channel Routing | Chat Using Live Agent Routing | Live Message Channel |
|---|---|---|---|
| `RoutableId` | `LiveChatTranscript.Id` | N/A | `MessagingSession.Id` |
| `ContactId` | `LiveChatTranscript.ContactId` | `LiveChatTranscript.ContactId` | `MessagingEndUser.ContactId` |
| `EndUserId` | `LiveChatTranscript.LiveChatVisitorId` | N/A | `MessagingSession.MessagingEndUserId` |
| `ChatKey` | `LiveChatTranscript.ChatKey` | `LiveChatTranscript.ChatKey` | N/A |
| `_LastCustomerInput` | Represents the most recent message that the customer entered. | | |

You can use any of these context variables to access the fields behind them. But what if you want access to information that isn't available out of the box? Like how do you access the `FirstName__c` field that you created? You can tackle that problem in two different ways.

**a.** Write Apex code to grab what you need and make the Apex method accessible to the Bot Builder using the `@InvocableMethod` annotation. For instance, if you're using Omni-Channel, you can use the `RoutableId` context variable to get the `LiveChatTranscript` record, extract the custom field value, and then pass it back to your bot.

**b.** Use the Salesforce Metadata API to add context variables that aren't available out of the box. For instance, you can create a context variable to gain access to a `LiveChatTranscript` custom field.

This recipe includes both methods. Pick the one that best suits your situation.

**7.** (Option 1 of 2: **Apex Code Solution**) Extract the `FirstName__c` field from the `LiveChatTranscript` object using an invocable Apex method.

Tip: For details about the power of Apex for your bot, see Call an Apex Action.

**a.** Create an invocable Apex method for accessing the pre-chat fields.

Let's use Apex code to extract information from the `LiveChatTranscript` record. This code takes the `LiveChatTranscript` record ID as the input, queries for the record using SOQL, extracts the `FirstName__c` field from the record, and returns the first name as an output.

From Setup, use the Quick Find box to find **Apex Classes**. Add a new class.

```
public with sharing class CookbookBot_GetTranscriptVariables {

  public class TranscriptInput {
    @InvocableVariable(required=true)
    public ID routableID;
  }

  public class VisitorNameOutput {
    @InvocableVariable(required=true)
    public String sFirstName;
  }
```

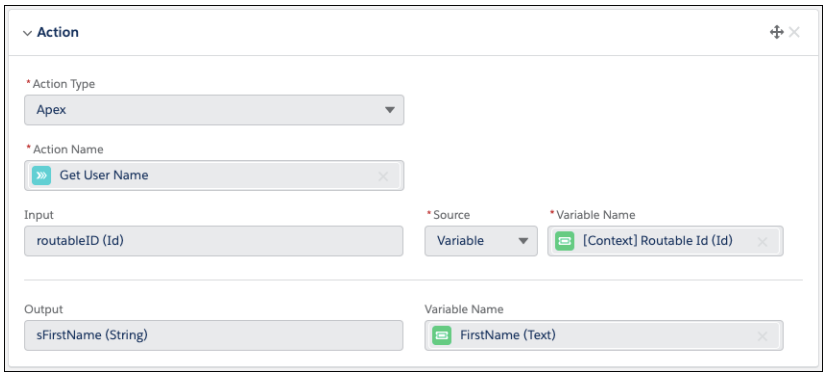
```
 @InvocableMethod(label='Get User Name')
 public static List<VisitorNameOutput> getUserName(List<TranscriptInput> transcripts)
{

   List<VisitorNameOutput> names = new List<VisitorNameOutput>();

   for (TranscriptInput transcript : transcripts) {

     // Query for the transcript record based on the ID
     LiveChatTranscript transcriptRecord = [SELECT Name, FirstName__c
                                            FROM LiveChatTranscript
                                            WHERE Id = :transcript.routableID
                                            LIMIT 1];

     // Store the first name in an output variable
     VisitorNameOutput nameData = new VisitorNameOutput();
     nameData.sFirstName = transcriptRecord.FirstName__c;

     // Add the name to the list of outputs
     names.add(nameData);
   }

   return names;
  }
}
```

**b.** Give the bot permission to access the Apex class.

From Setup, use the Quick Find box to find **Permission Sets**. Select the **sfdc.chatbot.service.permset** permission set. From the permission set options, select **Apex Class Access**. Now you can edit the access settings so that your new class is accessible to the bot. Always make sure that the bot has permission to access objects and classes as required.

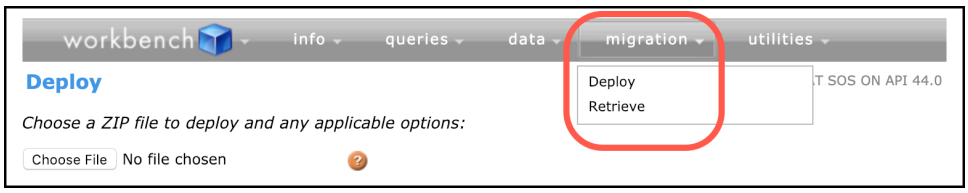
**c.** In Bot Builder, create an Apex action to access the invocable method.

In your bot's "Welcome" dialog, add an action at the top of the flow using the Apex action type. Because you annotated the Apex code with `@InvocableMethod`, Bot Builder can find the Apex method. It prompts you for the required input and output variables. For the input, use the built-in variable `[Context] Routable Id`. When working with Omni-Channel, this variable maps to the ID of the `LiveChatTranscript` record. For the output, use the `FirstName` variable that you created earlier.




**8.** (Option 2 of 2: **Metadata API Solution**) Extract the `FirstName__c` field from the `LiveChatTranscript` object by creating a new context variable with Metadata API.

Note: If you're not familiar with the Metadata API, review the Metadata Developer Guide. To access the API features, you can use a web-based tool called Workbench, the Salesforce Extensions for Visual Studio Code, or the Ant Migration Tool. If you're using Workbench, use the retrieve and deploy migration options.




**a.** Retrieve the metadata for your bot using the Metadata API.

The first step is to retrieve bot metadata using the Metadata API. To retrieve metadata, provide a zip package containing a `package.xml` manifest file. You can use this sample code to get you started.

To retrieve metadata, provide a zip package containing a `package.xml` manifest file. You can use this sample code to get you started. Replace `MY_BOT_API_NAME` with your bot's API name. Also replace the version number with the latest API version of your org.

```
<?xml version="1.0" encoding="UTF-8"?>
<Package xmlns="http://soap.sforce.com/2006/04/metadata">
    <types>
        <members>MY_BOT_API_NAME</members>
        <name>Bot</name>
    </types>
    <types>
        <members>*</members>
        <name>MlDomain</name>
    </types>
    <version>45.0</version>
</Package>
```

**b.** Add the context variable to the retrieved metadata and deploy it back to your org.

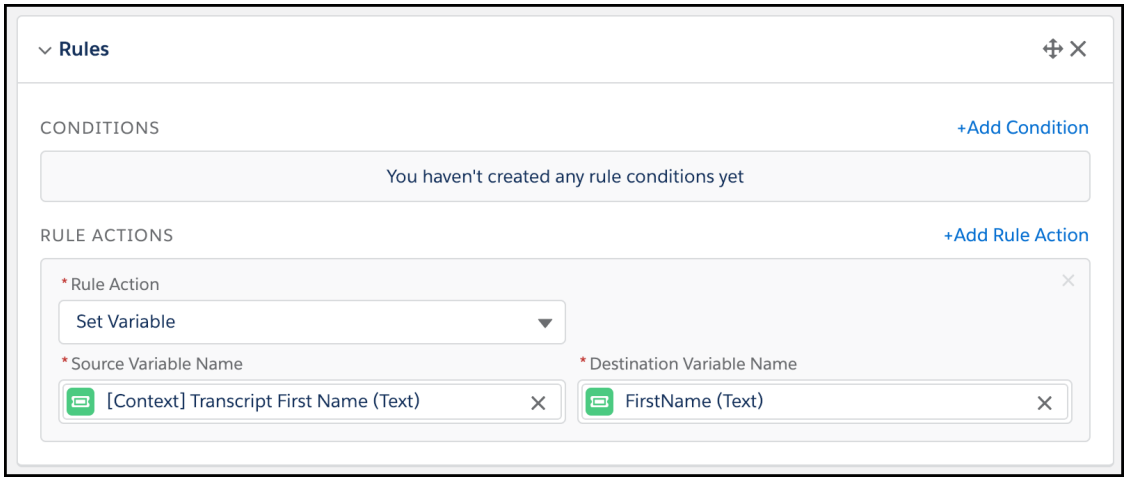
Now you can add a context variable (to the content that you previously retrieved) and then deploy the package back to your org. In the `.bot` file, search for the area that has the context variables, and add the following XML snippet. This snippet adds the context variable using the `FirstName__c` field of the `LiveChatTranscript` object.

```
<contextVariables>
    <contextVariableMappings>
        <SObjectType>LiveChatTranscript</SObjectType>
        <fieldName>LiveChatTranscript.FirstName__c</fieldName>
        <messageType>WebChat</messageType>
    </contextVariableMappings>
    <dataType>Text</dataType>
    <developerName>TranscriptFirstName</developerName>
    <label>Transcript First Name</label>
</contextVariables>
```

After you deploy the package back to your org, you can access the new context variable.

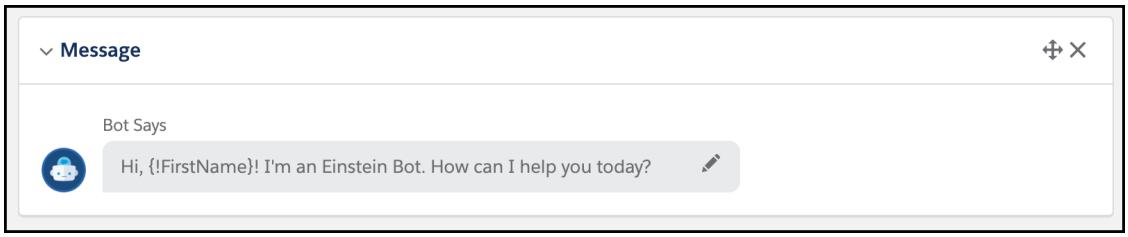
**c.** Create a **Set Variable** rule in Bot Builder for accessing the new context variable.

In your bot's "Welcome" dialog, add an action at the top of the flow using the **Set Variable** action type. For the source, select the new `Transcript First Name` context variable. For the destination, select the `FirstName` variable.



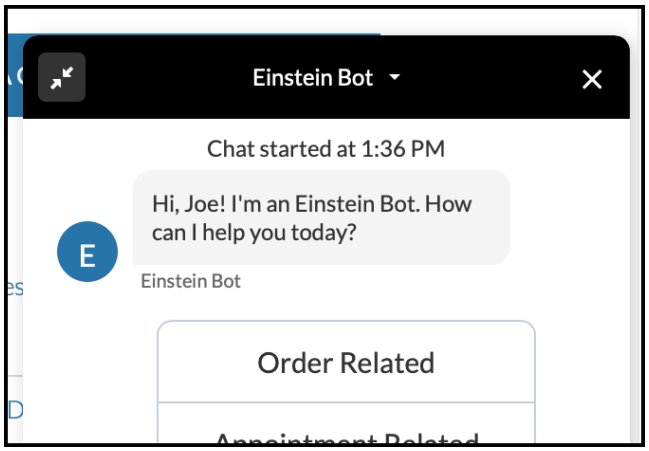

**9.** Merge the `FirstName` variable into the bot's greeting message.

Below the previous action that you just created (either using an **Apex** action or a **Set Variable** rule), update the welcome message so that it includes the `FirstName` variable. The format for merging a variable in a message is `{!Variable_API_Name}`. So to use the `FirstName` variable, specify `{!FirstName}`. The message looks something like this example: `Hi, {!FirstName}! I'm an Einstein Bot. How can I help you today?`




**10.** Test your greeting message.

Using any method described in Set Up Your First Einstein Bot to preview your bot and test out the new functionality.




If the bot is not working as expected, we've found that most errors fall into one of two issues:

- If the bot goes directly to the agent and skips the greeting, the Apex may have failed and should be revisited.
- If the bot is unable to convert the variable and shows {!FirstName}, the Apex was unable to pass info along to the bot or the Snippet Settings file is incorrect.

Note: It's a common request for Embedded Chat to show different avatars when the customer is talking with a bot as opposed to an agent. Although the Lightning component doesn't support different avatars, you can use JavaScript code from `SnippetSettingsFile`. The attribute name is `embedded_svc.snippetSettingsFile.chatbotAvatarImgURL` for a bot avatar and `embedded_svc.snippetSettingsFile.avatarImgURL` for an agent avatar. The same is true if you are configuring from an external site, but add the `embedded_svc.settings` prefix. For more supported attributes in the `snippetSettingsFile`, see the Embedded Service for Web Developer Guide.

## Greet the Customer in Salesforce Classic Chat

If you're using Salesforce Classic Chat, follow these steps to greet the customer.

1. Create custom fields on the `LiveChatTranscript` object to give the bot access to customer information.

   In the Service Cloud Console, agents can view all chat fields on the Chat Details tab. Admins can use the Console Integration Toolkit API to give agents even more information. Because bots don't have a view of an agent's console, the data must be committed to a platform object. Commitment is necessary for giving the bot access to information programmatically. Bots can use the `LiveChatTranscript` object to access pre-chat variables.

   To capture the first name, last name, and email address data from the pre-chat form, create three fields on the `LiveChatTranscript` object.

| | | |
|---|---|---|
| Email | Email__c | Email |
| End Time | EndTime | Date/Time |
| Ended By | EndedBy | Picklist |
| FirstName | FirstName__c | Text(80) |
| Last Modified By | LastModifiedById | Lookup(User) |
| Last Modified Date | LastModifiedDate | Date/Time |
| LastName | LastName__c | Text(80) |

2. Capture the customer name using a pre-chat form.

   Create or verify that the customer's first name is captured in a pre-chat form. For example, this Visualforce page gathers the customer's first name, last name, and email address.

```
<apex:page showHeader="false">

<!-- This script takes the endpoint URL parameter passed from the deployment page and
makes it the action for the form -->
<script type='text/javascript'>
(function() {
function handlePageLoad() {
var endpointMatcher = new RegExp("[\\?\\&]endpoint=([^&#]*)");
document.getElementById('prechatForm').setAttribute('action',
decodeURIComponent(endpointMatcher.exec(document.location.search)[1].replace("javascript:",
 """)));
} if (window.addEventListener) {
window.addEventListener('load', handlePageLoad, false);
} else { window.attachEvent('onload', handlePageLoad, false);
}})();
</script>

<h1>Live Agent Pre-Chat Form</h1>

<!-- Form that gathers information from the chat visitor and sets the values to Live
```

```
Agent Custom Details used later in the example -->
<form method='post' id='prechatForm'>
    First name: <input type='text' name='liveagent.prechat:FirstName' id='firstName'
/><br />
    Last name: <input type='text' name='liveagent.prechat:LastName' id='lastName' /><br
 />
    Email: <input type='text' name='liveagent.prechat:Email' id='email' /><br />

    <!-- Used to set the visitor's name for the agent in the Console -->
    <input type="hidden" name="liveagent.prechat.name" id="prechat_field_name" />

<input type='submit' value='Chat Now' id='prechat_submit' onclick="setName()"/>

<!-- Set the visitor's name for the agent in the Console to first and last name provided
 by the customer -->
<script type="text/javascript">
   function setName() {
    document.getElementById("prechat_field_name").value =
        document.getElementById("firstName").value + " " +
document.getElementById("lastName").value;
    }
</script>

<style type="text/css">
p {font-weight: bolder }
</style>

</form>
</apex:page>
```

3. Add the pre-chat form to the Live Agent button.

   Open the chat button configuration page and set the custom pre-chat form attribute to the Visualforce page created in the previous step. When the customer clicks the chat button, it now opens the custom pre-chat form to gather additional information before the chat session starts.

4. Map the `LiveChatTranscript` field to the pre-chat form.

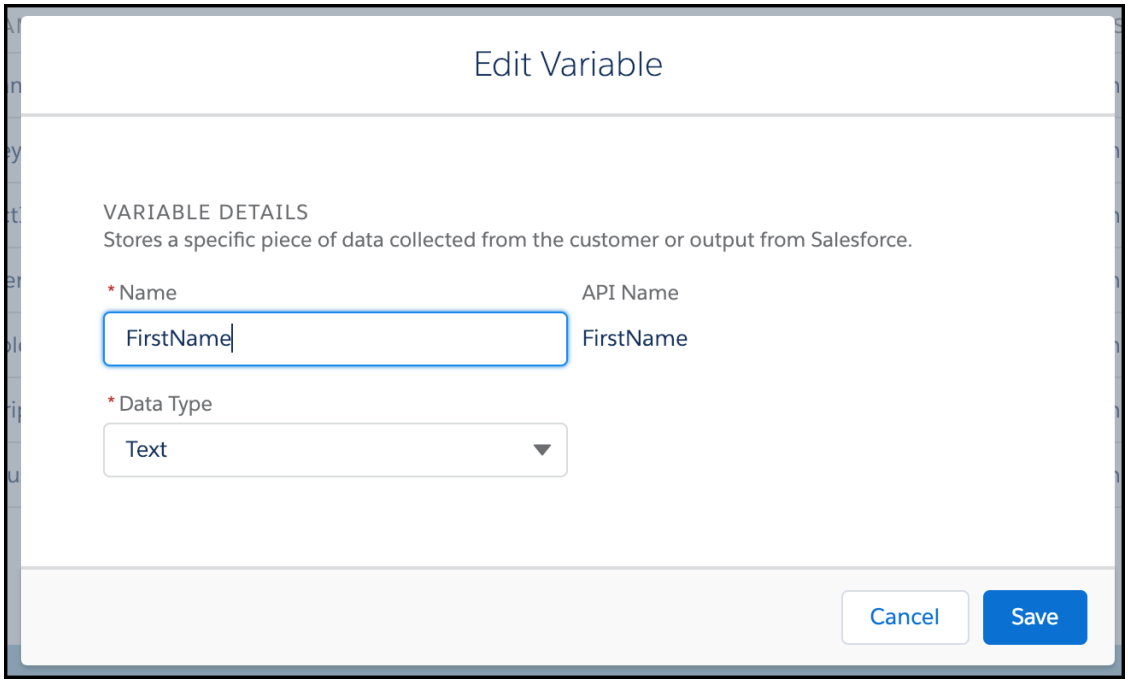
   In the pre-chat form, specify the three custom fields on the `LiveChatTranscript` object using the `liveagent.prechat.save()` pre-chat API. Add the three hidden input fields after the text input fields.

```
First name: <input type='text' name='liveagent.prechat:FirstName' id='firstName' /><br
 />
Last name: <input type='text' name='liveagent.prechat:LastName' id='lastName' /><br />
Email: <input type='text' name='liveagent.prechat:Email' id='email' /><br />

<input type="hidden" name= "liveagent.prechat.save:FirstName" value= "FirstName__c" />
<input type="hidden" name= "liveagent.prechat.save:LastName" value= "LastName__c" />
<input type="hidden" name= "liveagent.prechat.save:Email" value= "Email__c" />
```

   The element name specified after `liveagent.prechat.save:` is the chat detail variable name. Set the value attribute of the input tag to the API name of the `LiveChatTranscript` field to populate.

5. Deactivate the bot and create a variable for first name.

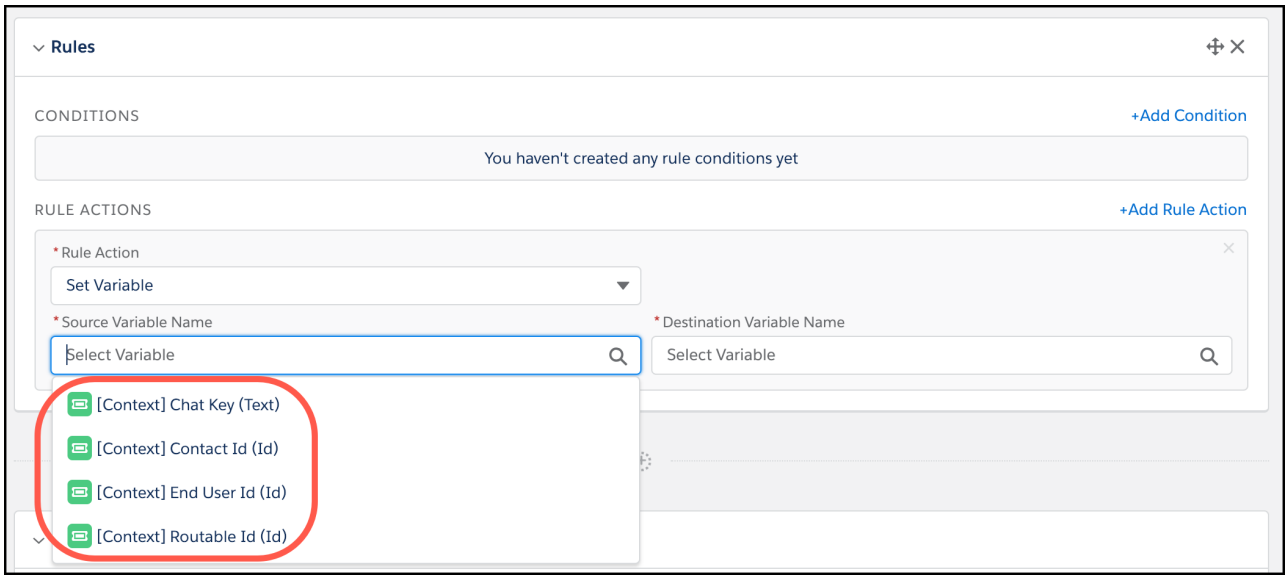
A bot variable stores and passes data between different steps in a bot session. In Bot Builder, on the **Variables** tab, create a variable with the API name `FirstName` and with a data type of **Text**.




We can pass the pre-chat data to a variable, and use the variable to merge the customer name into the greeting message.

6. We now want to extract the value of the `FirstName__c` field from the `LiveChatTranscript` object, and get it into the `FirstName` variable. You can get the `FirstName__c` value in two different ways, but first it's important to understand context variables.

   Let's say you create a **Rule** in a bot flow and specify a **Set Variable** rule action. You can choose from a set of built-in variables to access objects and fields related to the chat session.




The values of these context variables depend on how you set up your bot.

**Table 2: Context Variable Definitions**

| Context Variable | Chat Using Omni-Channel Routing | Chat Using Live Agent Routing | Live Message Channel |
|---|---|---|---|
| `RoutableId` | `LiveChatTranscript.Id` | N/A | `MessagingSession.Id` |
| `ContactId` | `LiveChatTranscript.ContactId` | `LiveChatTranscript.ContactId` | `MessagingEndUser.ContactId` |
| `EndUserId` | `LiveChatTranscript.LiveChatVisitorId` | N/A | `MessagingSession.MessagingEndUserId` |
| `ChatKey` | `LiveChatTranscript.ChatKey` | `LiveChatTranscript.ChatKey` | N/A |
| `_LastCustomerInput` | Represents the most recent message that the customer entered. | | |

You can use any of these context variables to access the fields behind them. But what if you want access to information that isn't available out of the box? Like how do you access the `FirstName__c` field that you created? You can tackle that problem in two different ways.

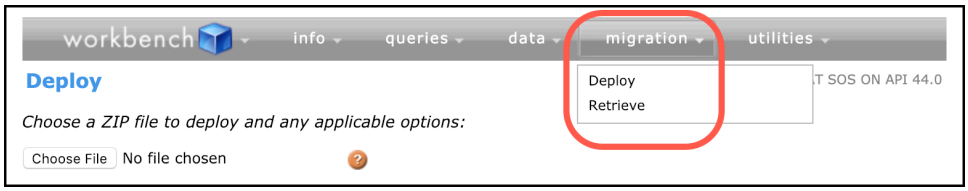
**a.** Write Apex code to grab what you need and make the Apex method accessible to the Bot Builder using the `@InvocableMethod` annotation. For instance, if you're using Omni-Channel, you can use the `RoutableId` context variable to get the `LiveChatTranscript` record, extract the custom field value, and then pass it back to your bot.

**b.** Use the Salesforce Metadata API to add context variables that aren't available out of the box. For instance, you can create a context variable to gain access to a `LiveChatTranscript` custom field.

This recipe includes both methods. Pick the one that best suits your situation.

**7.** (Option 2 of 2: **Metadata API Solution**) Extract the `FirstName__c` field from the `LiveChatTranscript` object by creating a new context variable with Metadata API.

> **Note:** If you're not familiar with the Metadata API, review the Metadata Developer Guide. To access the API features, you can use a web-based tool called Workbench, the Salesforce Extensions for Visual Studio Code, or the Ant Migration Tool. If you're using Workbench, use the retrieve and deploy migration options.




**a.** Retrieve the metadata for your bot using the Metadata API.

The first step is to retrieve bot metadata using the Metadata API. To retrieve metadata, provide a zip package containing a `package.xml` manifest file. You can use this sample code to get you started.

To retrieve metadata, provide a zip package containing a `package.xml` manifest file. You can use this sample code to get you started. Replace `MY_BOT_API_NAME` with your bot's API name. Also replace the version number with the latest API version of your org.

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

```
    <types>
        <members>MY_BOT_API_NAME</members>
        <name>Bot</name>
    </types>
    <types>
        <members>*</members>
        <name>MlDomain</name>
    </types>
    <version>45.0</version>
</Package>
```

**b.** Add the context variable to the retrieved metadata and deploy it back to your org.

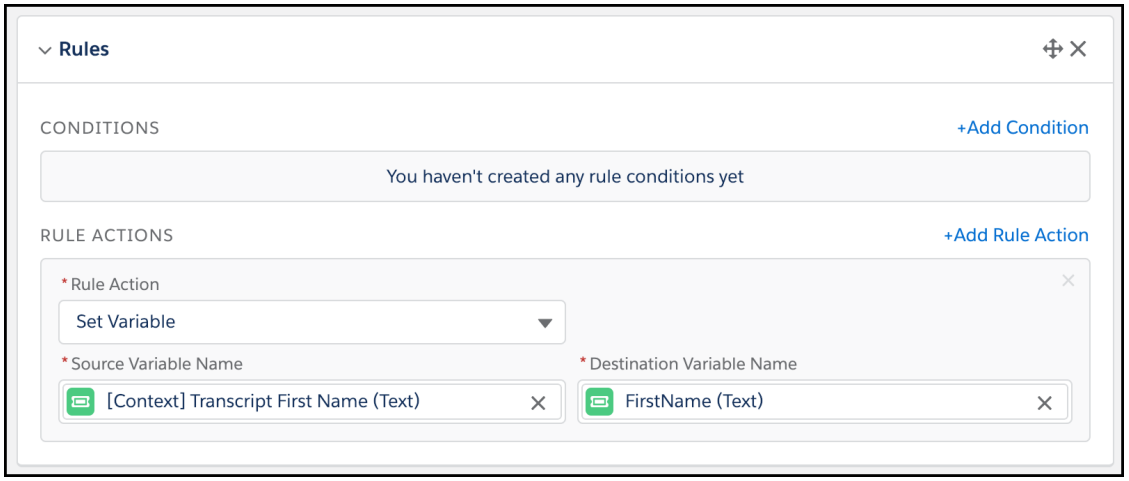
Now you can add a context variable (to the content that you previously retrieved) and then deploy the package back to your org. In the `.bot` file, search for the area that has the context variables, and add the following XML snippet. This snippet adds the context variable using the `FirstName__c` field of the `LiveChatTranscript` object.

```
<contextVariables>
    <contextVariableMappings>
        <SObjectType>LiveChatTranscript</SObjectType>
        <fieldName>LiveChatTranscript.FirstName__c</fieldName>
        <messageType>WebChat</messageType>
    </contextVariableMappings>
    <dataType>Text</dataType>
    <developerName>TranscriptFirstName</developerName>
    <label>Transcript First Name</label>
</contextVariables>
```

After you deploy the package back to your org, you can access the new context variable.

**c.** Create a **Set Variable** rule in Bot Builder for accessing the new context variable.

In your bot's "Welcome" dialog, add an action at the top of the flow using the **Set Variable** action type. For the source, select the new `Transcript First Name` context variable. For the destination, select the `FirstName` variable.




**8.** (Option 1 of 2: **Apex Code Solution**) Extract the `FirstName__c` field from the `LiveChatTranscript` object using an invocable Apex method.

Tip: For details about the power of Apex for your bot, see Call an Apex Action.

**a.** Create an invocable Apex method for accessing the pre-chat fields.

Let's use Apex code to extract information from the `LiveChatTranscript` record. This code takes the `LiveChatTranscript` record ID as the input, queries for the record using SOQL, extracts the `FirstName__c` field from the record, and returns the first name as an output.

From Setup, use the Quick Find box to find **Apex Classes**. Add a new class.

```
public with sharing class CookbookBot_GetTranscriptVariables {

  public class TranscriptInput {
    @InvocableVariable(required=true)
    public ID routableID;
  }

  public class VisitorNameOutput {
    @InvocableVariable(required=true)
    public String sFirstName;
  }

  @InvocableMethod(label='Get User Name')
  public static List<VisitorNameOutput> getUserName(List<TranscriptInput> transcripts)
 {

    List<VisitorNameOutput> names = new List<VisitorNameOutput>();

    for (TranscriptInput transcript : transcripts) {

      // Query for the transcript record based on the ID
      LiveChatTranscript transcriptRecord = [SELECT Name, FirstName__c
                                             FROM LiveChatTranscript
                                             WHERE Id = :transcript.routableID
                                             LIMIT 1];

      // Store the first name in an output variable
      VisitorNameOutput nameData = new VisitorNameOutput();
      nameData.sFirstName = transcriptRecord.FirstName__c;

      // Add the name to the list of outputs
      names.add(nameData);
    }

    return names;
  }
}
```

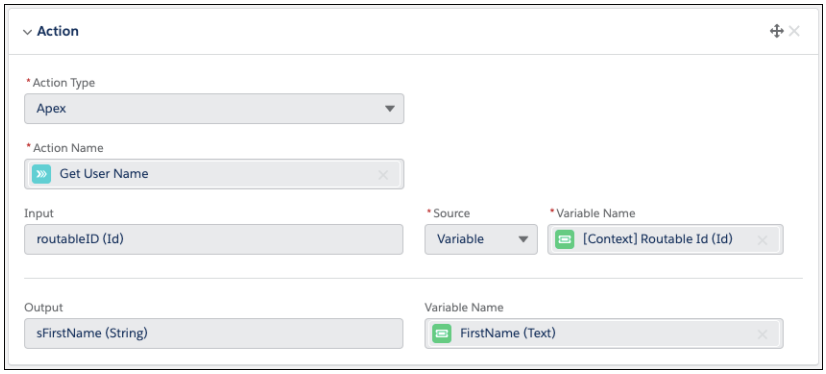
**b.** Give the bot permission to access the Apex class.

From Setup, use the Quick Find box to find **Permission Sets**. Select the **sfdc.chatbot.service.permset** permission set. From the permission set options, select **Apex Class Access**. Now you can edit the access settings so that your new class is accessible to the bot. Always make sure that the bot has permission to access objects and classes as required.

**c.** In Bot Builder, create an Apex action to access the invocable method.

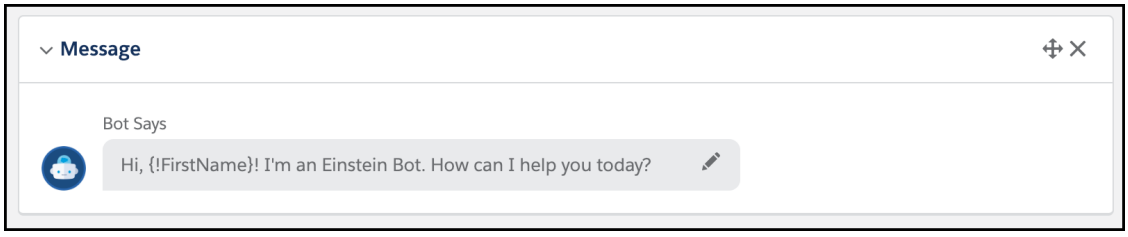
In your bot's "Welcome" dialog, add an action at the top of the flow using the Apex action type. Because you annotated the Apex code with `@InvocableMethod`, Bot Builder can find the Apex method. It prompts you for the required input and output variables. For the input, use the built-in variable `[Context] Routable Id`. When working with Omni-Channel,

this variable maps to the ID of the `LiveChatTranscript` record. For the output, use the `FirstName` variable that you created earlier.



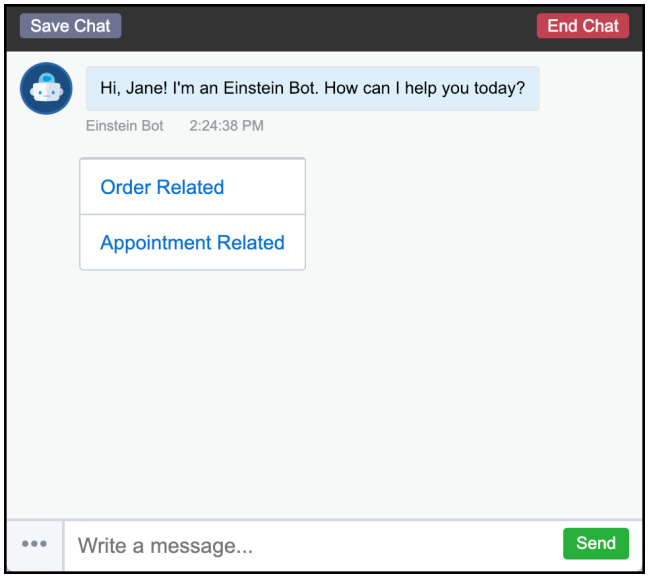

9. Merge the `FirstName` variable into the bot's greeting message.

   Below the previous action that you just created (either using an **Apex** action or a **Set Variable** rule), update the welcome message so that it includes the `FirstName` variable. The format for merging a variable in a message is `{!Variable_API_Name}`. So to use the `FirstName` variable, specify `{!FirstName}`. The message looks something like this example: `Hi, {!FirstName}! I'm an Einstein Bot. How can I help you today?`




10. Test your greeting message.

    Using any method described in Set Up Your First Einstein Botto preview your bot and test out the new functionality.



Warning: Adding a custom pre-chat form to a Live Agent button configuration page breaks any Snap-ins deployments that are built on top of this chat button. Snap-ins chat deployments have a pre-chat form built in, so it doesn't expect the underlying chat button to have another Classic pre-chat page. Test with Snap-ins or use Preview in the Bot Builder to create another Live Agent button with no pre-chat form.

## Greet the Customer with Messaging

If you're using bots with Messaging, follow these steps to greet the customer.

**Prerequisites**

- Set Up Your First Einstein Bot on page 1

**In this recipe you learn how to:**

- Access context variables with the Einstein Bot Builder.
- Use Apex code to search for a contact based on a phone number.
- Greet the customer by name.
- Learn about intent detection and why it's particularly valuable with bots and messaging.

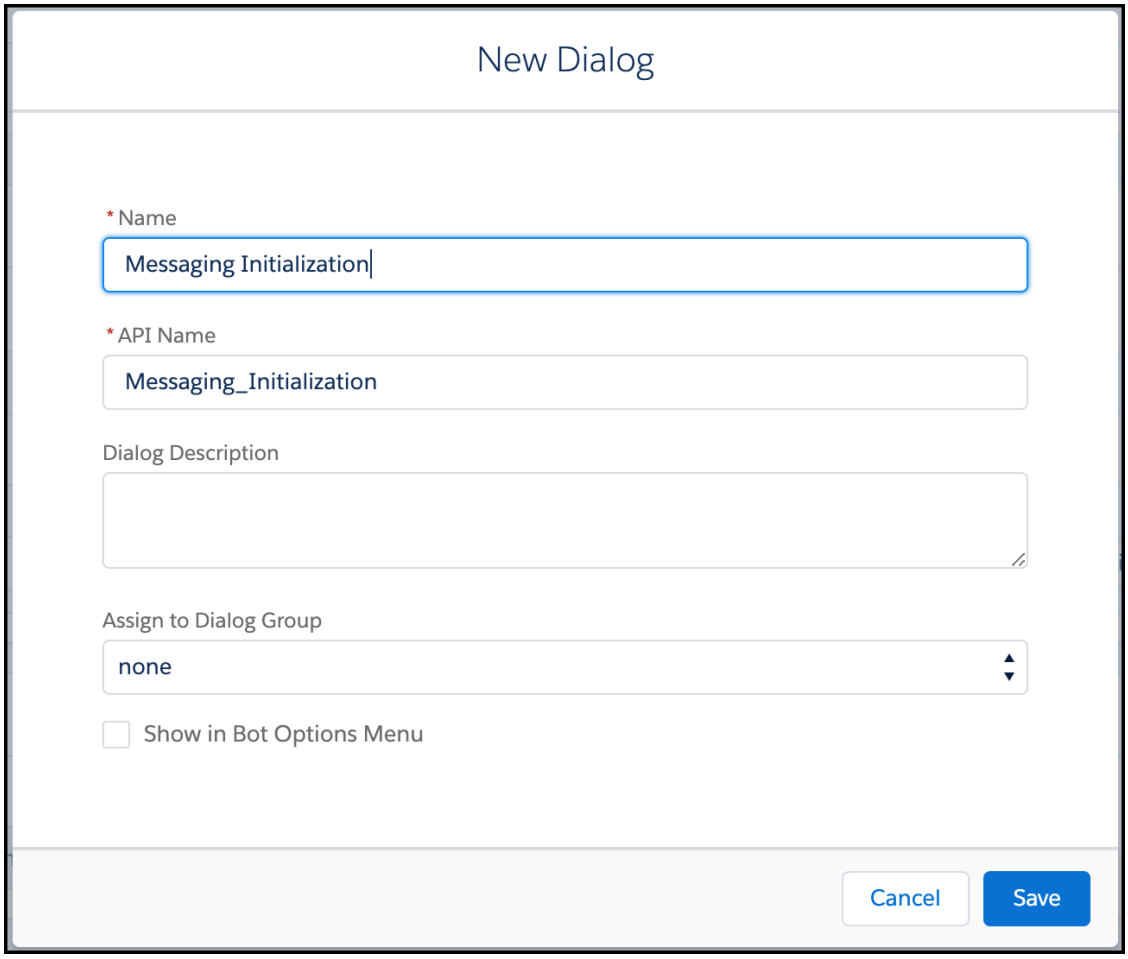
The beginning of a messaging conversation is a little different than a chat session. For one, there isn't any notion of a pre-chat form—you can't query customers in advance for information. However, you can still query your org using the information that you do have. For some messaging channels (such as SMS), you have their phone number. For other channels (such as Facebook Messenger), you have a username or an ID. With this information, you can use Apex or a flow to get contact information from your org, if it exists.

Also, unlike a chat session, the customer can initiate the conversation by asking a question. This flow is different than a chat session, where your bot can start the conversation. However, you can use intent detection and other techniques to direct a customer to the appropriate dialog within your bot flow.

In what follows, we outline the process to handle these issues. First, we discuss how to find information about a user without the benefit of a pre-chat form.

1. Create a dialog for messaging initialization.

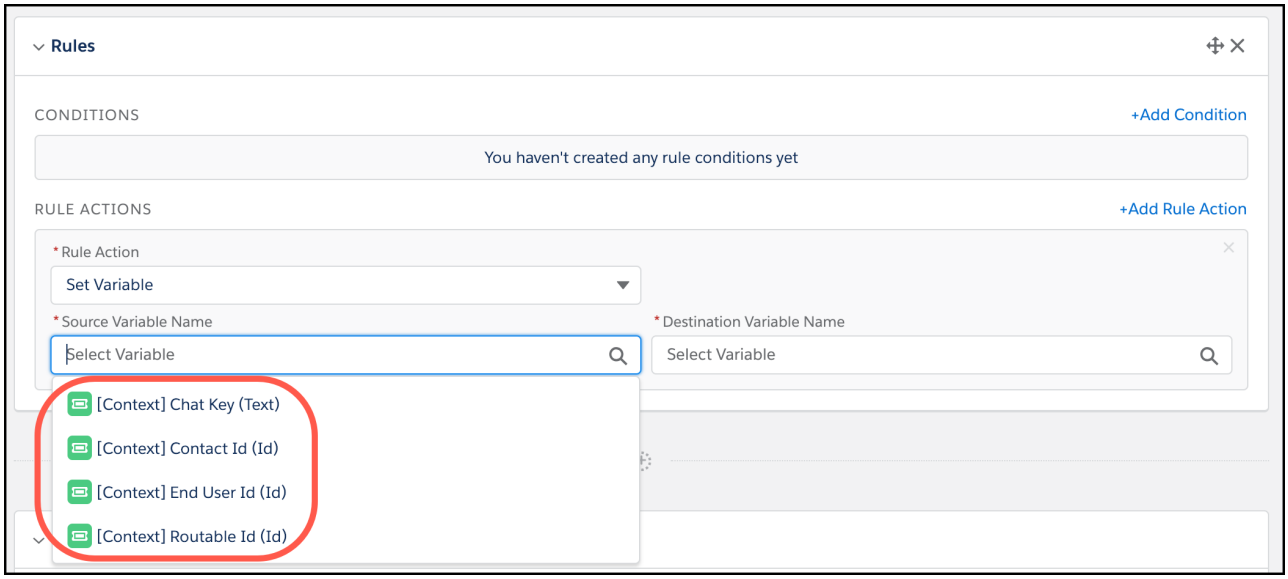
   We could add the initialization process directly to the "Welcome" dialog, but as you'll soon see, a messaging conversation could start in another dialog, especially once you learn about intent detection on page 71. For this reason, let's create a separate initialization dialog that we can call from the "Welcome" dialog and from other spots in our bot design. Go to the **Dialogs** tab and confirm that your bot is deactivated to edit your bot. Click the **+** button and select **New Dialog**. Let's create a dialog called "Messaging Initialization".




2. Review the available context variables.

   Although we have a limited amount of information to work with, we can still use the `MessagingEndUser` record to get some information about the user. But before we grab information from this record, let's discuss the information available to us from within a bot dialog.

   When creating a **Rule** or an **Action** in a bot flow, you can choose from a set of built-in variables to access objects and fields related to the session.




The values of these context variables depend on how you set up your bot.

**Table 3: Context Variable Definitions**

| Context Variable | Chat Using Omni-Channel Routing | Chat Using Live Agent Routing | Live Message Channel |
|---|---|---|---|
| `RoutableId` | `LiveChatTranscript.Id` | N/A | `MessagingSession.Id` |
| `ContactId` | `LiveChatTranscript.ContactId` | `LiveChatTranscript.ContactId` | `MessagingEndUser.ContactId` |
| `EndUserId` | `LiveChatTranscript.LiveChatVisitorId` | N/A | `MessagingSession.MessagingEndUserId` |
| `ChatKey` | `LiveChatTranscript.ChatKey` | `LiveChatTranscript.ChatKey` | N/A |
| `_LastCustomerInput` | Represents the most recent message that the customer entered. | | |

Note: With Messaging, the `ContactId` context variable, which is tied to the `MessagingEndUser.ContactId` field, is only filled if an agent has previously linked the Messaging User to the Contact record. This link can be done when an agent receives a link notification or through Channel-Object Linking. For this reason, the `ContactId` variable is `null` the first time a Messaging user reaches out, or for the returning Messaging user who has only encountered a bot. The Apex code in this recipe therefore uses the `MessagingEndUser` record in the `EndUserId` variable to more reliably access contact information.

You can use any of these context variables to access the fields behind them. In the next few steps, we use the `EndUserId`, which contains the ID of the `MessagingEndUser`. This object contains information depending on the particular messaging channel. For a text message, the `Name` field contains the user's phone number. For other channels, it can be the username.

3. Write Apex code to grab info from the `MessagingEndUser` record and then search for a contact based on that info.

Tip: For details about the power of Apex for your bot, see Call an Apex Action. If you'd rather perform this action with Flow Builder, see Call a Flow Action on page 43.

Create an invocable Apex method to access the messaging end user. If the messaging user is a text messaging user, you can grab the phone number from the `Name` field and use that number to look up a `Contact` record with the same number.

From Setup, use the Quick Find box to find **Apex Classes**. Add a new class.

```
public with sharing class CookbookBot_MessagingContact {

  public class MessagingInput {
    @InvocableVariable(required=false)
    public ID endUserID;
  }

  public class MessagingOutput {
    @InvocableVariable(required=false)
    public String sContactName;
  }

  private static String scrubPhoneNumber(String rawNumber) {
    if (rawNumber == null) {
      return null;
    }

    // Remove all non-digit values
    String scrubbedNumber = rawNumber.replaceAll('[^0-9]', '');

    // NOTE: You can add some additional logic here to handle
    //       international numbers and various edge cases.

    return scrubbedNumber;
  }

  @InvocableMethod(label='Find Contact for Messaging User')
  public static List<MessagingOutput> getUserName(List<MessagingInput> inputs) {

    List<MessagingOutput> outputs = new List<MessagingOutput>();

    for (MessagingInput input : inputs) {

      // Create a variable to store the user name
      String name = 'messaging user';

      // Get the messaging end user record from the ID
      List<MessagingEndUser> messagingUsers = [SELECT Id, Name
                                               FROM MessagingEndUser
                                               WHERE Id = :input.endUserID
                                               LIMIT 1];
      // Did we find the messaging user?
      if (!messagingUsers.isEmpty()) {

        // Strip out any invalid values from the phone number
        String phoneNumber = scrubPhoneNumber(messagingUsers[0].Name);
```

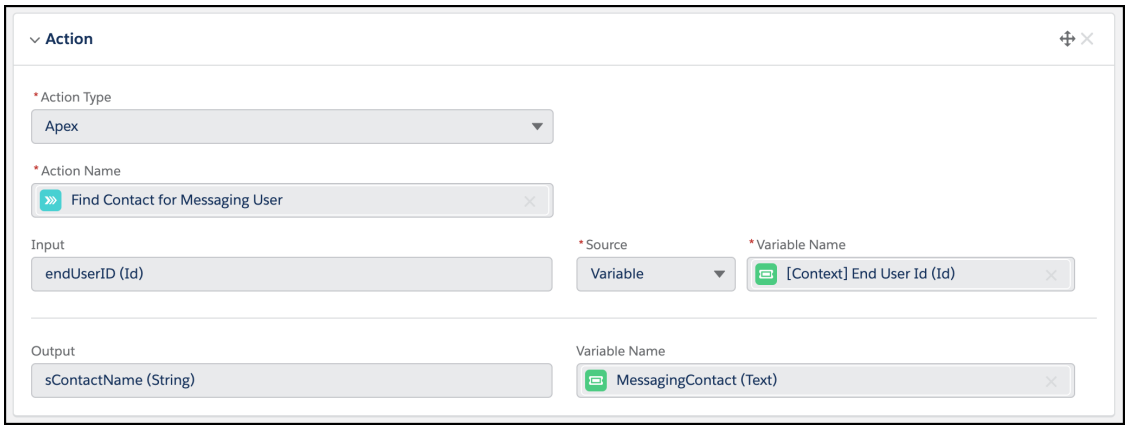
```
        // Get the contact based on the phone number
        // (NOTE: This is just a simple example query.
        //  Adjust this query as needed...)
        List<Contact> contacts = [SELECT Name
                                  FROM Contact
                                  WHERE Phone = :phoneNumber
                                  LIMIT 1];

        // Did we find the contact?
        if (!contacts.isEmpty()) {

          // Grab the name of the contact
          name = contacts[0].Name;
        }
      }

      // Store the name in an output variable
      MessagingOutput output = new MessagingOutput();
      output.sContactName = name;

      // Add the name to the list of outputs
      outputs.add(output);
    }

    return outputs;
  }
}
```

**Important:** As mentioned in Set Up Your First Einstein Bot, you must give the bot permission to access an Apex class. From Setup, use the Quick Find box to find **Permission Sets**. Select the **sfdc.chatbot.service.permset** permission set. From the permission set options, select **Apex Class Access**. Now you can edit the access settings so that your new class is accessible to the bot. Always make sure that the bot has permission to access objects and classes as required.

4. Create an Apex action in Bot Builder to access the invocable method.

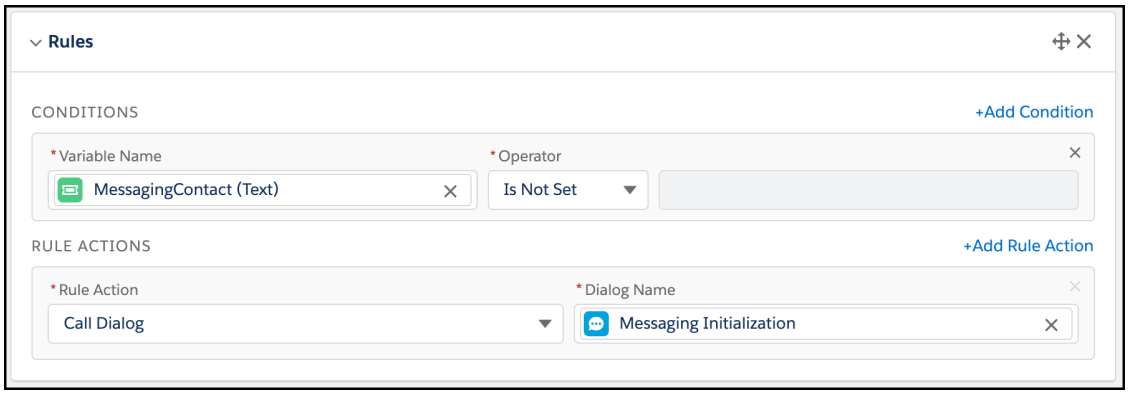
   In your bot's "Messaging Initialization" dialog, add an action at the top of the flow using the Apex action type. Because you annotated the Apex code with `@InvocableMethod`, Bot Builder is smart enough to find the Apex method, and it prompts you for the required input and output variables.

   For the input, use the built-in variable `[Context] End User Id`. This variable grabs the `MessagingEndUser` ID. For the output, create a new variable named `MessagingContact`.



5. Return to the "Welcome" dialog and create a **Call Dialog** rule.

   Add a rule to the top of the "Welcome" dialog flow. Since we want to reuse this logic in a few dialogs and since we don't want to call our code more than one time per session, let's add a condition. Click **Add Condition** to the rule. Now configure the condition so that **MessagingContact** is the variable name and that the operator is **Is Not Set**. Set the **Rule Action** to `Call Dialog` and the **Dialog Name** to `Messaging Initialization`. Now this rule calls the initialization dialog only if the variable isn't already set. Once the dialog completes, control returns to this dialog.

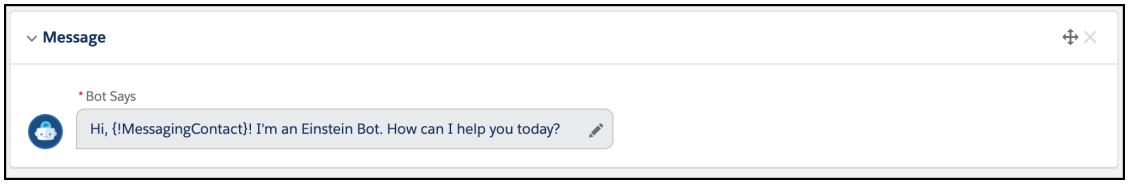



   After you add intent detection on page 71 to other dialogs, add this same rule to the beginning of those dialogs as well.

6. Merge the `MessagingContact` variable into the bot's greeting message.

   Below the action that you just created, update the welcome message so that it includes the `MessagingContact` variable. The format for merging a variable in a message is {!Variable_API_Name}. So to use the `{!Variable_API_Name}` variable, specify `{!MessagingContact}`.

   The message looks something like this: `Hi, {!MessagingContact}! I'm an Einstein Bot. How can I help you today?`



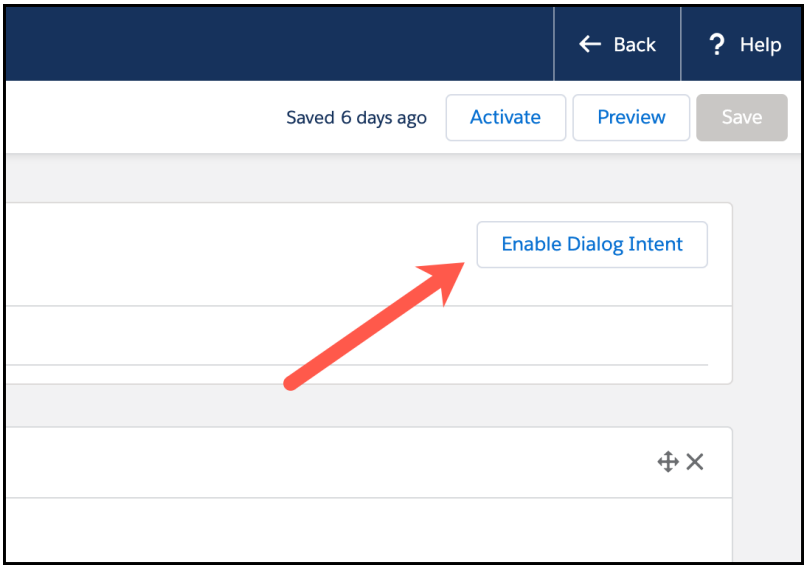

7. Test your greeting message.

   If the phone number you're testing against has a contact record in your org, you can expect to see the name of the contact in your test message.


This dialog is still a bit simplistic for an ideal greeting. For instance, if the Apex code doesn't find a contact, it returns "messaging user" as the name. Not the most elegant way to speak with a customer. As a follow-up exercise, you can change the flow so that the Apex

method returns an empty string if it can't find a contact. Then, the dialog can check whether it got a contact back. If so, it can use the contact's name, as shown above. If not, it can display a different, more generic message.

To address scenarios where the user initiates a conversation before the bot gets in the action, you can use intent detection. With intent detection, any of our dialogs can automatically detect a relevant input for a particular flow and then take control of the conversation. This behavior involves turning on intent detection for a dialog, and then adding enough utterances for the natural language processing to figure out when it's time to kick in.




Once our other dialogs are more fleshed out, you'll see the power of getting the customer to the right part of your bot flow as fast as possible. To learn more about intent detection, see the Natural Language Processing on page 71 section.

# Prompt Customer with Menu Options

In this recipe, we add more options to the main menu. We use Dialog Groups to create internal structure to the dialogs, and the Next Step tool to create menus and submenus for the bot to deliver to the customer.

**Channels**

- Chat
- Messaging

**Prerequisites**

- Set Up Your First Einstein Bot on page 1

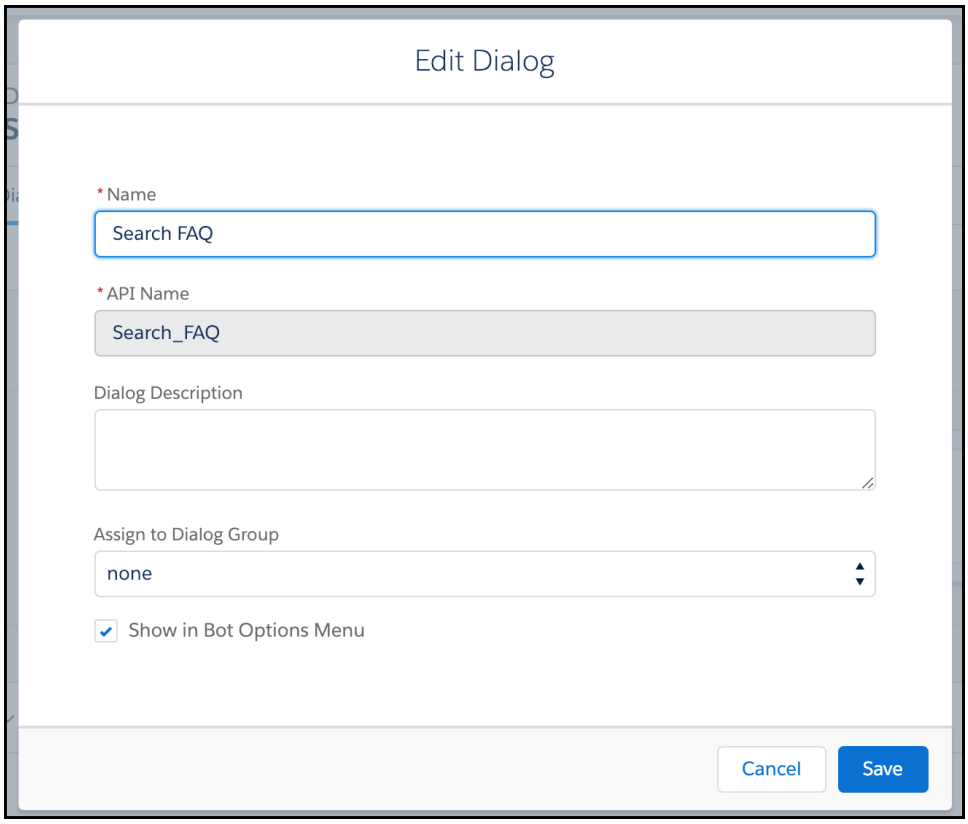
**In this recipe you learn how to:**

- Use Dialog Groups to create an internal structure to bot dialogs.
- Use the Next Step Tool to add a submenu.

When our cookbook bot was first created, we entered **Order Related** and **Appointment Related** in the setup wizard as our initial main menu options. The result was a couple of empty dialog placeholders that are listed from the main menu dialog. Let's add more options to the main menu.

1. Add a few more dialogs.

Since every menu option points to a dialog, a dialog must be built for each of the menu options. Go to the **Dialogs** tab and confirm that your bot is deactivated to edit your bot. Click the **+** button and select **New Dialog**. Let's create a dialog called "Search FAQ".



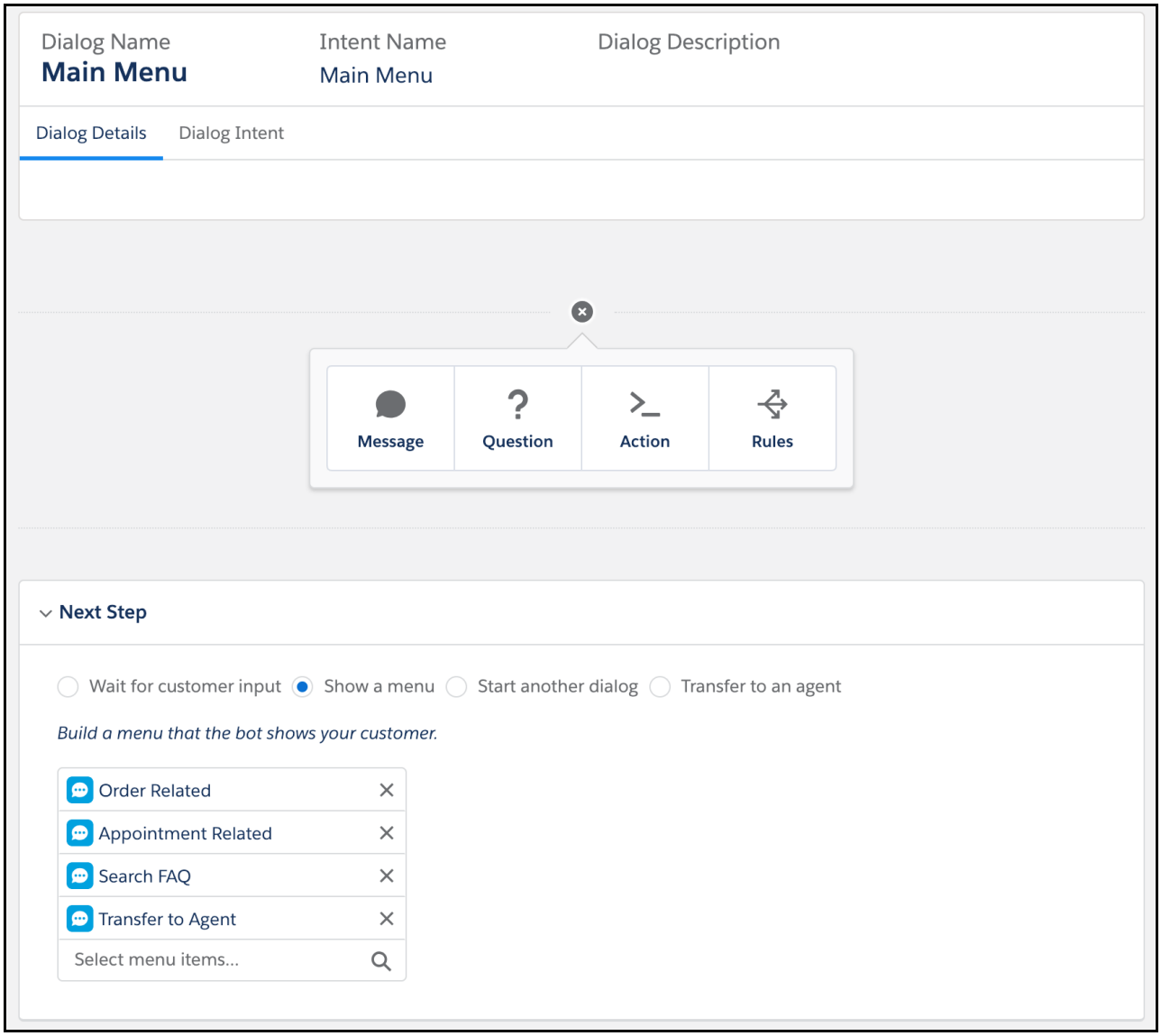

In creating a dialog, there are two options that are optional but allow for better organization: Assign to Dialog Group and Show in Bot Options Menu. The setting **Assign to Dialog Group** provides admins an easy way to group similar dialogs together in the Bot Builder for easy navigation.

The setting **Show in Bot Options menu** gives customers a quick way to kick off this dialog from the shortcut menu at any time during the chat conversation. To access this menu, look for the **...** icon to the left of the chat input text box on the chat window.

2. Add the dialogs to the main menu.

   Now open the "Main Menu" dialog. At the bottom of the screen, there is the **Next Step** configuration area. This area is where we say what the next step is after this dialog. You can set it to **Show a menu**, **Start another dialog**, **Transfer to an agent**, or simply **Wait for customer input**, which implies you will use Natural Language Processing (NLP) on page 71 to identify the customer intent and kick off the next dialog.

   For the "Main Menu" dialog, the choice is obviously to **Show a menu**. To add the newly created "Search FAQ" dialog option to the menu, click the magnifying glass . There are also a few pre-built dialogs such as **Transfer to Agent** or **End Chat**. Let's add the **Transfer to Agent** option as well.

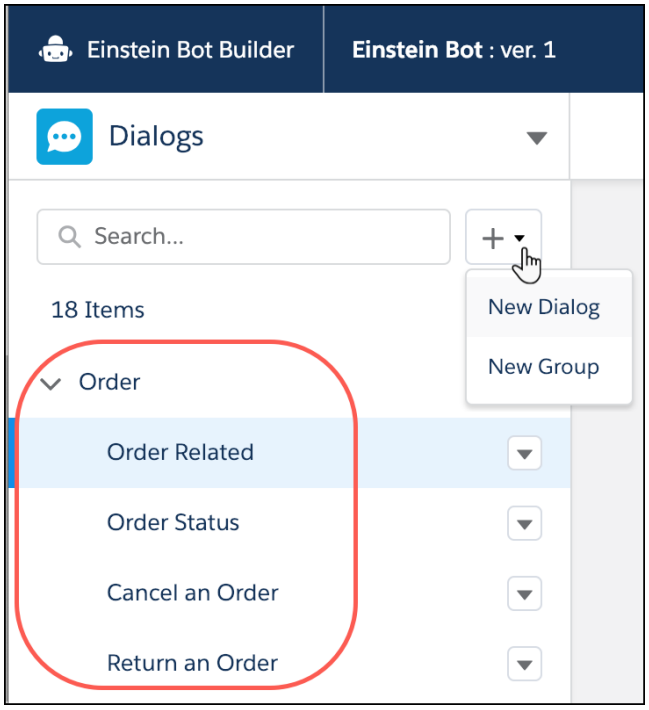



3. Add a submenu.

   While you can add more options to the main menu to support different inquiry types, it's probably not a good idea to list them all in one place when you have many menu options. It's likely going to overwhelm customers. The list could get longer than the height of their phone so that they have to keep scrolling to see the menu options. It's important to take into account your channel as you're creating menus and submenus. In Chat, when a bot has more than 5 menu options, it's recommended to group them in different categories and use submenus to add more options. In Messaging, the limit is 3 menu options.

   Using the Next Step tool on the "Order Related" dialog, let's create a submenu. In this submenu, we support "Order Status", "Cancel an Order" and "Return an Order" transactions. Similar to main menu options, each submenu option also maps to a dialog. Let's go ahead and create these three dialogs. You don't have to add anything yet. We'll create them as placeholders for now. Since these dialogs are all related to customer orders, we can also create a dialog group named "Order" in Bot Builder to group them together.

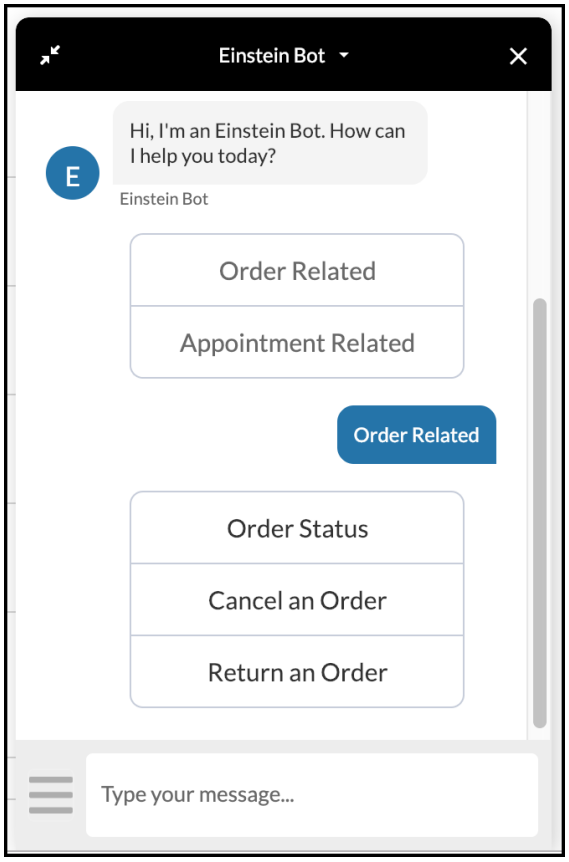
   To create a group, click the **+** button and select **New Group**. Then assign your new dialogs to this new group.




4. Test!

   OK. Let's test the new flow as a customer. Don't forget to **Activate** the bot first. Preview your bot in any of the ways described in Set Up Your First Einstein Bot.

   Voila, here is our new menu navigation.




While it's not a best practice to have a long menu, it's also not a best practice to have so many levels of menus that the customer will request a transfer to an agent. The more inquiry types the bot supports, the more we should leverage Natural Language Processing (NLP) to help identify customer intent and jump to the topic right away.

# Beginner Bot Recipes

Use these recipes to learn how to perform some basic integrations to your bot.

Before working through these recipes, go through the recipes in Get Started with Bots.

Gather Customer Information Using the Bot
In this recipe, we ask the user for information and store that information using entities and variables.

Call an Apex Action
In this recipe, we use an invocable Apex method to get the status of an order.

Call a Flow Action
In this recipe, we continue building out our order support functions with a flow that allows a customer to cancel an order.

Optimize Bot Flow with Pre-Chat Data
In this recipe, we gather information from a pre-chat form and use this data to improve the user experience.

Understand Customer Intent with Natural Language Processing
In this recipe, we learn how to set up Natural Language Processing (NLP) so we can make the bot smarter, and more conversational.

Create a Self-Service Bot with Knowledge, Flows, and Cases

In this recipe, we use multiple Einstein Bot features to solve a common use case: what happens when your agents aren't available but your customers have a question? In this business case, we want the bot to ask the customer about the issue, search the Knowledge base to deliver a self-serve solution, and then if the customer needs further assistance, create a case in Salesforce for your agents.

# Gather Customer Information Using the Bot

In this recipe, we ask the user for information and store that information using entities and variables.

**Channels**

- Chat
- Messaging

**Prerequisites**

- Set Up Your First Einstein Bot on page 1

**In this recipe you learn how to:**

- Create entities and variables to store formatted data.

So far we have created a few dialogs. But they are mostly empty. Let's build conversations in them. There are four types of elements you can add to a dialog: **Message**, **Question**, **Action**, and **Rules**. We have already seen how a **Message** works from the welcome dialog. It displays a message to your customer and you can merge variables in a message with this format: `{!VariableAPIName}`.

In this recipe, we'll talk about the **Question** element, where you configure the bot to ask a question and store the answer in a variable. When we are building out our cookbook bot, guess what we need to know first to do any of the order-related inquiries? That's right, we need to ask for the order number and store it somewhere.

1. Create a custom object to store customer order information.

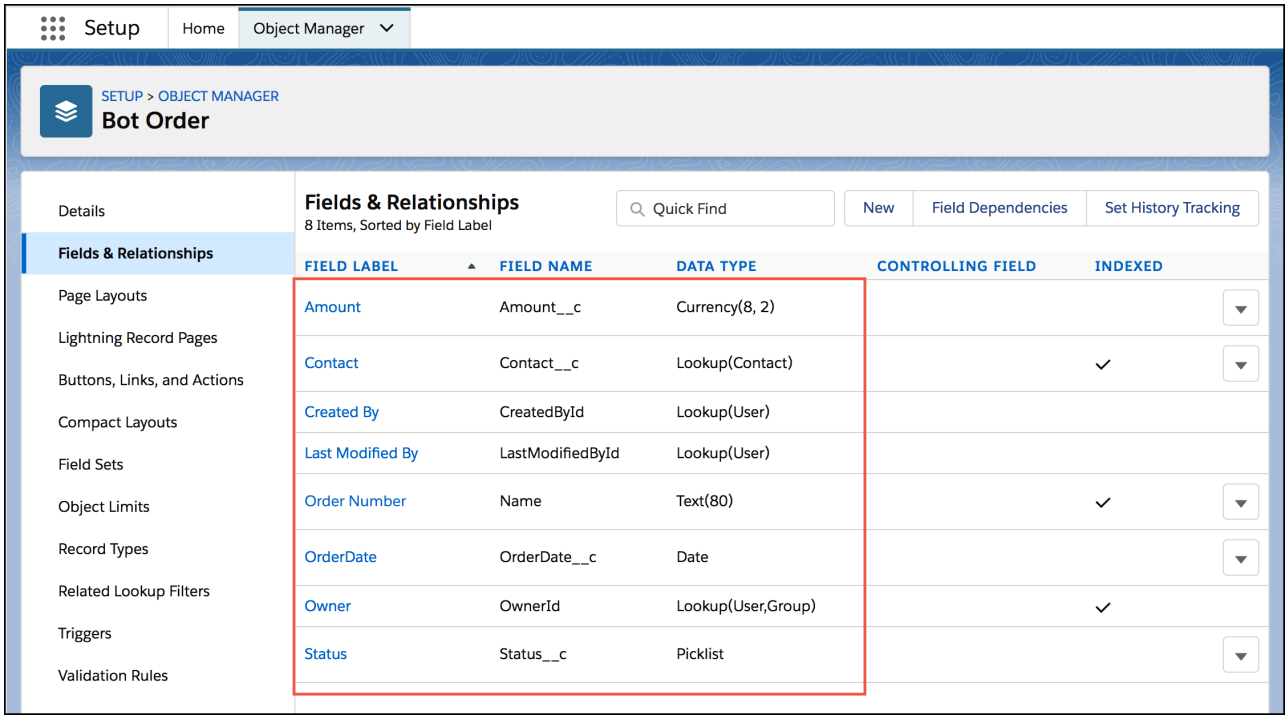
   Many use cases in this cookbook revolve around customer orders, so let's create a custom object to store order data. We won't use this object yet, but it'll be handy for future recipes.

   Let's create a custom object with the API name of `Bot_Order__c` and with these fields:

   - `Name` (Text)
   - `Status__c` (Picklist): `New`, `Packaging`, `Shipped`, `Canceled`, `Returning`, `Returned`
   - `OrderDate__c` (Date)
   - `Contact__c` (lookup to `Contact`)

   The actual order number is stored in the `Name` field.

   When you build this `Bot_Order` object in your org, it should look like this screenshot:




**Important:** As mentioned in Set Up Your First Einstein Bot, you need to give the bot permission to access this custom object and these fields. From Setup, use the Quick Find box to find **Permission Sets**. Select the **sfdc.chatbot.service.permset** permission set. From the permission set options, select **Object Settings**. Select your custom object and then make sure that this permission set has read and edit permissions to the object and all the relevant fields.

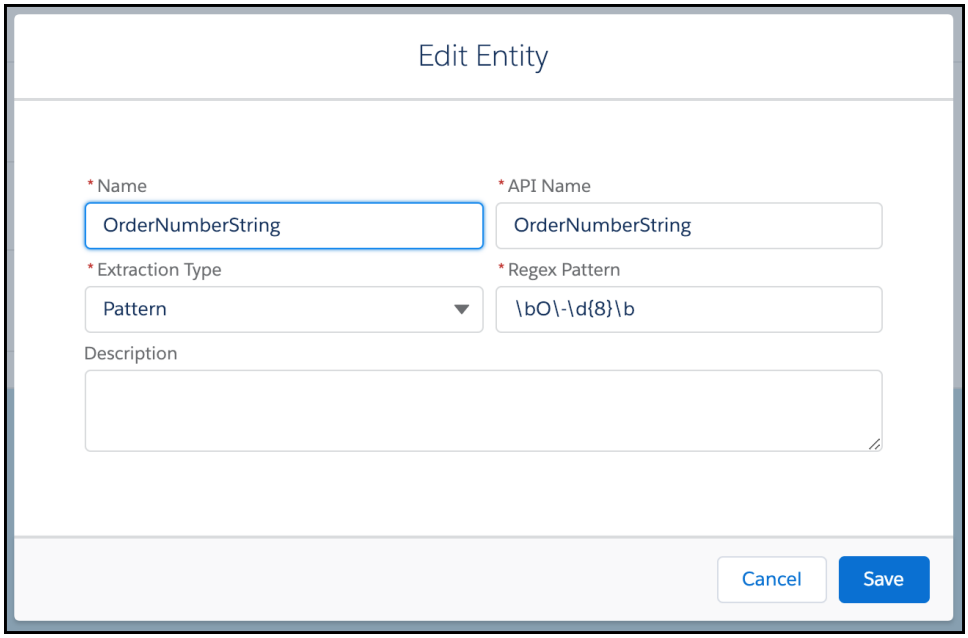
While we're at it, let's create a few test records for this object. Make sure that you remember the order number for the record you create so that you can use it later in this cookbook. Create at least one record with a name of `O-00123456` for later use.

2. Create an entity.

   We touched on this topic briefly in Greet the Customer when we created the first variable for customer `FirstName` and we used the system entity **Text**. We can also create custom entities.

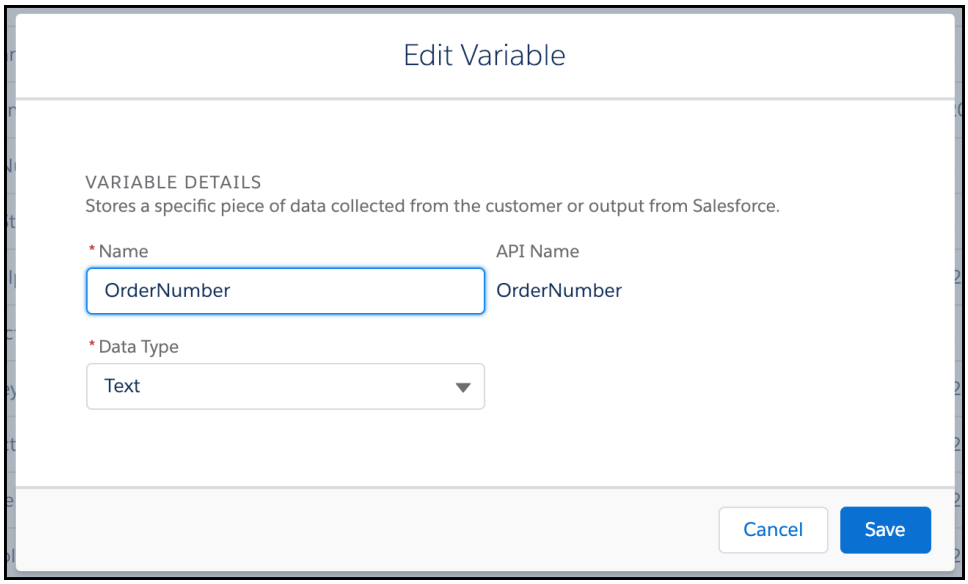
   In any order management system, order numbers typically follow a certain format. Let's say in our system, it always starts with letter O, then a dash, then 8 numbers. Something like: `O-00123456`. Knowing this format helps the bot recognize when customers enter an order number in a conversation.

   Let's create a new entity from the **Entities** tab. Name the entity `OrderNumberString` with an extraction type **Pattern** and specify a **RegEx Pattern** of `\bO\-\d{8}\b` . For those of you who are not familiar with regular expressions, this expression indicates that an order number must be one word, starting with a letter O, followed by a dash, then ending with 8 numbers.




3. Create a variable.

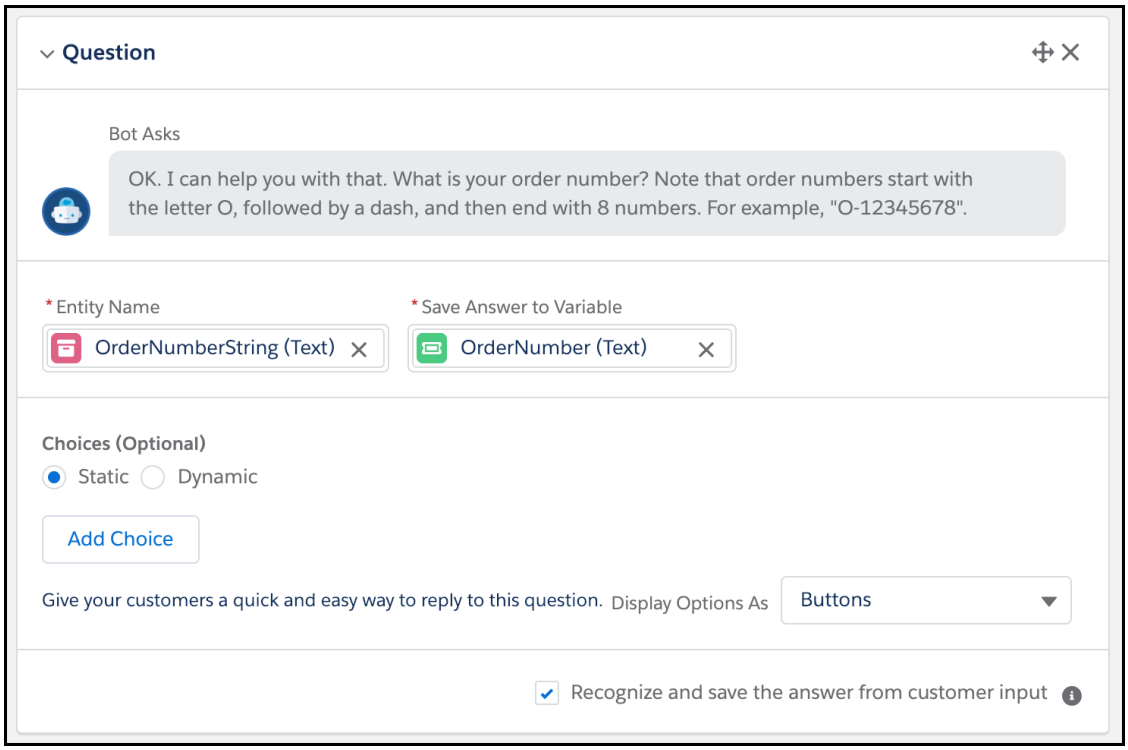
   Now we can create a variable from the **Variables** tab. Name the variable `OrderNumber`. Let's select **Text** as its data type. We'll apply the entity `OrderNumberString` to it when we ask the question, thus we need to make sure the base data types match between the entity and variable.




4. Add a question to the "Order Status" dialog.

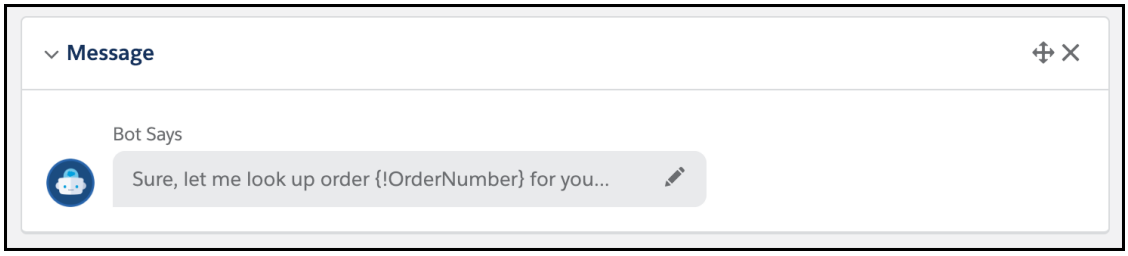
   Open the "Order Status" dialog, click the **+** button in the main body of the dialog and select **Question**. In the **Bot Asks** input box, add a question: `OK. I can help you with that. What is your order number? Note that order numbers start with the letter O, followed by a dash, and then end with 8 numbers. For example, "O-12345678".`

Set the **Entity Name** to `OrderNumberString` entity and the **Save Answer to Variable** attribute to the `OrderNumber` variable. This means that we'll be looking for input from customers for the order number. We'll apply the `OrderNumberString` entity to identify the input and the result will be saved into the `OrderNumber` variable we just created.



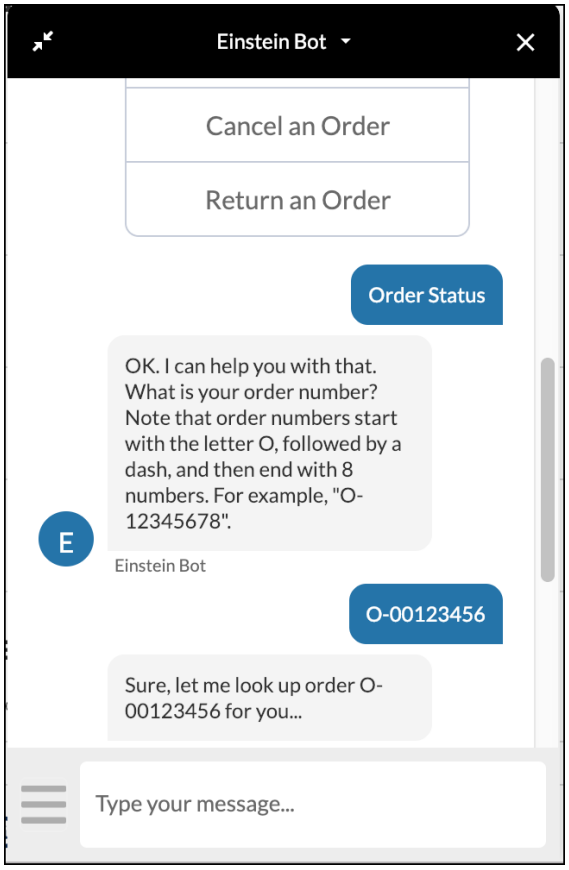

5. Add a response message to the "Order Status" dialog.

   To help verify we indeed captured the customer entry, add a new **Message** with the text: `Sure, let me look up order {!OrderNumber} for you...` This way, the variable gets merged into a message so that we can verify the result.




6. Test!

   Now, when a customer asks about order status, we ask for the order number so we can look it up for the customer later.




Try using an incorrect format for the order number and see what happens.

In Call an Apex Action and Call a Flow Action, we'll actually look up these orders in our custom object.

# Call an Apex Action

In this recipe, we use an invocable Apex method to get the status of an order.

**Channels**

- Chat
- Messaging

**Prerequisites**

- Set Up Your First Einstein Bot on page 1
- Greet the Customer on page 6
- Prompt Customer with Menu Options on page 29

**In this recipe you learn how to:**

- Create an invocable action with Apex.
- Run Apex code from Bot Builder.
- Pass variables back and forth between Bot Builder and Apex code.
- Get information from the user and perform an action based on that information.

- Return a result to the user after performing an action.

Tip: Apex Action or Flow Action? We provide one recipe to introduce you to flow actions and another recipe to introduce you to Apex actions. You'll quickly see that you can do many of the same things with both solutions. Which one should you use? Think about the flow option first for simple solutions. It's declarative and is easier to maintain. Apex, on the other hand, offers much power and many more use cases not available with a simple flow. Another option is to use both. You could add Apex actions to a flow. From within **Flow Builder**, just drag an **Apex Action** element onto the canvas. Or, you can invoke a flow from Apex. After some experience with both flows and Apex, you'll be able to determine how to most effectively use these tools together to get the job done.

How does the bot get the status of a particular order? The **Action** element in dialogs supports different types of actions—such as an Apex action or a Flow action—to trigger a transaction or process. Among all of the action types, Apex has the broadest use cases, so we talk about Apex first. Since every company's order management system is different, you can store the order data in Salesforce in a custom object, or externally somewhere in a data warehouse—so we don't expect the bot has every possible scenario built out for a 100% point-and-click configuration. Therefore, we leverage invocable Apex methods for the heavy lifting!

We use invocable Apex methods in other product areas, such as Process Builder. As long as the Apex method has a particular annotation (`@InvocableMethod`) and formats its method signature properly, the Bot Builder is able to list this method as an available bot action to call. The builder recognizes the method signature, so you can specify which variable you pass in as an input parameter and which variable you use to store the return value. Then you can merge the returned variable to a message, call another action, or do anything else you can do with a variable.

In a typical order status use case, you assign an `OrderNumber` variable as an input parameter of an Apex method and return the status to another variable `OrderStatus`.

1. Before we write any code, make sure you've created a custom object with a few records, as described in the first step of Gather Customer Information Using the Bot.
2. Now let's build our invocable Apex class and method.

   From Setup, use the Quick Find box to find **Apex Classes**. Add a new class.

```
public with sharing class CookbookBot_GetOrderStatus {

  public class OrderOutput {
    @InvocableVariable(required=true)
    public String sOrderStatus;
  }

  public class OrderInput {
    @InvocableVariable(required=true)
    public String sOrderNumber;
  }

  @InvocableMethod(label='Get Order Status')
  public static List<OrderOutput> getOrderStatus(List<OrderInput> orderInputs) {
    Set<String> orderNumbers = new Set<String>();

    // Get the order numbers from the input
    for (OrderInput orderInput : orderInputs) {
      orderNumbers.add(orderInput.sOrderNumber);
    }

    // Get the order objects from the set of order numbers
```

```
    List<Bot_Order__c> orders =
      [SELECT Name, Status__c FROM Bot_Order__c where Name in :orderNumbers];

    // Create a map of order numbers and order status values
    Map<String, String> mapNameStatus = new Map<String, String>();
    if (orders.size() > 0) {
        for (Bot_Order__c order : orders) {
          mapNameStatus.put(order.Name, order.Status__c);
        }
    }

    // Build a list of order status values for the output
    List<OrderOutput> orderOutputs = new List<OrderOutput>();
    for (OrderInput orderInput : orderInputs) {
      OrderOutput orderOutput = new OrderOutput();

      // Do we have a status for this order number?
      if (mapNameStatus.containsKey(orderInput.sOrderNumber)) {
          // If so, then add the status
          orderOutput.sOrderStatus = mapNameStatus.get(orderInput.sOrderNumber);
      } else {
          // If not, then add an unknown status value
          orderOutput.sOrderStatus = 'Order not found';
      }
      orderOutputs.add(orderOutput);
    }

    return orderOutputs;
  }
}
```

**Important:** As mentioned in Set Up Your First Einstein Bot, you must give the bot permission to access an Apex class. From Setup, use the Quick Find box to find **Permission Sets**. Select the **sfdc.chatbot.service.permset** permission set. From the permission set options, select **Apex Class Access**. Now you can edit the access settings so that your new class is accessible to the bot. Always make sure that the bot has permission to access objects and classes as required.

Some notes on the code:

- The public class `OrderInput` and `OrderOutput` are wrapper classes that include all the input and output variables. Each input and output parameter in these two classes has the `@InvocableVariable` annotation.
- The `getOrderStatus()` method has the `@InvocableMethod` annotation. This method performs a SOQL query to find the order records by order number, then returns the status. All the extra code using `for` loops is to make sure that this method works for bulk invocations.
- The class is `public` in this sample code. Change the visibility to `global` if you plan on including this class in a package.

Some of the key considerations for `@InvocableVariable` and `@InvocableMethod` are:

- When you open the Bot Builder, all invocable actions in the org are loaded.
- The user-friendly label attribute in an `@InvocableMethod` annotation is listed in the Bot Builder as the name of the available action.
- The input and output parameters of the core method have to be `List<DataType>` because `@InvocableMethod` needs a common mechanism to allow the method to be bulk proof. For example, you could have a Process Builder that mass-updates all children records when a parent record changes. The first element of the output list would be the return value for the first element of the input list. The second output element maps to the second input element, and so on.

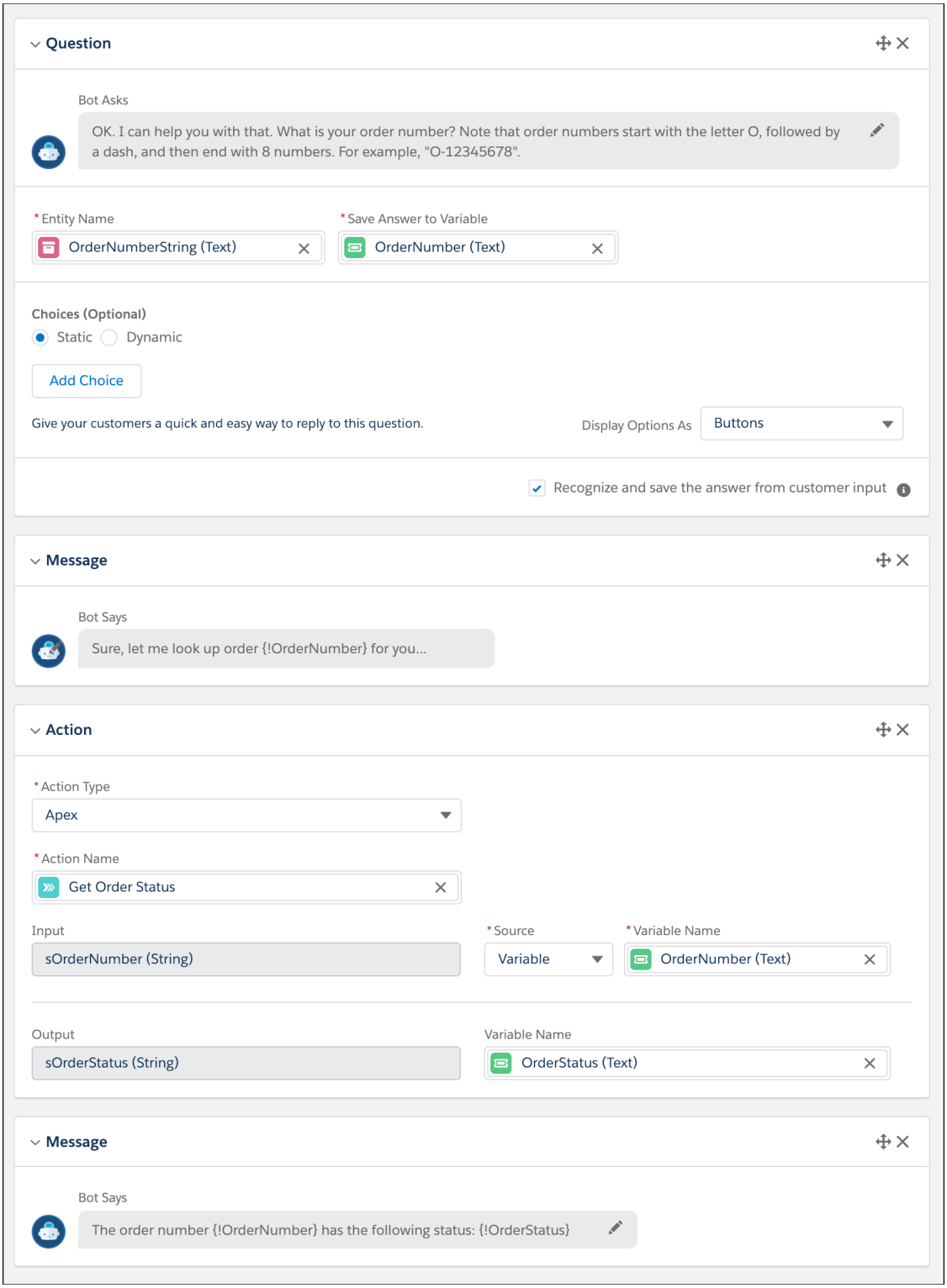
- For most use cases, you can get away with not having it be bulk proof. The code can simply operate on the first element of each input/output list. We made our function bulk proof anyway to illustrate the right way to build the invocable method in case the code is used in other scenarios.
- The public class `OrderInput` and `OrderOutput` are optional when you only have one input parameter and one output parameter. However, if you have more than one input or output parameter, you have to wrap them in a class, as shown in our example. This is because `@InvocableMethod` can only take one input parameter. For detailed information and other considerations about `@InvocableMethod`, see the Apex Developer Guide on the `InvocableMethod` Annotation.
- If you are using a wrapper class for input and output parameters, make sure to add the `@InvocableVariable` annotation. Otherwise the Bot Builder doesn't expose this parameter for you to pass in any data from a variable. For detailed information and other considerations about `@InvocableVariable`, see the Apex Developer Guide on the `InvocableVariable` Annotation.

3. Create an Apex Action from Bot Builder.

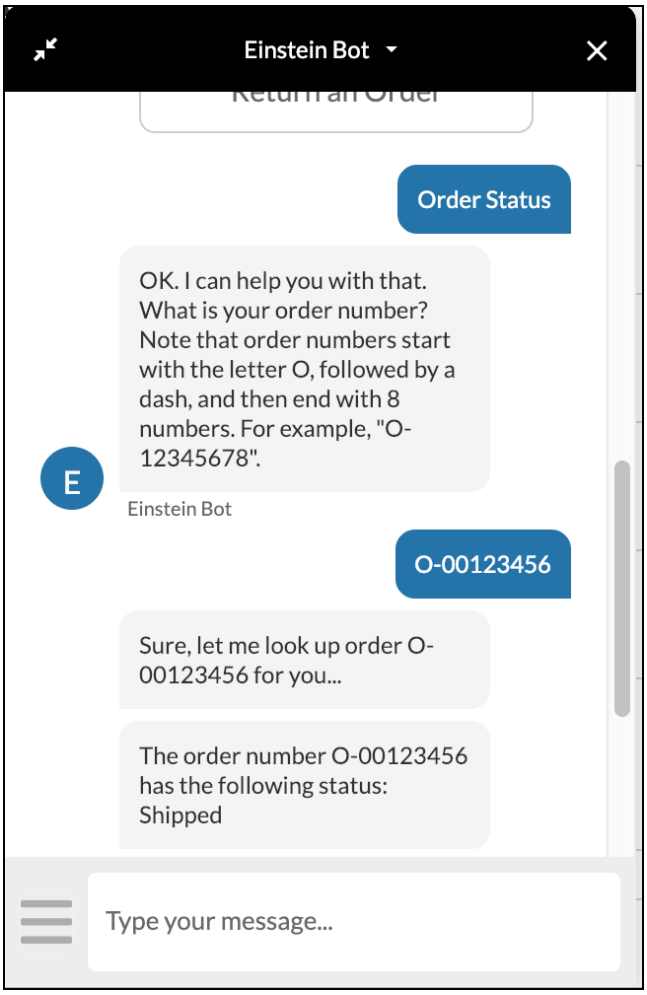
   Open the "Order Status" dialog that we created in Prompt Customer with Menu Options. After asking for the **OrderNumber**, we will add a new **Action**. Select the **Apex** action type. Click the magnifier next to **Action Name** field. You see all the Apex methods with `@InvocableMethod` annotation listed. Select **Get Order Status**. The Bot Builder automatically parses the method signature and populates additional fields for admins to map the variables to input and output invocable variables. If you go back to our `GetOrderStatus` Apex class, you can see that we have one input invocable variable, `sOrderNumber`, and one output invocable variable, `sOrderStatus`. They are exactly what the Bot Builder recognizes and lists as the parameters for this action. Use **OrderNumber** as the input variable. Create a text variable called **OrderStatus** for the output variable.

   Let's create another message and include the `{!OrderStatus}` variable into the response back to the customer with this message: `The order number {!OrderNumber} has the following status: {!OrderStatus}`.

   This screenshot illustrates roughly how your dialog looks.




4. Now we can successfully retrieve the customer order status from the `Bot_Order__c` object!




Our example is simple, but it opens the door to Apex code, from which you can leverage almost everything Apex has to offer for integration tasks: External Object, Apex Callout, Salesforce Connect, Salesforce Integration... you name it!

# Call a Flow Action

In this recipe, we continue building out our order support functions with a flow that allows a customer to cancel an order.

**Channels**

- Chat
- Messaging

**Prerequisites**

- Set Up Your First Einstein Bot on page 1
- Greet the Customer on page 6
- Prompt Customer with Menu Options on page 29

**In this recipe you learn how to:**

- Create an autolaunched flow with Flow Builder.

- Launch a flow with Bot Builder.
- Pass variables back and forth between Bot Builder and Flow Builder.
- Get information from the user and perform an action based on that information.
- Return a result to the user after performing an action.

Tip: Apex Action or Flow Action? We provide one recipe to introduce you to flow actions and another recipe to introduce you to Apex actions. You'll quickly see that you can do many of the same things with both solutions. Which one should you use? Think about the flow option first for simple solutions. It's declarative and is easier to maintain. Apex, on the other hand, offers much power and many more use cases not available with a simple flow. Another option is to use both. You could add Apex actions to a flow. From within **Flow Builder**, just drag an **Apex Action** element onto the canvas. Or, you can invoke a flow from Apex. After some experience with both flows and Apex, you'll be able to determine how to most effectively use these tools together to get the job done.

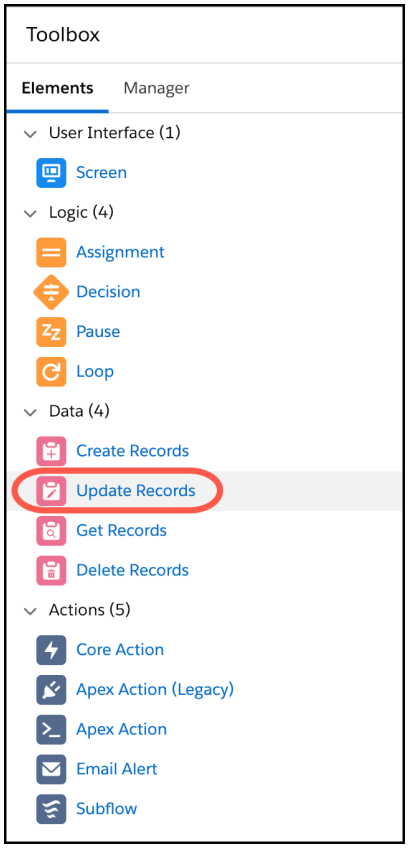
Since we are using a custom object on the platform (`Bot_Order__c`) to store the Order information, a CRUD (create, read, update, and delete) operation such as a record update can be configured in an autolaunched flow.

1. Before we write any code, make sure you've created a custom object with a few records, as described in the first step of Gather Customer Information Using the Bot.
2. Create an autolaunched flow.

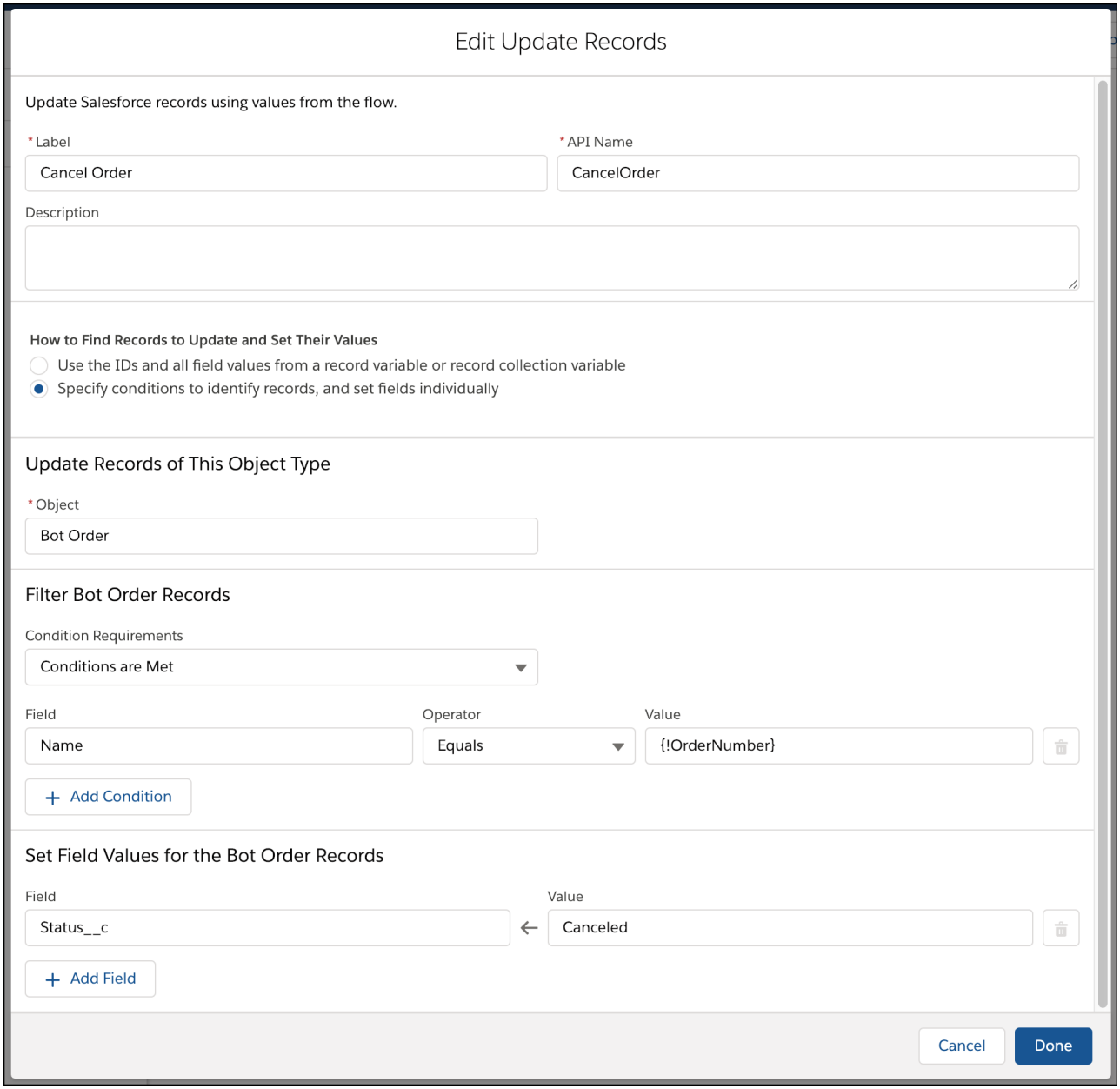
   Creating a flow is typically faster than writing Apex code. If you haven't worked on a flow before, see Flows in Salesforce Help.

   Note: As you work on this flow, we'll explain in more detail how the interaction works between the Bot Builder and the Flow Builder. At a high level, you need to make sure that all the flow variables that you want to set from Bot Builder are "available for input". And all flow variables that you want to pass back to Bot Builder are "available for output".

   From Setup, use the Quick Find box to find **Flows**. Create a new flow. The goal of this flow is to update a particular order based on the order number provided by the customer and set its status to **Canceled**. So, we'll start with an **Update Records** element.

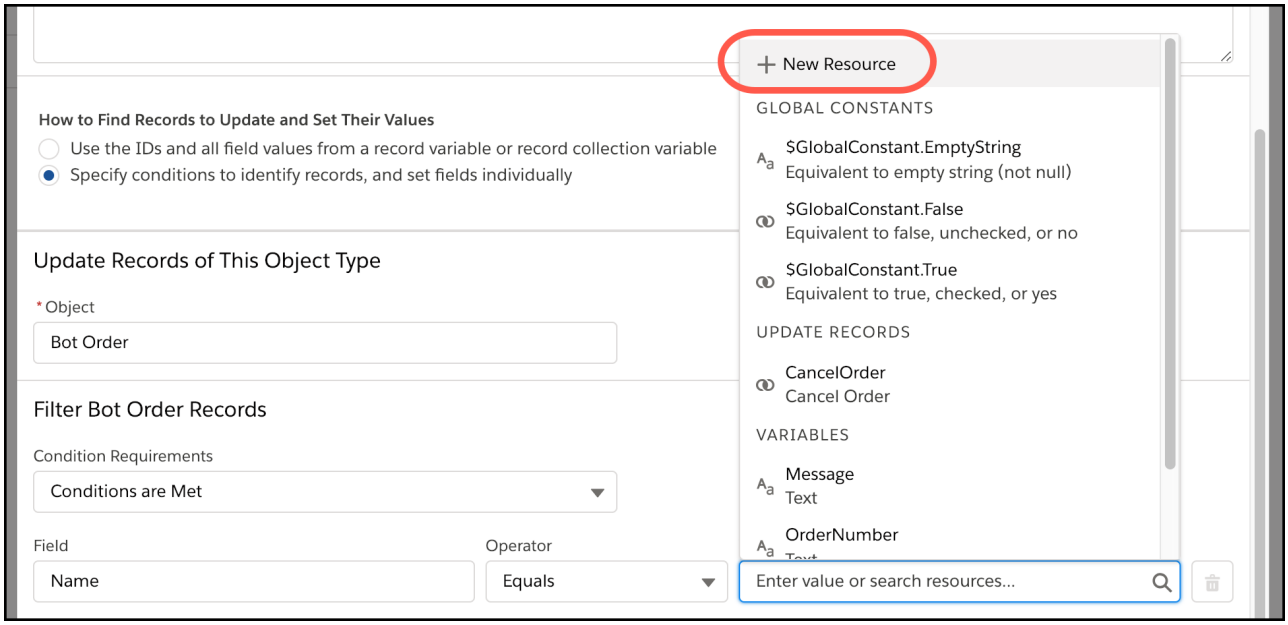



In Flow Builder, drag the **Update Records** element from the Toolbox onto the canvas. To find a record, select **Specify conditions to identify records, and set fields individually**. You can then search for all orders with the specified order number and set the status field to Canceled.

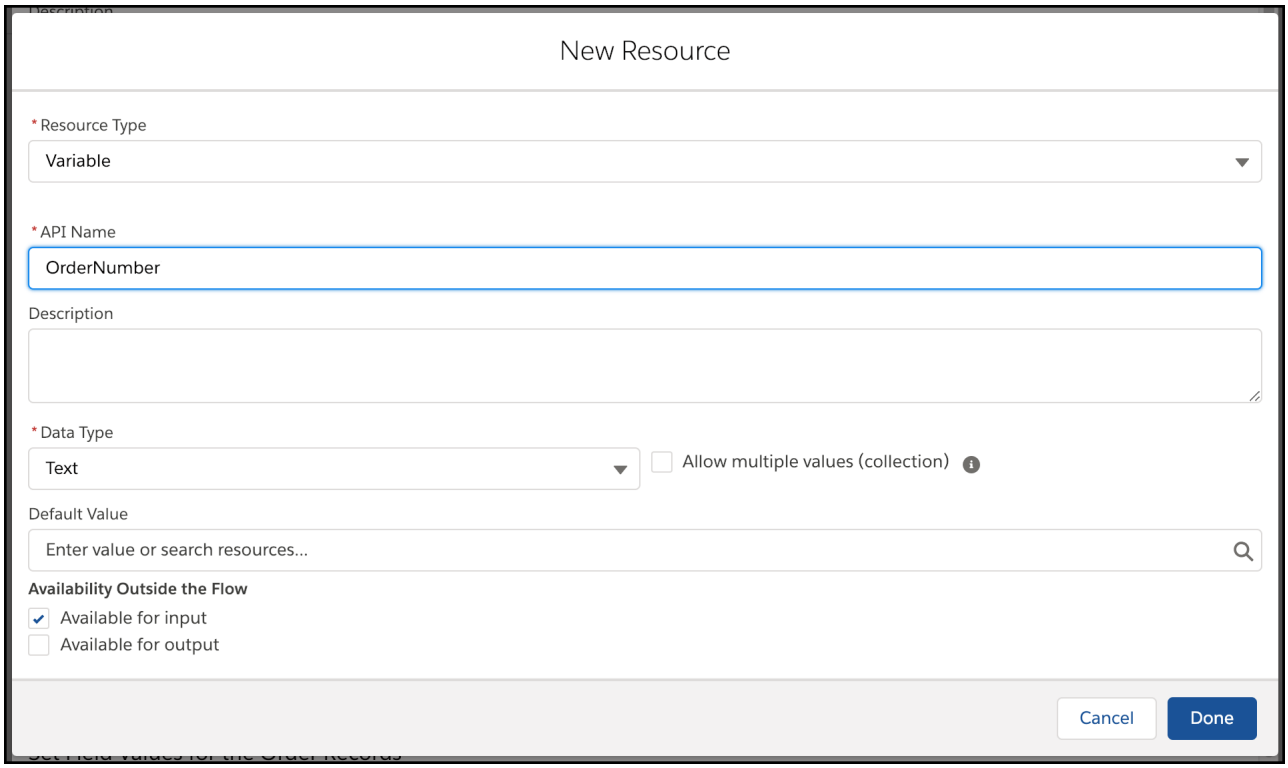



The previous screenshot shows the desired values for the **Update Records** element.

To build the condition, create an **OrderNumber** variable in the **Value** field. After clicking the **Value** field, select **New Resource** from the popup menu.



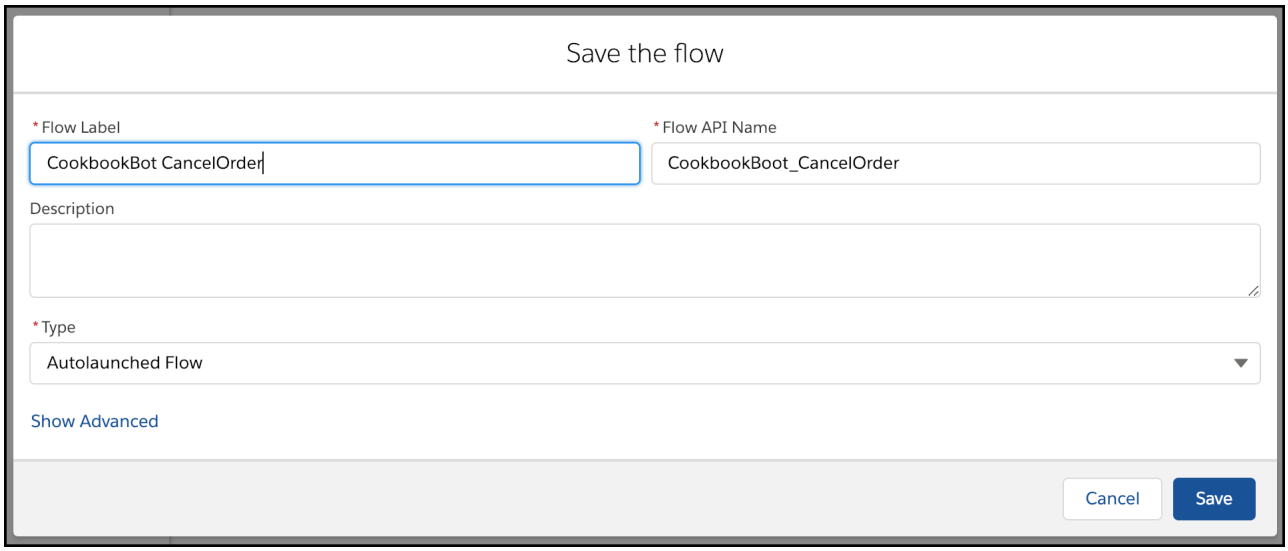

Now create a new **Variable** Resource Type and use the **Text** Data Type. Enter `OrderNumber` for the API name.




Note: Be sure to make this resource **Available for input**. This checkbox allows the Bot Builder to use this variable as an input parameter to the flow.

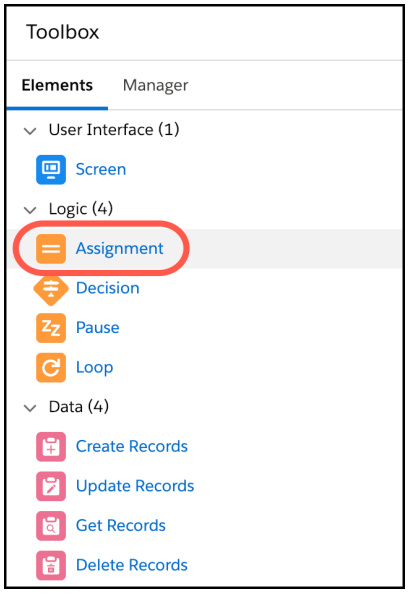
Once you complete this step, the Value field should get set to `{!OrderNumber}`.

Now, you can go ahead and save the flow. Give it a unique label such as `CookbookBot CancelOrder` and specify the type as **Autolaunched Flow**. Other types such as **Screen Flow** don't apply here, because there is no UI element and this flow is triggered by a bot as a server-side process.

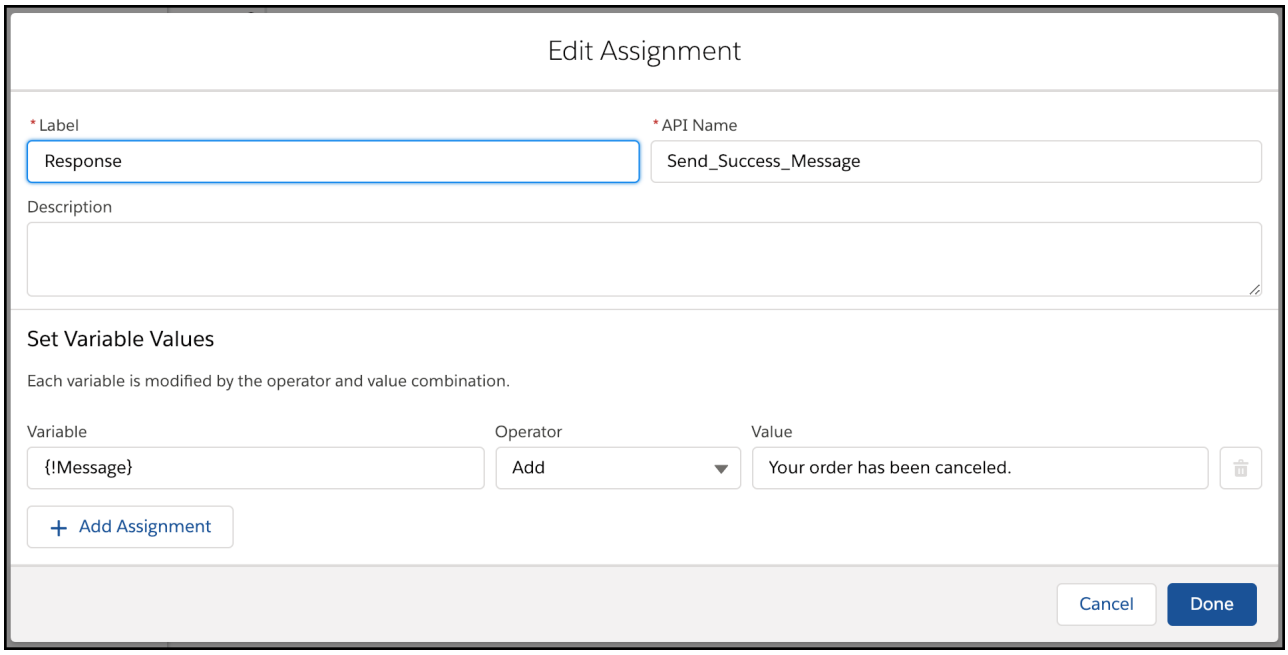



3. Add some basic logic to the flow.

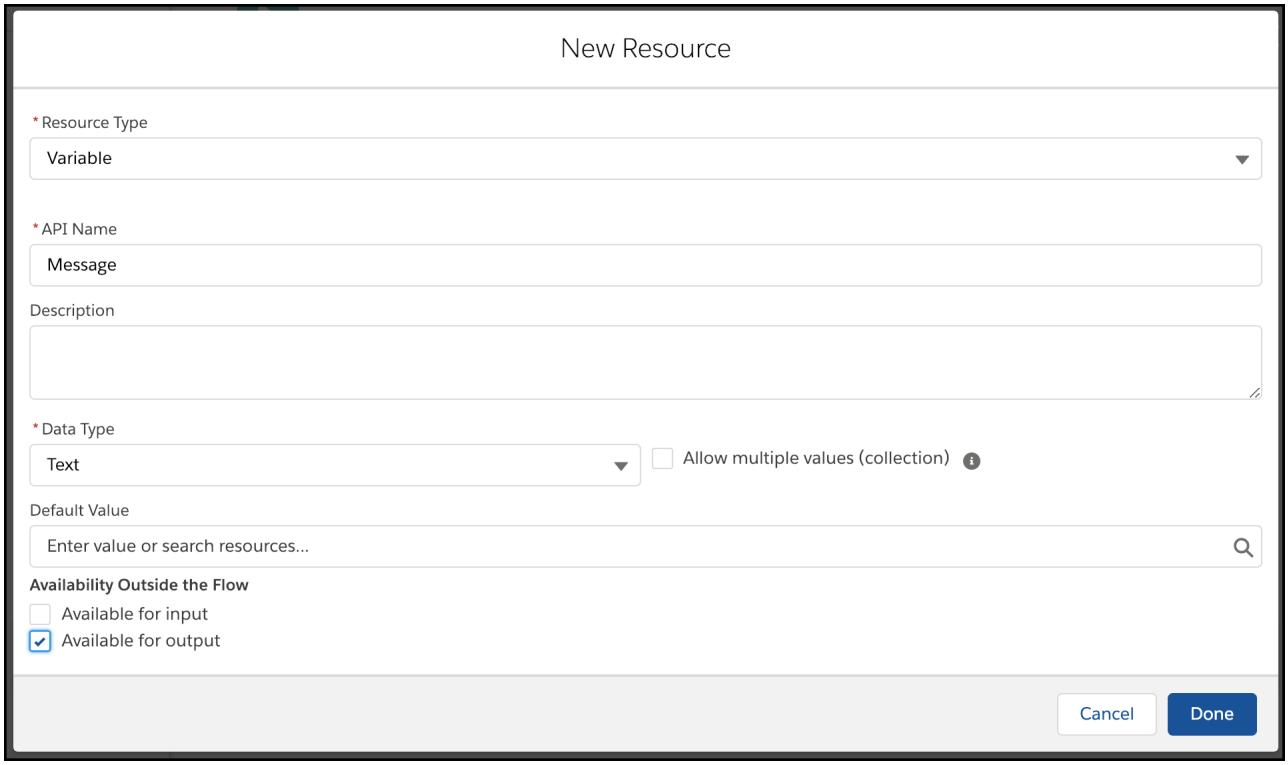
   At the simplest level, we could wire up a flow that updates the record and then returns a message. To make this basic flow, add an **Assignment** element to your flow.




   Edit the new assignment.

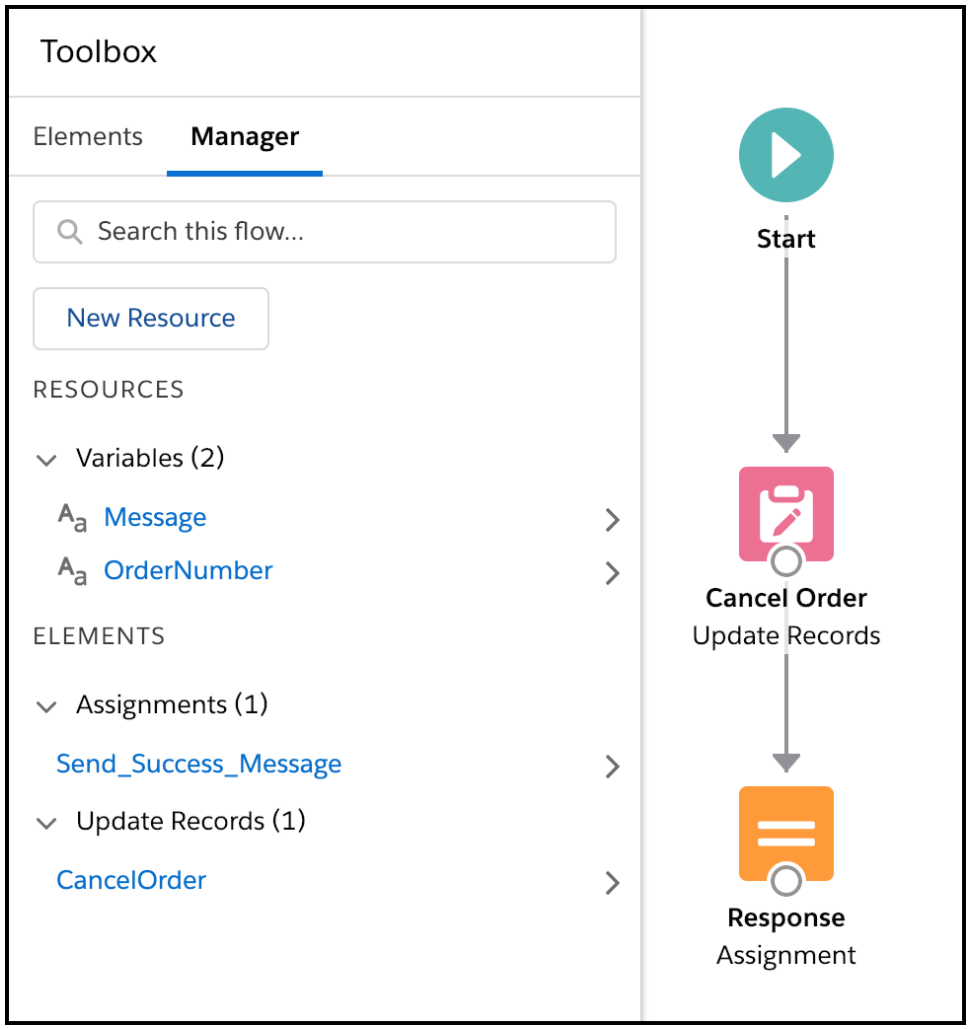



We need to create another variable for this assignment. Create a `Message` variable from the Variable field and specify the **Text** Data Type.




Note: Be sure to make this resource **Available for output**. This checkbox allows the Bot Builder to access this variable as an output parameter from the flow.

This screenshot shows the flow once it's wired up.




The **Manager** subtab in the Toolbox lists all the constants and variables that are used in this flow. One thing to double-check—we must have **OrderNumber** set as the **Input** variable and **Message** set as the **Output** variable.

Important: As mentioned in Set Up Your First Einstein Bot, you need to give the bot permission to access your flow. From Setup, use the Quick Find box to find **Permission Sets**. Select the **sfdc.chatbot.service.permset** permission set. From the permission set options, select **System Permissions**. Enable the **Run Flows** permission.

This flow is definitely not ready for prime time. It doesn't check if the order exists, or if the order has been returned or canceled, in which case we shouldn't let the customer cancel again. But don't worry, flow supports the necessary logic to make our flow more robust.

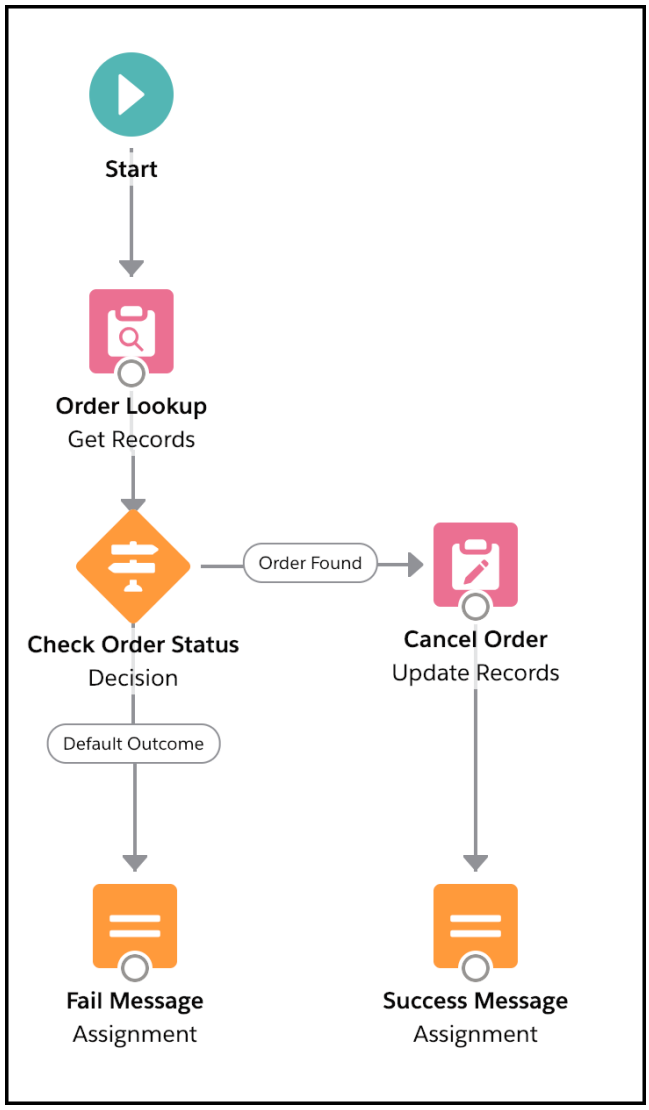
If you only need to understand how to trigger a flow action from Bot, you can probably jump to the next step. But if you are new to flow, now is a great time to learn about additional flow capabilities. You can design extremely powerful flows without even writing a line of code. To learn more, see Flows in Salesforce Help.

4. Add more robust logic to the flow.

   Use what you've learned about building a flow in the previous step to add more complex logic to the flow to handle the situation where the order can't be found. In particular:

- Use the **Get Records** element to perform an order lookup.
- Use a **Decision** element to determine if you found the order.
- If the order was found, connect the decision to your existing cancel order flow.
- Use another **Assignment** element to set a failure message if you can't find the order.

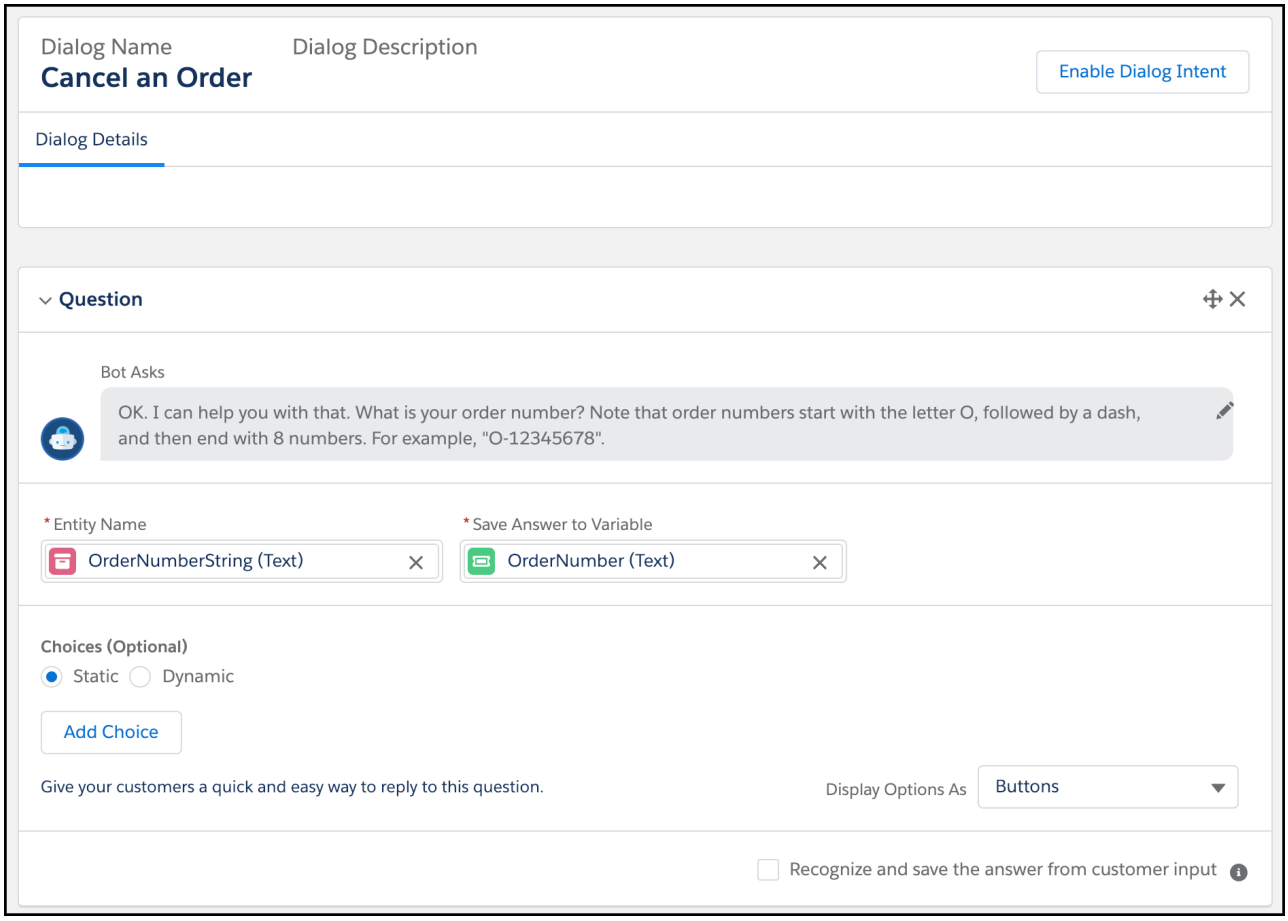
See the screenshot below for an illustration of the flow with additional logic to check the order status and cancel orders.



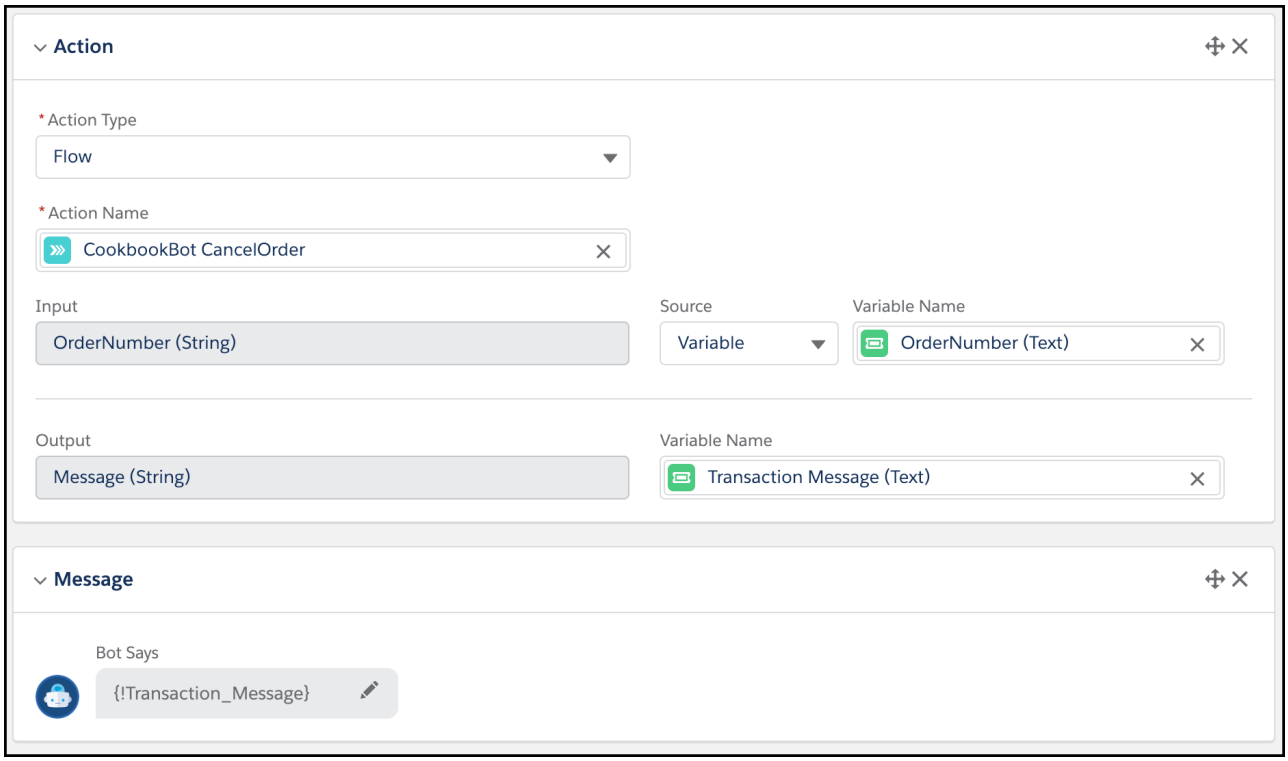

5. Create a flow action from Bot Builder.

   Open the "Cancel Order" dialog that we created in Prompt Customer with Menu Options. Similar to the "Order Status" dialog, ensure that this new dialog starts with us asking the customer a question. Then we call our action. And finally, we use the results of our action to send a message to the customer.

   The question element is the same as in Call an Apex Action. We save the customer's answer to the `OrderNumber` variable with an entity named `OrderNumberString`.



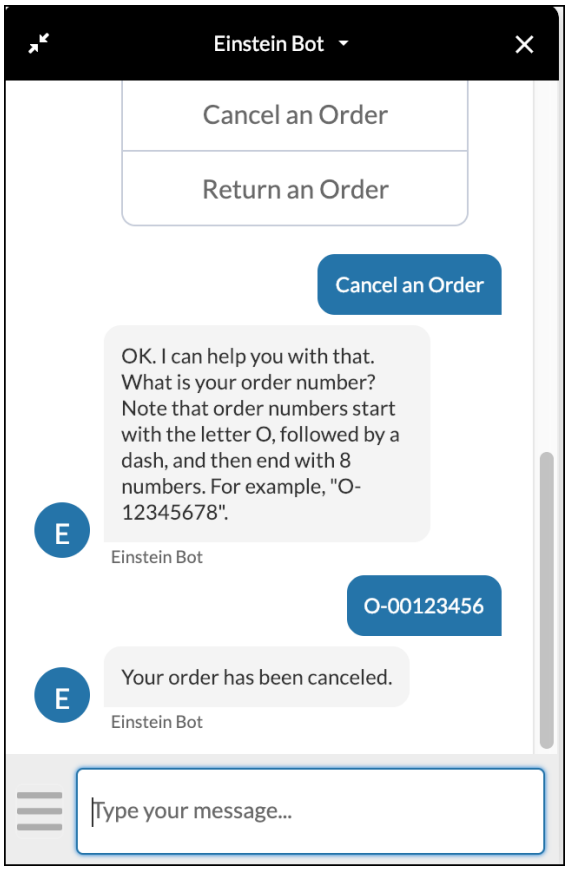

In the action element, we need to first select **Flow** as the **Action Type**. With that, Bot Builder lists all the available flow actions. Select the **Cookbook Bot CancelOrder** action. Notice that the input and output variable mapping are automatically expanded for the `OrderNumber` and `Message` input variables with placeholders. We need to create a new bot variable for the output. Let's name it **Transaction_Message**.




Finally, in the message element, we use the `{!Transaction_Message}` variable as the message to the customer.

6. Save and test the flow!

   Now, we can support customers canceling a newly placed order.




# Optimize Bot Flow with Pre-Chat Data

In this recipe, we gather information from a pre-chat form and use this data to improve the user experience.

**Important:** This feature only works when using your bot for Chat. It does not apply to Messaging.

Sometimes, if one piece of information is critical to help a customer with most inquiries, you may want to gather the information in a pre-chat form before the customer even starts a conversation, such as the order number. From the Greet the Customer section, we already know how to get a customer name through a pre-chat form. In this chapter, we'll take it one step further and show you how to gather inquiry-related information from a pre-chat form and use this data to improve the user experience.

Optimize Bot Flow with Embedded Chat
In this recipe, we use Embedded Chat to gather information from a pre-chat form.

Optimize Bot Flow in Salesforce Classic
In this recipe, we use Live Agent in Salesforce Classic to gather information from a pre-chat form.

## Optimize Bot Flow with Embedded Chat

In this recipe, we use Embedded Chat to gather information from a pre-chat form.

**Prerequisites**

- Set Up Your First Einstein Bot on page 1
- Greet the Customer on page 6
- Prompt Customer with Menu Options on page 29

**In this recipe you learn how to:**

- Create a custom field on an existing object.
- Add information to the pre-chat form.
- Update your site's code snippet so the bot can access this new information.
- Use Apex code or the Metadata API to access the custom field.
- Optimize the bot design.

Before beginning this recipe, be sure you've completed the Greet the Customer recipe. These steps build on that previous section.

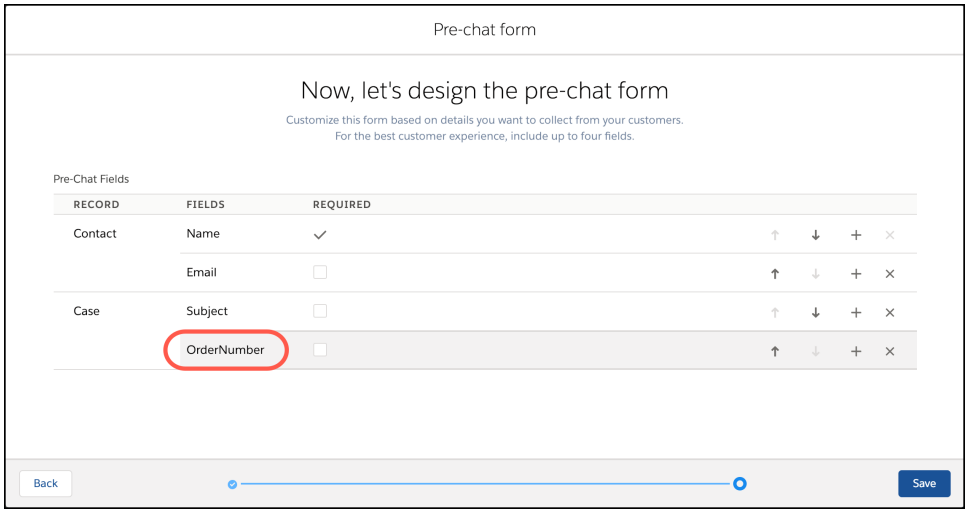
1. Create custom fields for the order number.

   First of all, we need to create `OrderNumber` custom fields on both the `LiveChatTranscript` and the `Case` objects. As explained in Greet the Customer, the bot pre-chat form action looks for fields on the `LiveChatTranscript` record to pull in pre-chat information. As for the `Case` field, Embedded Chat's pre-chat configuration only uses `Contact` and `Case` fields to build the form, thus we need the field on the `Case` object as well.

   Go ahead and create two **Text** custom fields, one on `Case` and one on `LiveChatTranscript`. Both can be labeled as `OrderNumber` with the API name `OrderNumber__c`.

2. Add the `Case.OrderNumber` field to our pre-chat page configuration.

   From Setup, use the Quick Find box to find **Embedded Service**. View your Embedded Service deployment. Edit your **Chat Settings**. Edit your **Pre-chat page**. From the Case row, add a new field and choose **OrderNumber**.




3. Update your Embedded Chat code with pre-chat info.

   As far as the bot accessing the pre-chat data, similar to the Greet the Customer section, there are a couple of options. You can use developer code to populate the custom fields on the `LiveChatTranscript` object using the same data from a contact or case. Or, you can call an Apex action to get the proper data from the parent record. In this tutorial, let's use the first method to populate custom fields on the `LiveChatTranscript` object.

   The code is slightly different depending on whether you have a customer website or an Experience Cloud site.

**a.** (Option 1 of 2) Update your code on a customer site.

If you are implementing Embedded Chat on a non-Salesforce-hosted website, embed the code snippet generated from your deployment. From Setup, use the Quick Find box to find **Embedded Service**. View your deployment. For more details, refer to the instructions in the Greet the Customer section.

Let's populate data on the `LiveChatTranscript` custom fields so the bot's pre-chat form action can access the data. We need to update the Embedded Service code. Specifically, we need to update the `embedded_svc.settings.extraPrechatFormDetails` array to include the order number pre-chat field and map it to the associated `LiveChatTranscript` custom field.

```
embedded_svc.settings.extraPrechatFormDetails = [{
  "label":"First Name",
  "transcriptFields": ["FirstName__c"]
},{
  "label":"Last Name",
  "transcriptFields": ["LastName__c"]
},{
  "label":"Email",
  "transcriptFields": ["Email__c"]
},{
  "label":"OrderNumber",
  "transcriptFields": ["OrderNumber__c"]
}];
```

**b.** (Option 2 of 2) Update your code using an Experience Cloud site.

If you are using an Experience Cloud site, you have to update the `SnippetSettingsFile` static resource and include additional mapping for the `OrderNumber` field. For more details, revisit the Greet the Customer section where we created the initial version. This is what your `CookbookBot_SnippetSettingsFile` static resource should look like now:

```
window._snapinsSnippetSettingsFile = (function() {

  // Logs that the snippet settings file was loaded successfully
  //console.log("Snippet settings file loaded.");

  embedded_svc.snippetSettingsFile.extraPrechatFormDetails = [{
    "label":"First Name",
    "transcriptFields": ["FirstName__c"]
  },{
    "label":"Last Name",
    "transcriptFields": ["LastName__c"]
  },{
    "label":"Email",
    "transcriptFields": ["Email__c"]
  },{
    "label":"OrderNumber",
    "transcriptFields": ["OrderNumber__c"]
  }];
})();
```

**4.** Get access to the `OrderNumber` custom field in the `LiveChatTranscript` record using either the Salesforce Metadata API or an invocable Apex method.

Now just like how we discussed in the Greet the Customer with Embedded Service Chat section, there are two ways to get access to this `OrderNumber` custom field.

**a.** Write Apex code to grab what you need and make the Apex method accessible to the Bot Builder using the `@InvocableMethod` annotation. For instance, if you're using Omni-Channel, you can use the `RoutableId` context variable to get the `LiveChatTranscript` record, extract the custom field value, and then pass it back to your bot.

**b.** Use the Salesforce Metadata API to add context variables that aren't available out of the box. For instance, you can create a context variable to gain access to a `LiveChatTranscript` custom field.

We give you guidance on how to perform either method.

**a.** (Option 1 of 2: **Apex Code Solution**) If you want to extract the `OrderNumber` field from the `LiveChatTranscript` object using an invocable Apex method, review the Apex Solution in the Greet the Customer section on page 12 and then follow these additional instructions.

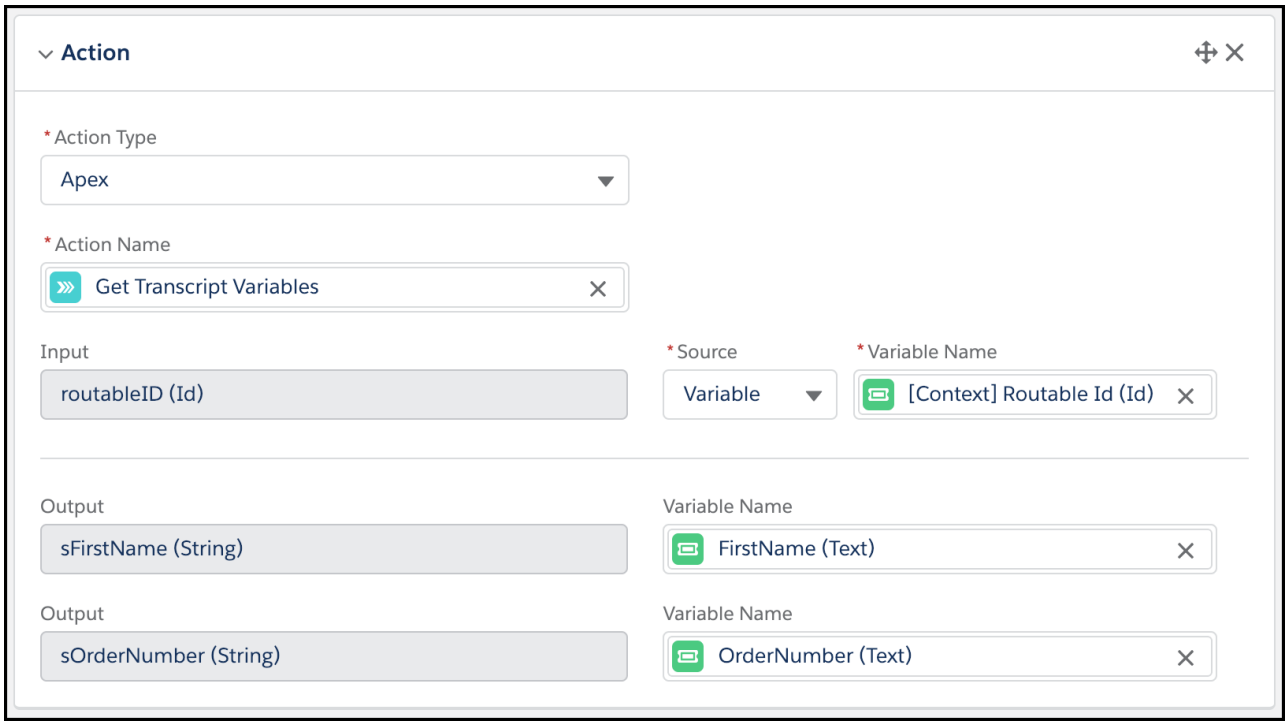
This time, you need to update your Apex code so that it grabs the order number from the `LiveChatTranscript` record. Use this new code in place of the code you previously created. You see that it's almost identical, except that it grabs the order number and puts it in a new output variable (`sOrderNumber`).

```
public with sharing class CookbookBot_GetTranscriptVariables {

  public class TranscriptInput {
    @InvocableVariable(required=true)
    public ID routableID;
  }

  public class VisitorNameOutput {
    @InvocableVariable(required=true)
    public String sFirstName;

    @InvocableVariable(required=true)
    public String sOrderNumber;
  }

  @InvocableMethod(label='Get Transcript Variables')
  public static List<VisitorNameOutput> getUserName(List<TranscriptInput> transcripts)
 {

    List<VisitorNameOutput> names = new List<VisitorNameOutput>();

    for (TranscriptInput transcript : transcripts) {

      // Query for the transcript record based on the ID
     LiveChatTranscript transcriptRecord = [SELECT Name, FirstName__c, OrderNumber__c

                                            FROM LiveChatTranscript
                                            WHERE Id = :transcript.routableID
                                            LIMIT 1];

      // Store the first name in an output variable
      VisitorNameOutput nameData = new VisitorNameOutput();
      nameData.sFirstName = transcriptRecord.FirstName__c;

      // Store the order number in an output variable
      nameData.sOrderNumber = transcriptRecord.OrderNumber__c;

      // Add the name to the list of outputs
      names.add(nameData);
```

```
    }

    return names;
  }
}
```

To pass data from our Apex code to the bot, we'll call the Apex action to set the `OrderNumber` variable along with the `FirstName` variable. We *could* call this action first thing in our "Welcome" dialog directly, similar to how we were able to set the `FirstName` variable. However, in this exercise, let's set up an initialization dialog. This way, we can wrap all the initialization actions in this dialog while keeping our welcome dialog a little easier to read.

Create a "Bot Initialization" dialog that calls our Apex action.



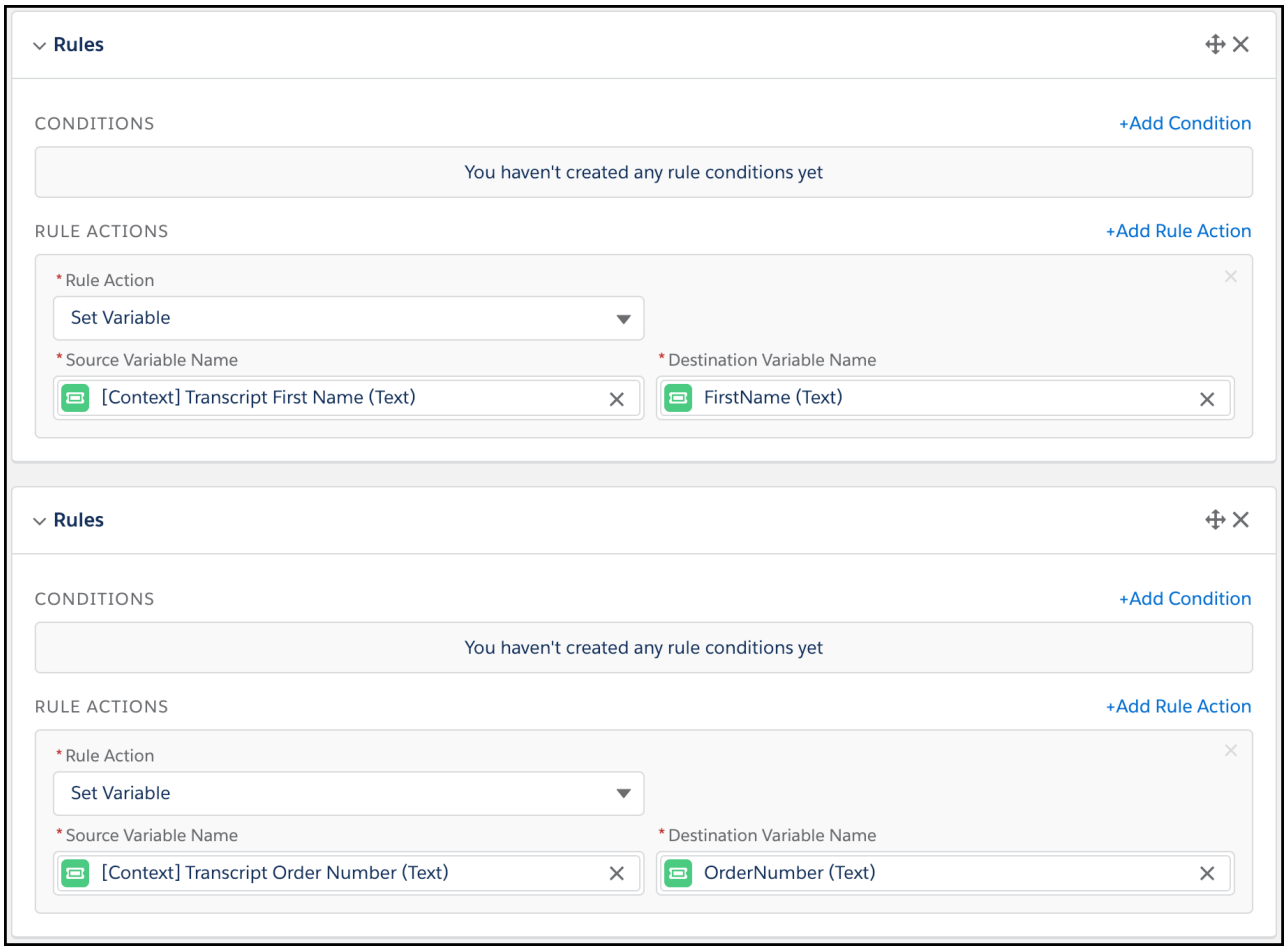

**b.** (Option 2 of 2: **Metadata API Solution**) If you want to extract the `OrderNumber` field from the `LiveChatTranscript` object using the Salesforce Metadata API, review the Metadata Solution in the Greet the Customer section on page 13 and then follow these additional instructions.

This time, add the following XML snippet (in addition to the previously specified XML on page 13 which added the first name) when deploying changes to your org.

```
<contextVariables>
  <contextVariableMappings>
    <SObjectType>LiveChatTranscript</SObjectType>
    <fieldName>LiveChatTranscript.OrderNumber__c</fieldName>
    <messageType>WebChat</messageType>
  </contextVariableMappings>
  <dataType>Text</dataType>
  <developerName>TranscriptOrderNumber</developerName>
  <label>Transcript Order Number</label>
</contextVariables>
```

To use this new context variable in our bot, we call another **Set Variable** rule using the new `Transcript Order Number` context variable (in addition to the previous rule for the `Transcript First Name` context variable). We *could* call this rule first thing in our "Welcome" dialog directly, similar to how we were able to set the `FirstName` variable. However, in this exercise, let's set up an initialization dialog. This way, we can wrap all the initialization actions in this dialog while keeping our welcome dialog a little easier to read.

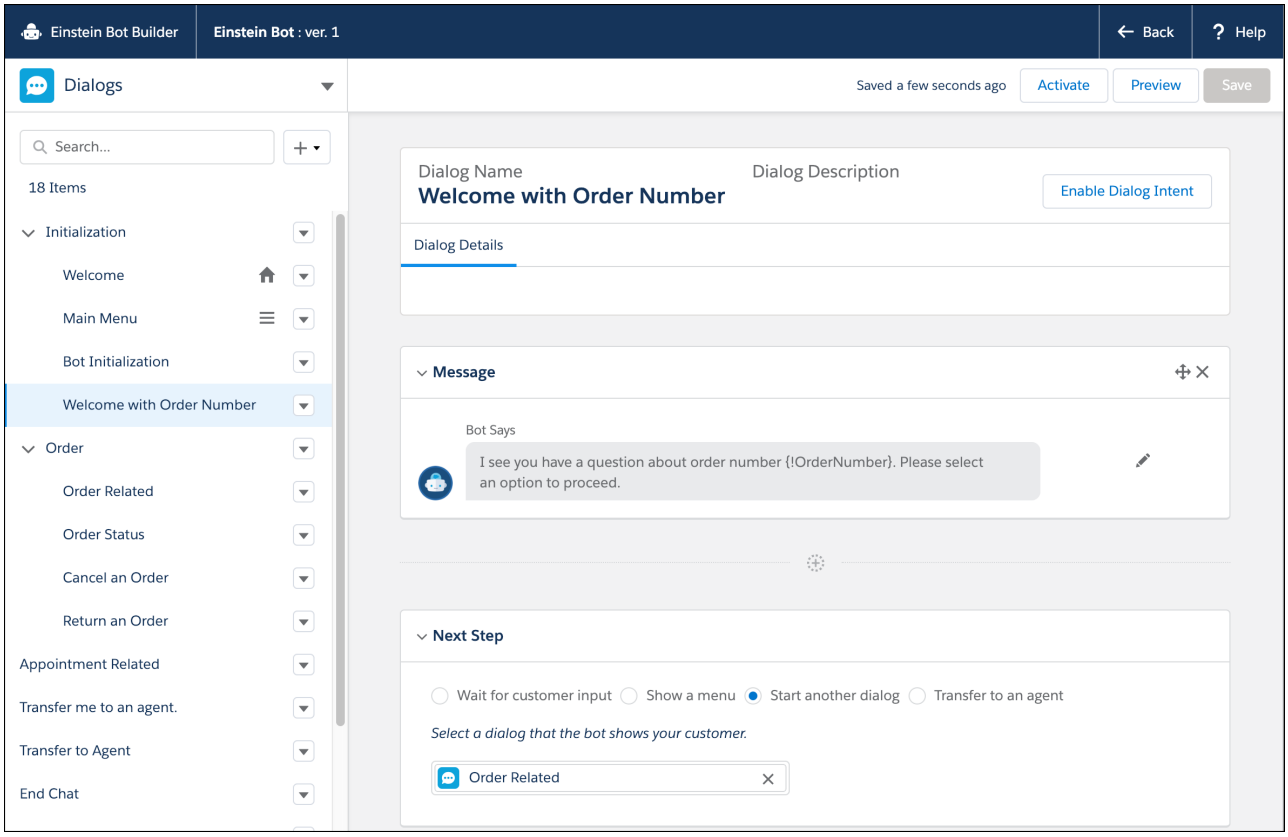
Create a "Bot Initialization" dialog that calls both of our **Set Variable** rules.




5. Optimize the design with rules.

   Let's pause for a minute. So we added a non-required field on pre-chat form. We want this to be flexible so customers don't have to enter the order number if they are interested in other questions. But for those who do provide an order number, I think we can safely assume they have an order-related issue. So wouldn't it be better if we just send them directly to the "Order Related" sub menu? Yep, that's where the rules come in to help us guide users through different steps and optimize their chat experience.

   First, we need to build a new "Welcome with Order Number" dialog to greet the customer in a different way if they provide the order number in the pre-chat form. We use the message text: `I see you have a question about order number {!OrderNumber}. Please select an option to proceed.`




Note the next step from this dialog should be set to the "Order Related" dialog to present the options from the "Order Related" submenu.

Tip: You can group your dialogs in Dialog Groups to make them easier to manage. In this example, we've grouped all the initialization dialogs under one group. To create a group, use the same create button used when creating dialogs.

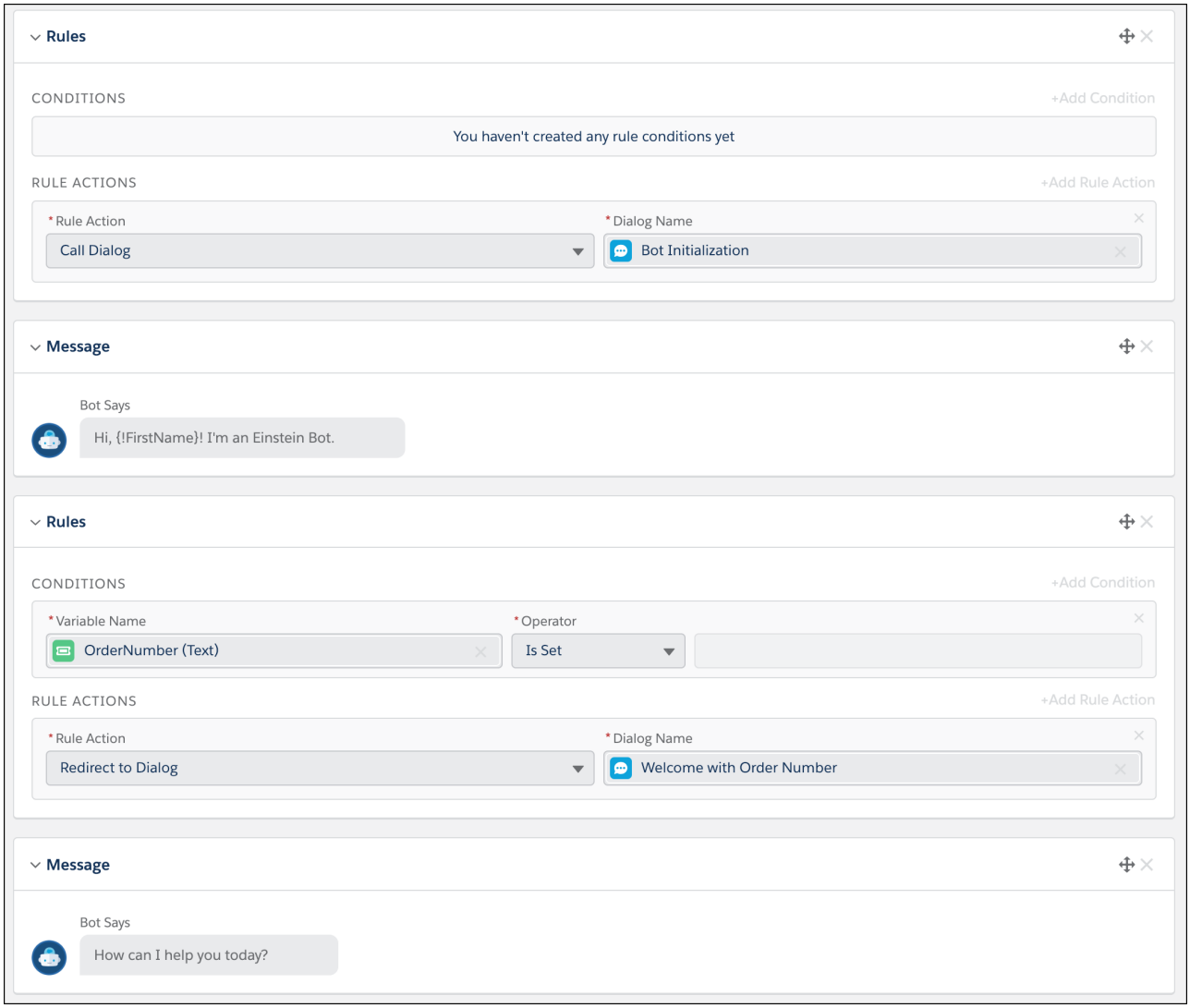
In the original "Welcome" dialog, let's perform the following steps:

**a.** Use a Rule Action to call the "Bot Initialization" dialog we just created to grab the order number from the pre-chat field.

**b.** Create a message to say hi to the customer, using the `FirstName` variable.

**c.** Use a Rule Action that redirects to the "Welcome with Order Number" dialog only if the `OrderNumber` variable has been set.

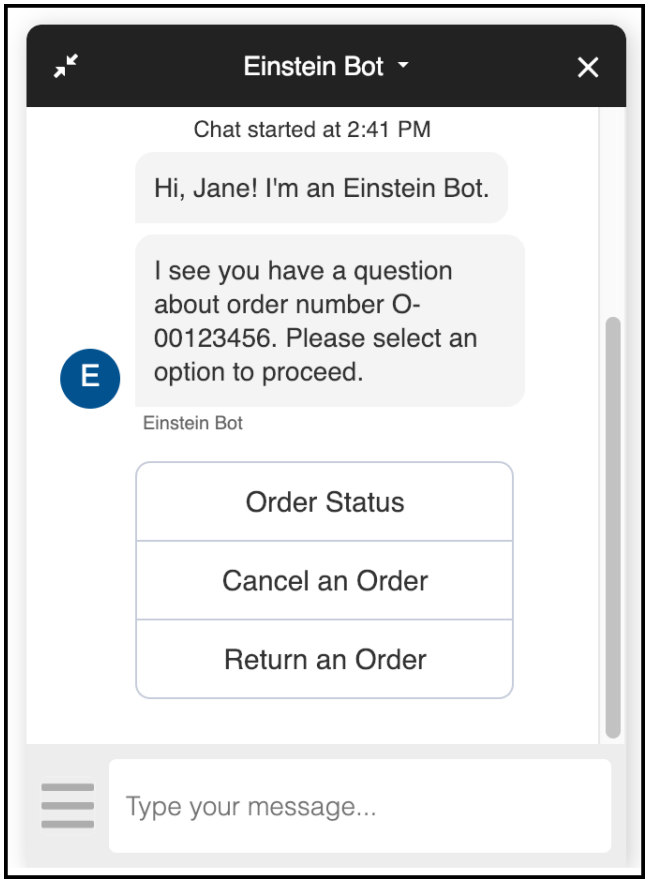
**d.** If the variable has not been set, then just ask the customer what they need help with.



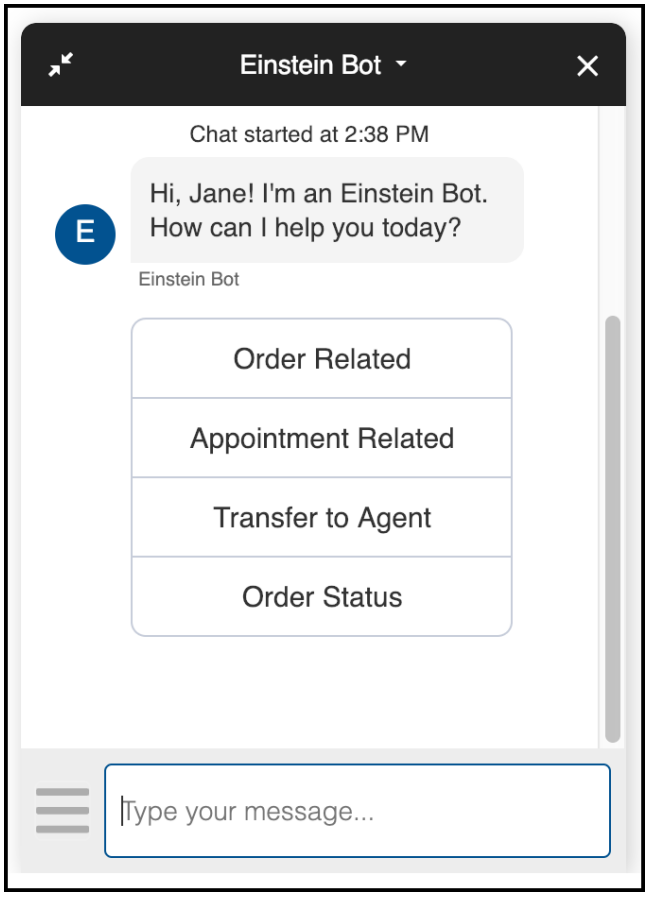

Notice that we split the original greeting message into two parts. We first greet the customer, then we have the rule action before the next message. If the customer doesn't provide an order number in the pre-chat form, we continue with the original question "How can I help you today?" Otherwise, the customer is sent to the "Welcome with Order Number" dialog for order-specific questions.

6. Test!

   Now let's test our optimized user flow. If the customer enters an order number in the pre-chat form, the bot can use that information to get right to the order-specific flow.




Otherwise, the bot falls back to the standard welcome flow.




**Best Practices and Additional Notes**

- **Call Dialog vs. Redirect to Dialog**. In the initial "Welcome" screen, we used two different dialog-related rule actions. So why does Bot Builder provide two rule actions that seem very similar to each other by their names? Well, the **Redirect to Dialog** action indicates the destination dialog will take over the flow at this point; while with **Call Dialog** action, the destination dialog will return the flow back to the original dialog. We are using the **Call Dialog** action with the "Bot Initialization" dialog, because we need to send the user back to welcome screen to continue with the branching logic. Also note that whatever configuration we select in the **Next Dialog** section for "Bot Initialization" dialog is ignored because it's only used in a **Call Dialog** action and flow needs to return back to the calling dialog.
- **Bot Variable Lifecycle**. Did you notice that when you click Order Status, the bot didn't ask for the order number if you already provided the information on pre-chat form? However, if you look at the "Order Status" dialog from Bot Builder, we didn't make it an optional step. So, what happened? One thing about the **Question** action—if the **Answer** variable has any value, it will skip the question. It's a smart feature. For example, let's say the customer provides an order number to check status then later decides to cancel that order. We don't have to ask for the order number again. On the other hand, if you do need to ask the order number again, let's say you want to build a process to allow the customer to check the status for another order, you can use the **Clear Variable** rule action to reset the **OrderNumber** variable first. This way, when the "Order Status" dialog is triggered again, the bot will ask the question again.
- **Variable Validation**. Have you tried to use an invalid order number on the pre-chat form (one that doesn't match the regular expression for entity **OrderNumberString**)? What happened? Think about what can we do to validate that the customer input is indeed an order number before skipping the order number question in the bot.

# Optimize Bot Flow in Salesforce Classic

In this recipe, we use Live Agent in Salesforce Classic to gather information from a pre-chat form.

**Prerequisites**

- Set Up Your First Einstein Bot on page 1
- Greet the Customer on page 6
- Prompt Customer with Menu Options on page 29

**In this recipe you learn how to:**

- Create a custom field on an existing object.
- Add information to the pre-chat form.
- Update your site's code snippet so the bot can access this new information.
- Use Apex code or the Metadata API to access the custom field.
- Optimize the bot design.

1. Create custom fields for the order number.

   First of all, we need to create `OrderNumber` custom fields on both the `LiveChatTranscript` and the `Case` objects. As explained in Greet the Customer, the bot pre-chat form action looks for fields on the `LiveChatTranscript` record to pull in pre-chat information. As for the `Case` field, Embedded Chat's pre-chat configuration only uses `Contact` and `Case` fields to build the form, thus we need the field on the `Case` object as well.

   Go ahead and create two **Text** custom fields, one on `Case` and one on `LiveChatTranscript`. Both can be labeled as `OrderNumber` with the API name `OrderNumber__c`.

2. Add an input field to the pre-chat Visualforce page.

   Continuing from the previous Greet the Customer example, let's add `liveagent.prechat:OrderNumber` as a new pre-chat input field. Add this code to the Visualforce pre-chat page, next to the `FirstName`, `LastName` and `Email` input fields. The second line in this example uses the pre-chat API to specify that data gathered in this chat variable should be stored in the `OrderNumber__c` custom field on the `LiveChatTranscript` object.

   ```
   Order Number: <input type='text' name='liveagent.prechat:OrderNumber'
                       id='OrderNumber' /><br />
   <input type="hidden" name= "liveagent.prechat.save:OrderNumber"
                       value= "OrderNumber__c" />
   ```

3. Get access to the `OrderNumber` custom field in the `LiveChatTranscript` record using either the Salesforce Metadata API or an invocable Apex method.

   Now just like how we discussed in the Greet the Customer with Embedded Service Chat section, there are two ways to get access to this `OrderNumber` custom field.

   a. Write Apex code to grab what you need and make the Apex method accessible to the Bot Builder using the `@InvocableMethod` annotation. For instance, if you're using Omni-Channel, you can use the `RoutableId` context variable to get the `LiveChatTranscript` record, extract the custom field value, and then pass it back to your bot.

   b. Use the Salesforce Metadata API to add context variables that aren't available out of the box. For instance, you can create a context variable to gain access to a `LiveChatTranscript` custom field.

   We give you guidance on how to perform either method.

**a.** (Option 1 of 2: **Apex Code Solution**) If you want to extract the `OrderNumber` field from the `LiveChatTranscript` object using an invocable Apex method, review the Apex Solution in the Greet the Customer section on page 12 and then follow these additional instructions.

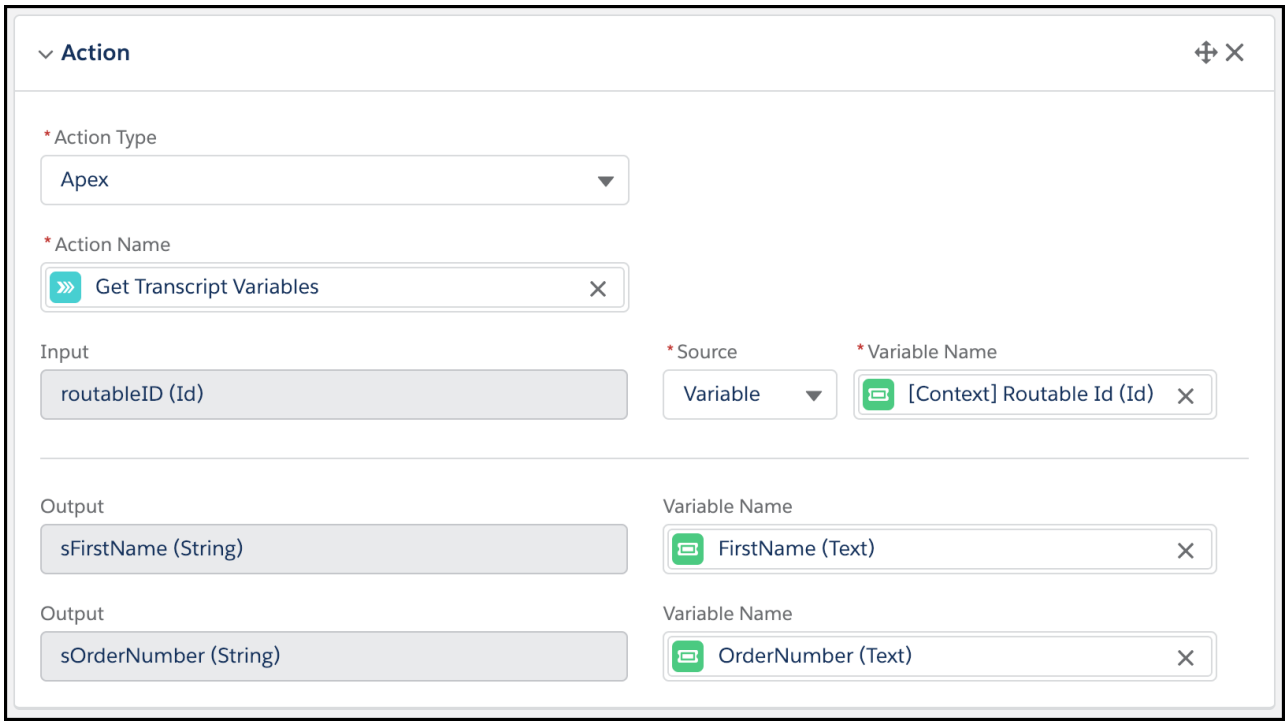
This time, you need to update your Apex code so that it grabs the order number from the `LiveChatTranscript` record. Use this new code in place of the code you previously created. You see that it's almost identical, except that it grabs the order number and puts it in a new output variable (`sOrderNumber`).

```
public with sharing class CookbookBot_GetTranscriptVariables {

  public class TranscriptInput {
    @InvocableVariable(required=true)
    public ID routableID;
  }

  public class VisitorNameOutput {
    @InvocableVariable(required=true)
    public String sFirstName;

    @InvocableVariable(required=true)
    public String sOrderNumber;
  }

  @InvocableMethod(label='Get Transcript Variables')
  public static List<VisitorNameOutput> getUserName(List<TranscriptInput> transcripts)
 {

    List<VisitorNameOutput> names = new List<VisitorNameOutput>();

    for (TranscriptInput transcript : transcripts) {

      // Query for the transcript record based on the ID
     LiveChatTranscript transcriptRecord = [SELECT Name, FirstName__c, OrderNumber__c

                                            FROM LiveChatTranscript
                                            WHERE Id = :transcript.routableID
                                            LIMIT 1];

      // Store the first name in an output variable
      VisitorNameOutput nameData = new VisitorNameOutput();
      nameData.sFirstName = transcriptRecord.FirstName__c;

      // Store the order number in an output variable
      nameData.sOrderNumber = transcriptRecord.OrderNumber__c;

      // Add the name to the list of outputs
      names.add(nameData);
    }

    return names;
  }
}
```

To pass data from our Apex code to the bot, we'll call the Apex action to set the `OrderNumber` variable along with the `FirstName` variable. We *could* call this action first thing in our "Welcome" dialog directly, similar to how we were able to set

the `FirstName` variable. However, in this exercise, let's set up an initialization dialog. This way, we can wrap all the initialization actions in this dialog while keeping our welcome dialog a little easier to read.

Create a "Bot Initialization" dialog that calls our Apex action.



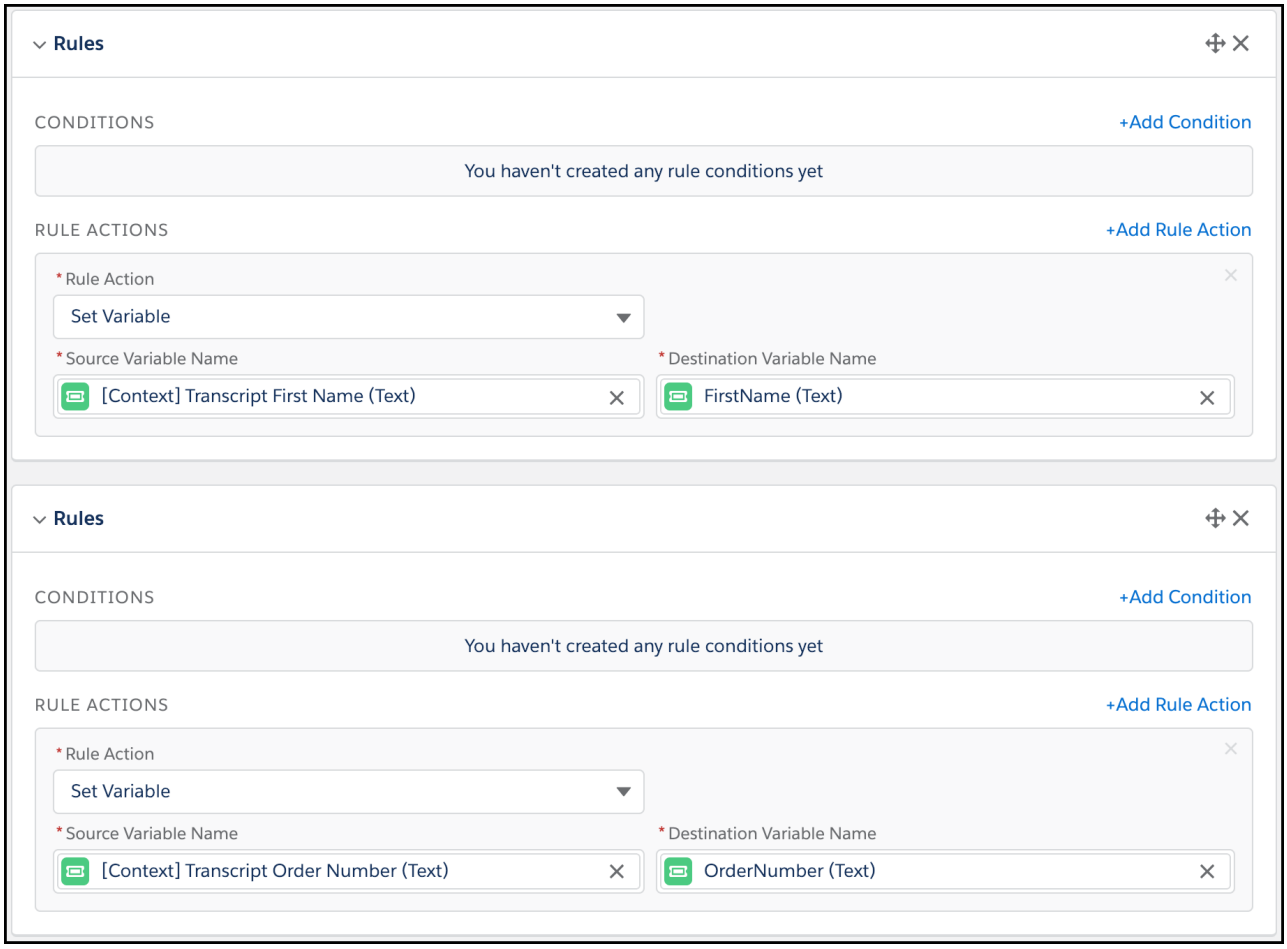

**b.** (Option 2 of 2: **Metadata API Solution**) If you want to extract the `OrderNumber` field from the `LiveChatTranscript` object using the Salesforce Metadata API, review the Metadata Solution in the Greet the Customer section on page 13 and then follow these additional instructions.

This time, add the following XML snippet (in addition to the previously specified XML on page 13 which added the first name) when deploying changes to your org.

```
<contextVariables>
  <contextVariableMappings>
    <SObjectType>LiveChatTranscript</SObjectType>
    <fieldName>LiveChatTranscript.OrderNumber__c</fieldName>
    <messageType>WebChat</messageType>
  </contextVariableMappings>
  <dataType>Text</dataType>
  <developerName>TranscriptOrderNumber</developerName>
  <label>Transcript Order Number</label>
</contextVariables>
```

To use this new context variable in our bot, we call another **Set Variable** rule using the new `Transcript Order Number` context variable (in addition to the previous rule for the `Transcript First Name` context variable). We *could* call this rule first thing in our "Welcome" dialog directly, similar to how we were able to set the `FirstName` variable. However, in this exercise, let's set up an initialization dialog. This way, we can wrap all the initialization actions in this dialog while keeping our welcome dialog a little easier to read.

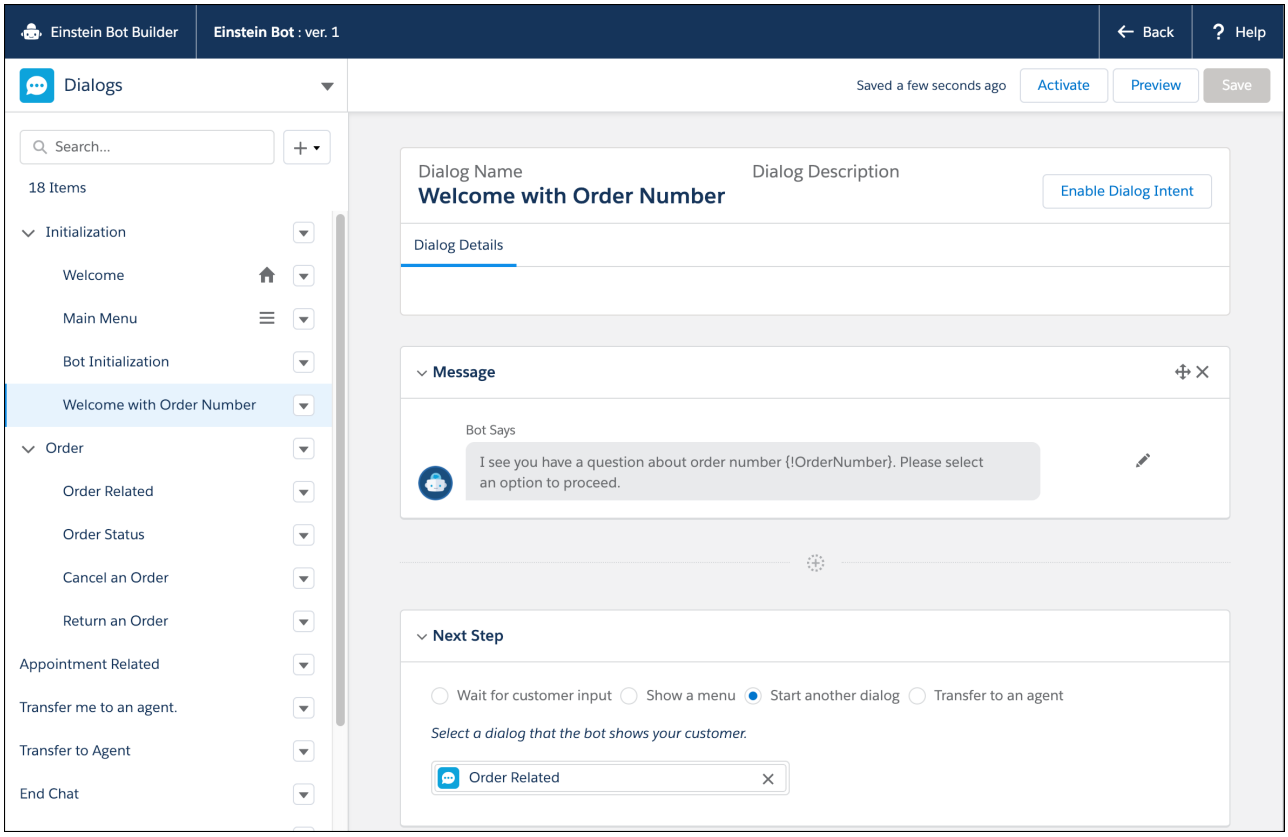
Create a "Bot Initialization" dialog that calls both of our **Set Variable** rules.




4. Optimize the design with rules.

   Let's pause for a minute. So we added a non-required field on pre-chat form. We want this to be flexible so customers don't have to enter the order number if they are interested in other questions. But for those who do provide an order number, I think we can safely assume they have an order-related issue. So wouldn't it be better if we just send them directly to the "Order Related" sub menu? Yep, that's where the rules come in to help us guide users through different steps and optimize their chat experience.

   First, we need to build a new "Welcome with Order Number" dialog to greet the customer in a different way if they provide the order number in the pre-chat form. We use the message text: `I see you have a question about order number {!OrderNumber}. Please select an option to proceed.`




Note the next step from this dialog should be set to the "Order Related" dialog to present the options from the "Order Related" submenu.

Tip: You can group your dialogs in Dialog Groups to make them easier to manage. In this example, we've grouped all the initialization dialogs under one group. To create a group, use the same create button used when creating dialogs.

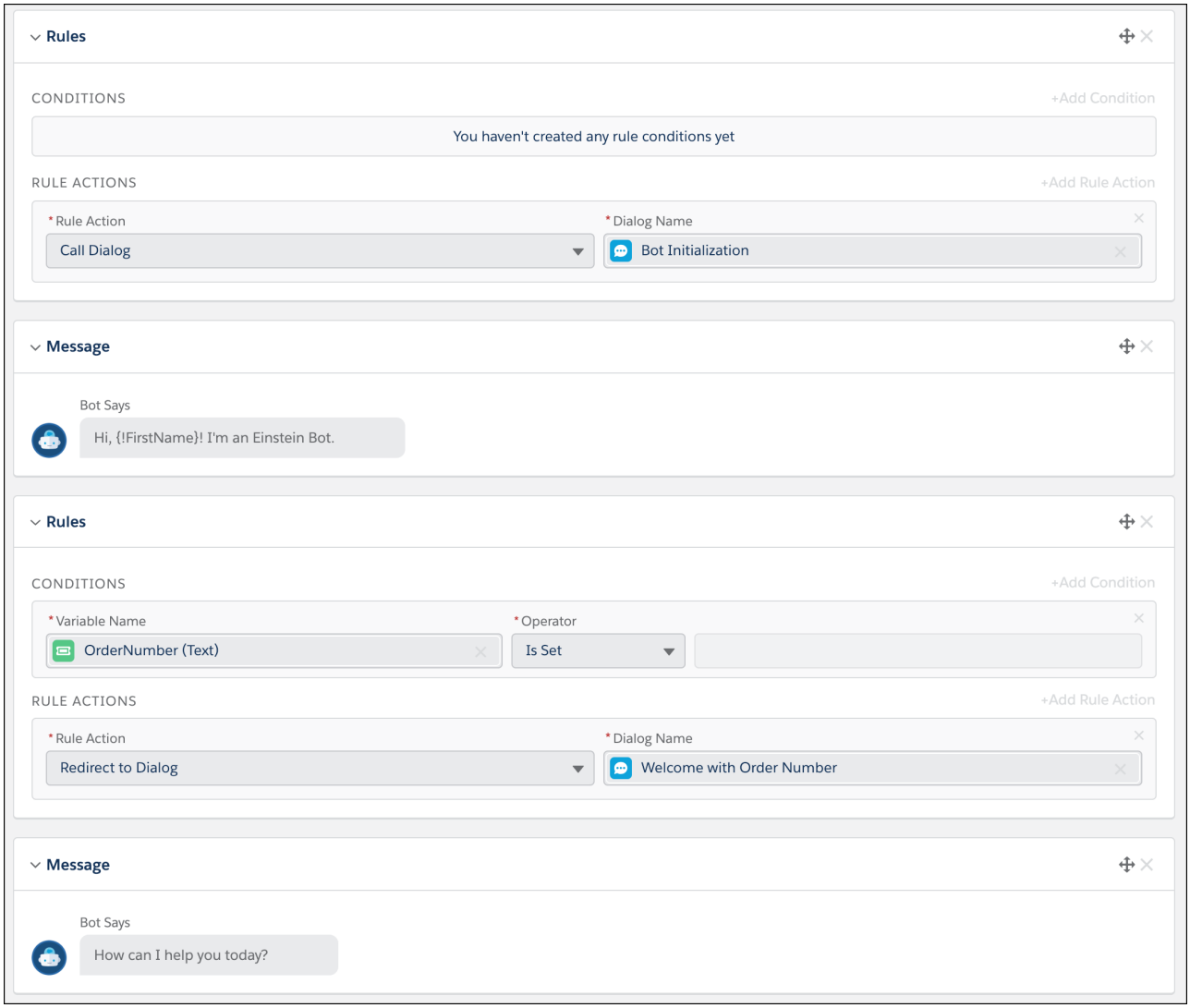
In the original "Welcome" dialog, let's perform the following steps:

**a.** Use a Rule Action to call the "Bot Initialization" dialog we just created to grab the order number from the pre-chat field.

**b.** Create a message to say hi to the customer, using the `FirstName` variable.

**c.** Use a Rule Action that redirects to the "Welcome with Order Number" dialog only if the `OrderNumber` variable has been set.

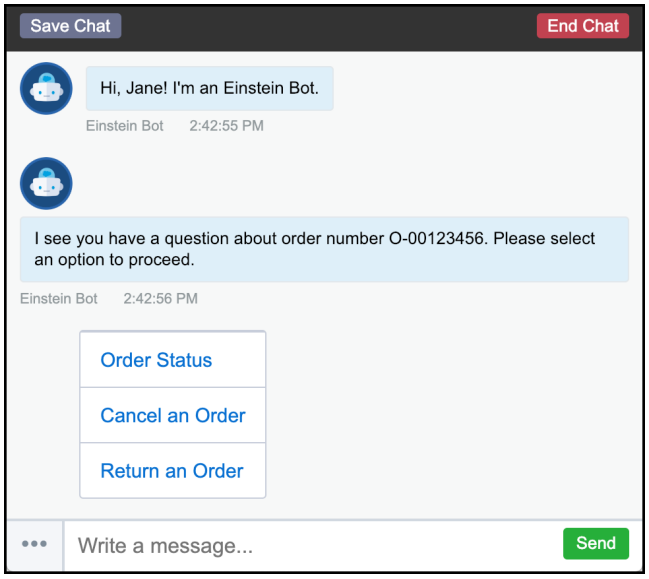
**d.** If the variable has not been set, then just ask the customer what they need help with.



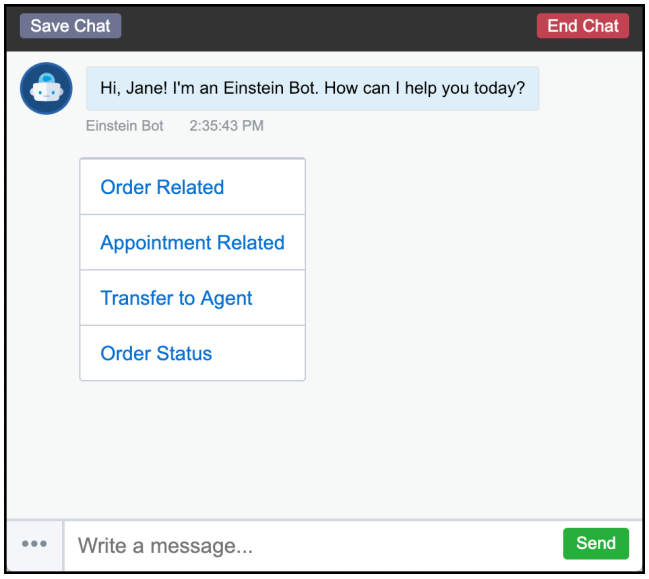

Notice that we split the original greeting message into two parts. We first greet the customer, then we have the rule action before the next message. If the customer doesn't provide an order number in the pre-chat form, we continue with the original question "How can I help you today?" Otherwise, the customer is sent to the "Welcome with Order Number" dialog for order-specific questions.

5. Test!

   Now let's test our optimized user flow. If the customer enters an order number in the pre-chat form, the bot can use that information to get right to the order-specific flow.




Otherwise, the bot falls back to the standard welcome flow.




### Best Practices and Additional Notes

- **Call Dialog vs. Redirect to Dialog**. In the initial "Welcome" screen, we used two different dialog-related rule actions. So why does Bot Builder provide two rule actions that seem very similar to each other by their names? Well, the **Redirect to Dialog** action indicates the destination dialog will take over the flow at this point; while with **Call Dialog** action, the destination dialog will return the flow back to the original dialog. We are using the **Call Dialog** action with the "Bot Initialization" dialog, because we need to send the user back to welcome screen to continue with the branching logic. Also note that whatever configuration we select in the **Next Dialog** section for "Bot Initialization" dialog is ignored because it's only used in a **Call Dialog** action and flow needs to return back to the calling dialog.
- **Bot Variable Lifecycle**. Did you notice that when you click Order Status, the bot didn't ask for the order number if you already provided the information on pre-chat form? However, if you look at the "Order Status" dialog from Bot Builder, we didn't make it an optional step. So, what happened? One thing about the **Question** action—if the **Answer** variable has any value, it will skip the question. It's a smart feature. For example, let's say the customer provides an order number to check status then later decides to cancel that order. We don't have to ask for the order number again. On the other hand, if you do need to ask the order number again,

let's say you want to build a process to allow the customer to check the status for another order, you can use the **Clear Variable** rule action to reset the **OrderNumber** variable first. This way, when the "Order Status" dialog is triggered again, the bot will ask the question again.

- **Variable Validation**. Have you tried to use an invalid order number on the pre-chat form (one that doesn't match the regular expression for entity **OrderNumberString**)? What happened? Think about what can we do to validate that the customer input is indeed an order number before skipping the order number question in the bot.

# Understand Customer Intent with Natural Language Processing

In this recipe, we learn how to set up Natural Language Processing (NLP) so we can make the bot smarter, and more conversational.

**Channels**

- Chat
- Messaging

**Prerequisites**

- Set Up Your First Einstein Bot on page 1
- Greet the Customer on page 6
- Prompt Customer with Menu Options on page 29

**In this recipe you learn how to:**

- Create an intent set with training data.
- Associate intents to dialogs.
- Train the bot with a dataset.
- Use entity recognition to recognize information in customer responses.

A good approach in bot solution design is to start with inquiry volume analysis. Understanding your typical customer inquiry types and their volume gives us a good picture for the things we want to tackle first. Once you have the initial set of target inquiries defined, you can start designing and building the dialogs. In general, each type of inquiry should be mapped to one dialog or a group of dialogs. These dialogs are the building blocks of our bot. After we have these dialogs in place, all our bot needs to learn next is how to understand the customer intent and kick off that particular conversation.

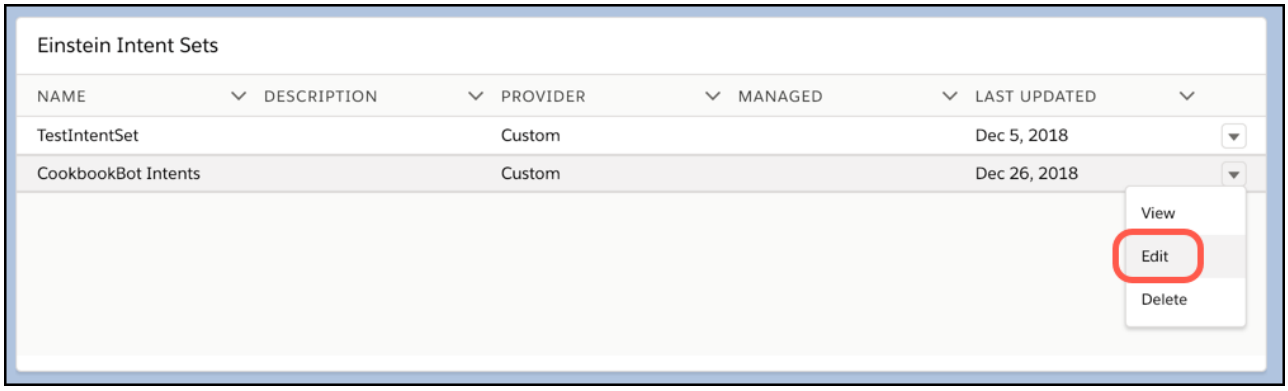
An interface with menus and buttons provides customers a basic-yet-effective navigation experience, especially during the early phase of your bot development when the support for different inquiry types is limited. However, as our bot develops with more inquiry types and dialogs, the navigation menus and submenus become longer and deeper, and less optimal. For a bot to support many types of transactions or to make a bot more conversational, Natural Language Processing (NLP) provides a more natural approach to identifying customer intent. Customers enter whatever they want to ask and the bot can ideally understand their intent and start the right dialog.

Although it takes iterations of training, learning, and refining to improve the accuracy of intent detection, the initial setup is easy. We can get our bot up and running with NLP in no time.

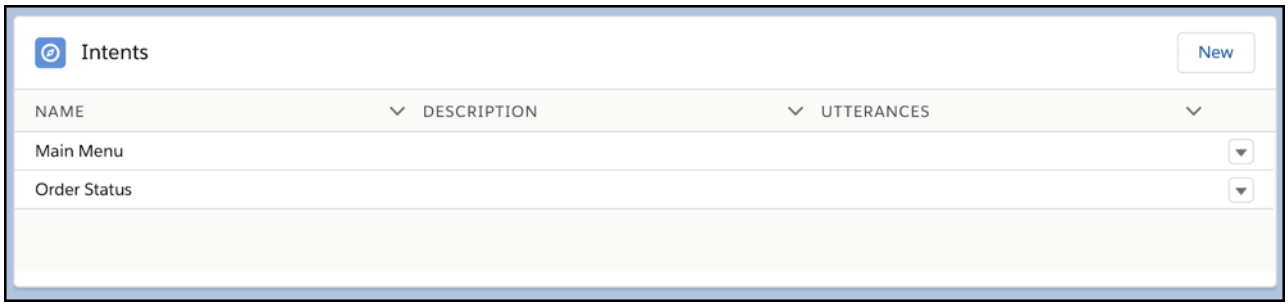
1. Create an Einstein intent set.

   The customer intent training dataset contains the many ways your customer may ask the bot about one particular thing, also known as an "intent". **Intent Sets** is where we can enter the training data—customer input or utterances.

   From Setup, use the Quick Find box to find **Einstein Intent Sets** and create a new intent set called `CookbookBot Intents`. After you've created the intent, edit this intent set.

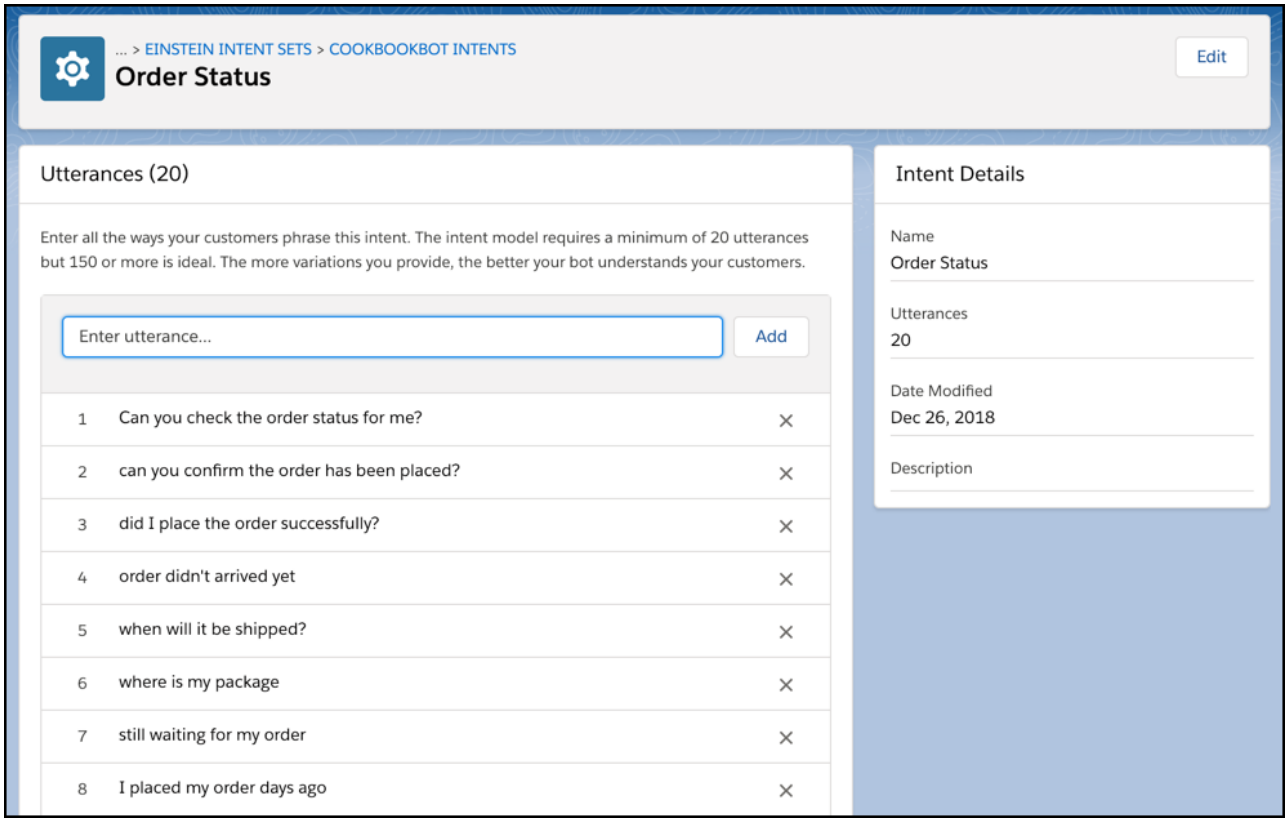



From the **Intents** section, create at least two intents: `Main Menu` and `Order Status`.




For each of the intents, enter at least 20 unique customer inputs. For example, for `Order Status`, you might enter:

- `Can you check the order status for me?`
- `did I place the order successfully?`
- `order didn't arrive yet`
- `when will it be shipped?`

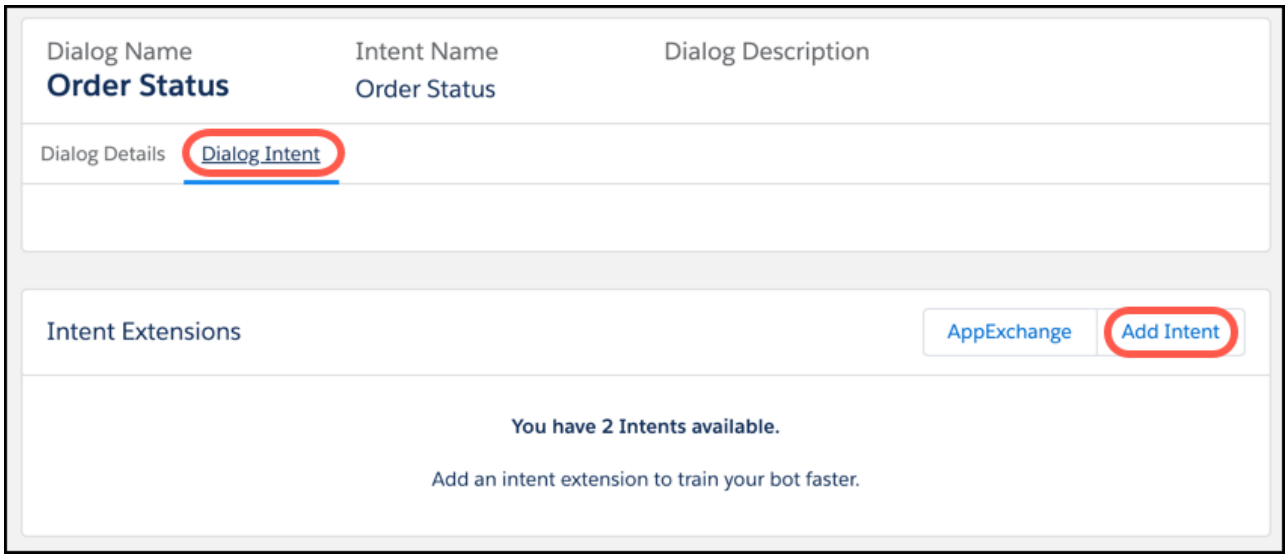



2. Associate intents to dialogs.

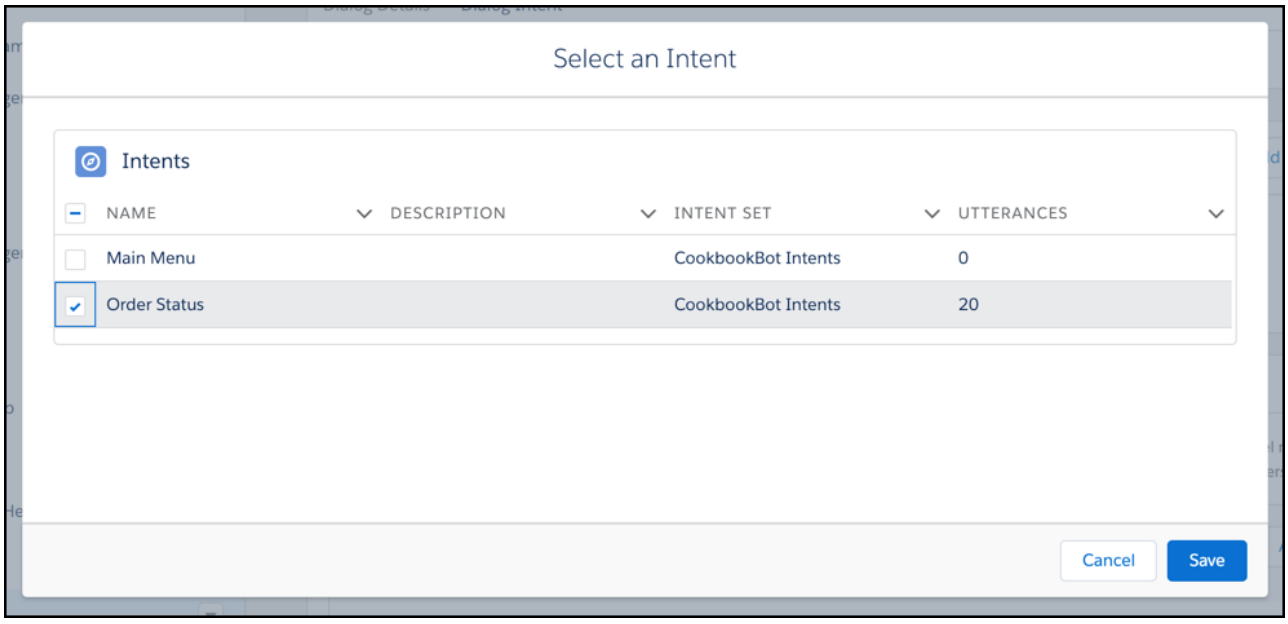
   The result of successfully identifying a customer intent is to trigger a dialog that's designed to address this inquiry. Therefore the intents we just created need to be associated with a dialog.

   Open the "Order Status" dialog from Bot Builder and select the **Dialog Intent** subtab in the dialog header.

   Important: If you have not yet enabled dialog intents for this bot, you'll have to click the **Enable Dialog Intent** button beside the dialog name before you can access the **Dialog Intent** subtab.

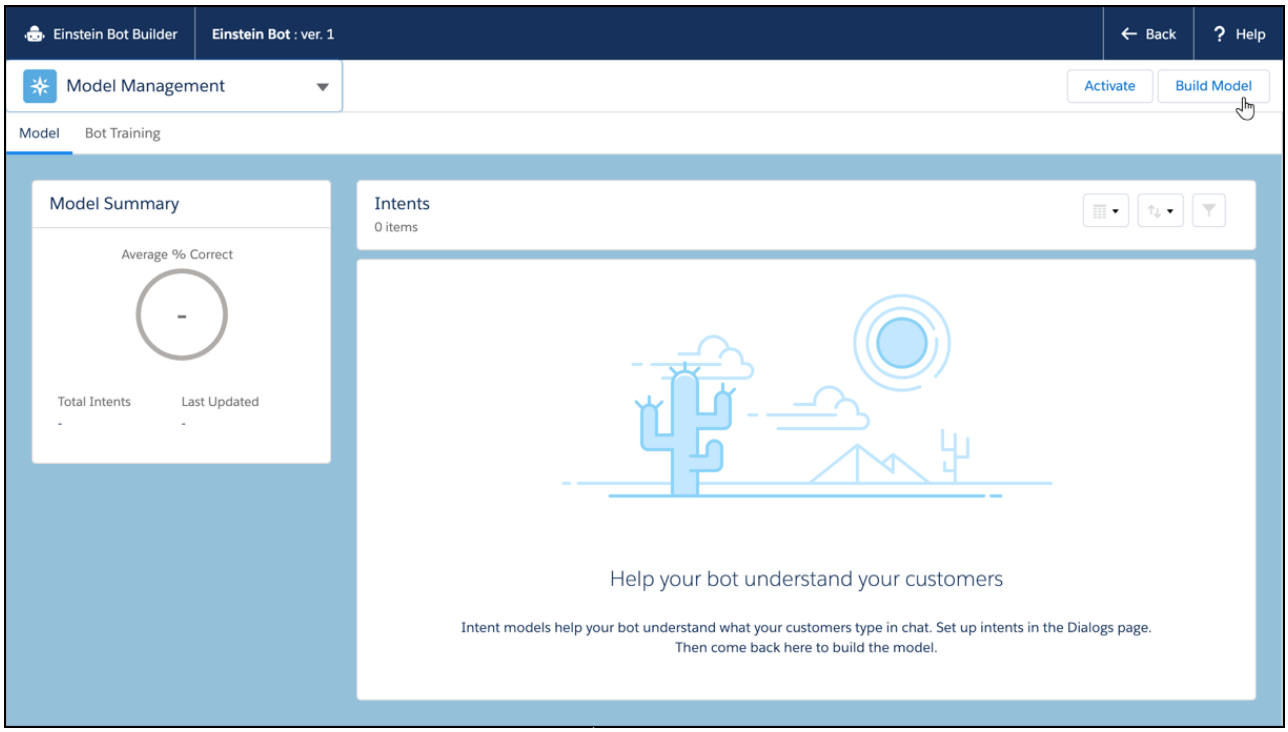


In the **Intent Extensions** section, select the **Add Intent** button and select the "Order Status" intent we created from last step. (Notice there's also an **AppExchange** button that allows you to search pre-built intent libraries and install them into your org.)




3. Train the bot.

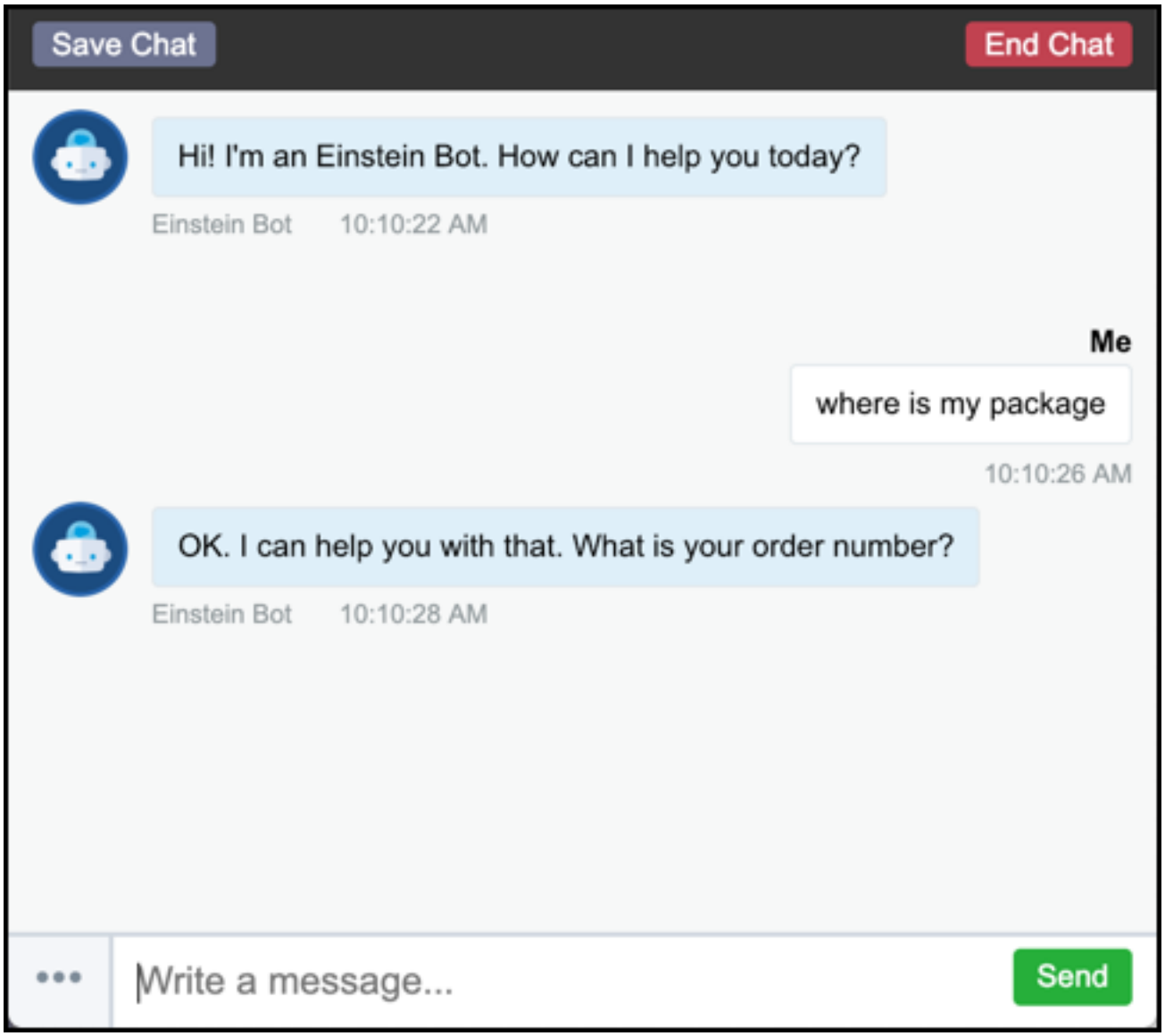
   The actual training itself is pretty uneventful. Go to the **Model Management** tab and click the **Build Model** button and that's about it.




   Once started, the model building process usually finishes in a few minutes given that our training dataset is small at this point.

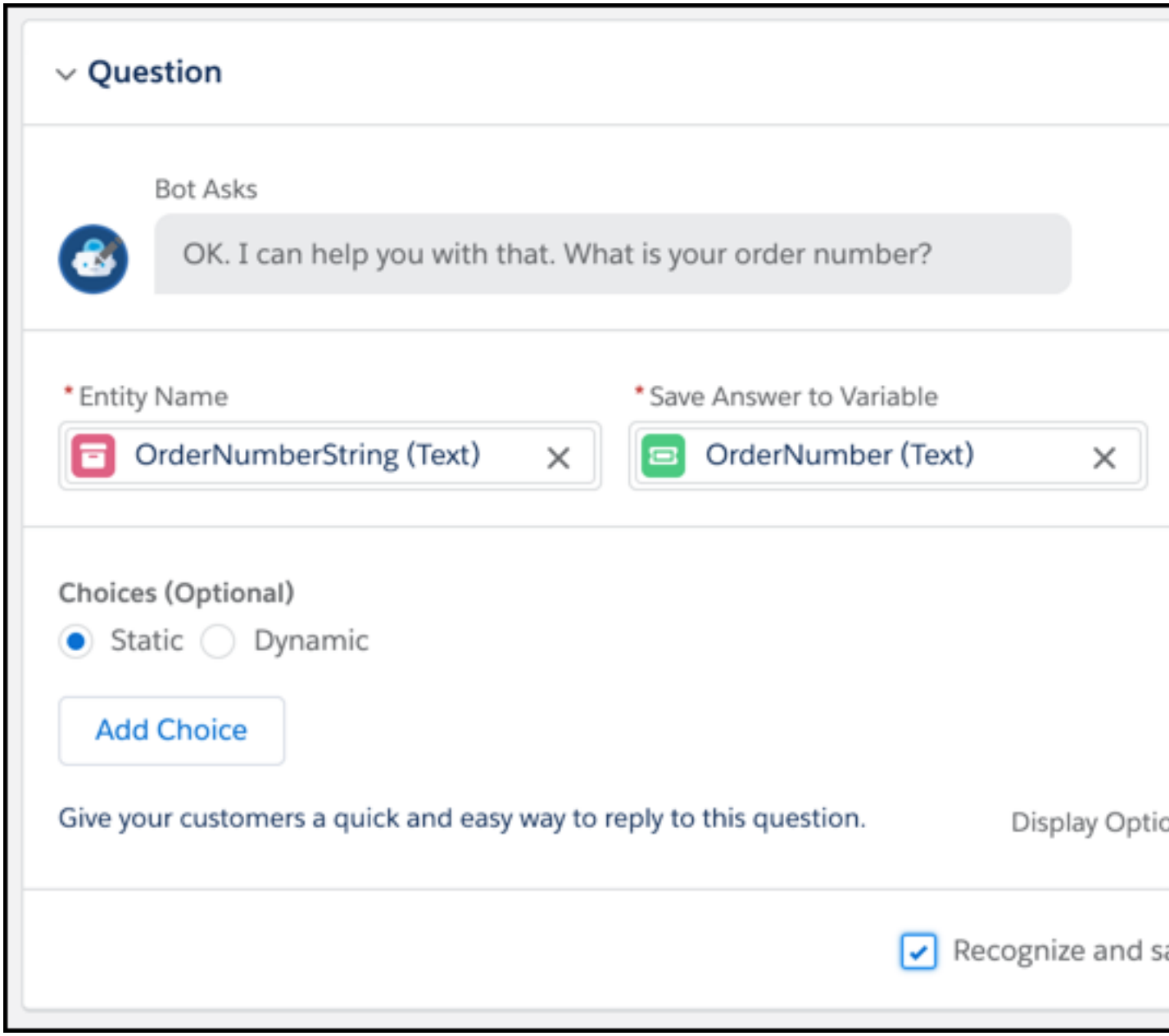
4. Test!

After the Einstein **Intent Management** status shows it has succeeded, we can launch the bot again. Since we have customer intents for the "Order Status" dialog, let's try that intent. After receiving the greeting message, instead of clicking one of the menu options, let's enter something like "where is my package". In our example, the bot correctly identified the intent and sends us to the "Order Status" dialog. The bot now asks for our order number right away!

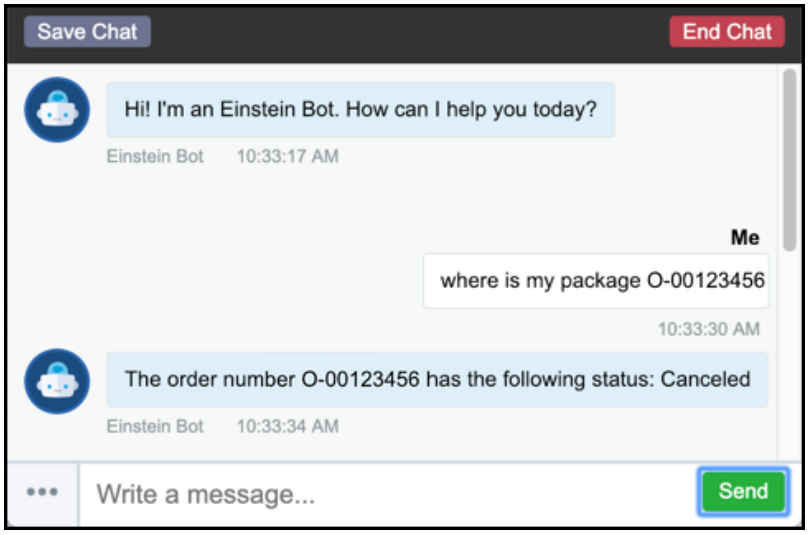



5. Test again with entity recognition.

   We are not done with NLP yet. Let's do something even more advanced. We'll test it one more time but before that we need to quickly verify one thing. Go back to the "Order Status" dialog and in the question where you ask for the order number, make sure that the check box **Recognize and save the answer from customer input** is checked. Note that the checkbox is at the bottom of the question element.




This recognition feature is probably one of the most often overlooked features that we should keep in mind when designing a conversational bot. To learn why, test your bot again. This time instead of asking "where is my package", ask "where is my package O-00123456". (Use whatever value is an actual order number in your org.)




The bot not only identified the intent of checking order status, but also correctly retrieved the subject (Order Number) of the question. Thus it skipped the order number question and jumped directly into the action to check order status.

So how did it do it? Once the bot identified the intent was an order status inquiry, it looked at the questions in this dialog with the **Recognize** checkbox enabled to see if there is any answer provided already in the original customer input. With the **OrderNumberString** entity specified in a regular expression (as described in Gather Customer Information Using the Bot), the bot easily found the order number.

A few additional considerations:

- There are two places where you can add customer inputs or utterances. You can do so via the **Intent Sets** that we have shown in this recipe or you can enter customer input directly into the dialog intent subtab. We recommend using the **Intent Sets** option for a number of reasons.
  - This design makes testing (or continuous improvement) much easier. With intent sets, you can separate the conversation design in Bot Builder and training set development in intent sets.
  - Intent sets can be mapped to fields in the `MlIntentUtterance` object where utterances are stored. This means we can use tools such as Data Loader to mass-upload intents to intent sets.
  - You may have to create multiple bots for different personas or for different use cases. You don't have to recreate all the utterances in every bot. Use one intent set and share it across multiple bots.
- Not all dialogs need customer inputs. For example, our "Welcome" dialog shouldn't have to be trained. You wouldn't expect the customer to be greeted again, right? Or if you have a group of dialogs to process one inquiry type, perhaps only the first dialog in the group should be trained.
- Although the minimum size of the customer inputs for each dialog is 20 unique utterances, we recommend at least 150 customer inputs for each dialog that requires training.
- If you're using a bot for messaging and you created a "Messaging Initialization" dialog, as described in Greet the Customer with Messaging on page 23, follow the instructions in that recipe to call your initialization logic from all dialogs that use intent detection.

# Create a Self-Service Bot with Knowledge, Flows, and Cases

In this recipe, we use multiple Einstein Bot features to solve a common use case: what happens when your agents aren't available but your customers have a question? In this business case, we want the bot to ask the customer about the issue, search the Knowledge base to deliver a self-serve solution, and then if the customer needs further assistance, create a case in Salesforce for your agents.

**Channels**

- Chat
- Messaging

**Prerequisites**

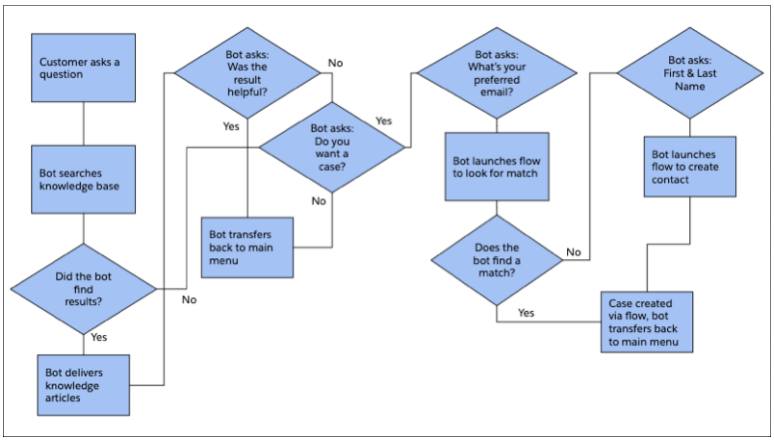
- Set Up Your First Einstein Bot on page 1
- An org with Lightning Knowledge set up
- A bot assigned to a Custom Bot User which has access to Cases & Contacts

**In this recipe you learn how to:**

- Set up Object Search to quickly find articles related to a customer question.
- Use Conditional Messaging to provide targeted responses to the customer.
- Create a case using Flows to complete the bot conversation.

Because this recipe is channel agnostic so that the bot works on any channel, we skip using the pre-chat form. Instead we use flows to look up the user via email. If your bot uses Messaging, it doesn't have access to the pre-chat form, so you must bake the contact creation into the bot dialogs. If your bot is only used for Chat, you can use the pre-chat form instead of the method that we show here. We highlight great conversation design methods when we ask questions to keep the customer involved in the process every step of the way.

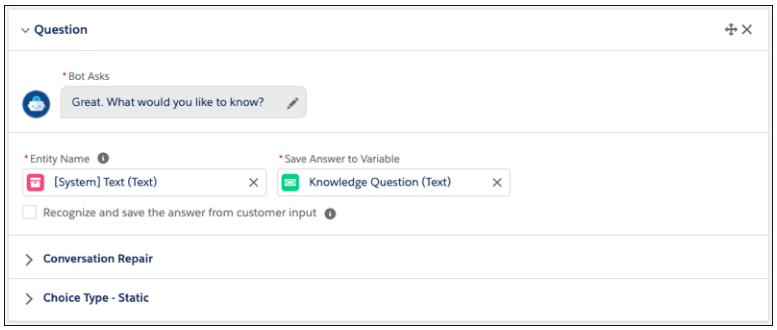
To give a high-level idea of the bot is doing, here's a flowchart:




1. We need a few dialogs for this recipe, so we create a dialog group called Knowledge Search to keep things organized. In this dialog group we have four dialogs:
   - Search Knowledge Base: the bot asks the customer what they're looking for and performs a knowledge search
   - Create a Case Question: the bot uses conditional messaging to ask the customer if the knowledge article solved their issue
   - Case Flow: a dialog where the bot attempts to find a contact and creates a case
   - More Case Info: a dialog where the bot asks for more information to create a contact, then creates a case

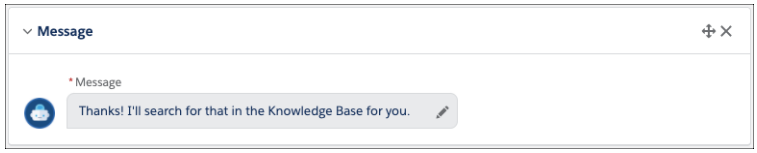
   To create a dialog group in the Bot Builder, click the plus icon next to the search bar and select **New Group**. Name the group `Knowledge Search`. Use the plus icon again and select **New Dialog** to create the four dialogs listed above. Make sure that you select the Knowledge Search group in the **Assign to Dialog Group** field.

2. To make sure that we wire this dialog to the main bot, add the **Search Knowledge Base** dialog onto the Main Menu dialog.
3. Click the **Search Knowledge Base** dialog to build the introductory messages. Click the plus icon and start with a Question Dialog Step. In the Bot Asks field, enter `"Great. What would you like to know?"` as the message.

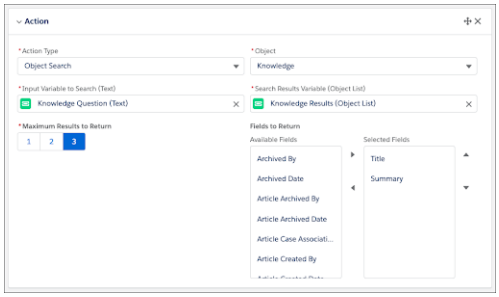



We must store the customer input in a variable, so select **[System] Text (Text)** as the Entity Name to capture the text data. In the Save Answer to Variable field, click to create the Text Variable **Knowledge Question**.

It's a good conversational design practice to let the customer know the bot's next step. To add a Message Dialog Step, click the plus icon and enter `Thanks! I'll search for that in the Knowledge Base for you`.




4. Next, let's add an action to use Object Search. Add an action dialog step and select Object Search as the Action Type. Select **Knowledge** in the Object field. In the Input Variable to Search, select your Knowledge Question variable, and create a List Variable `Knowledge Results` to serve as the Search Results variable.

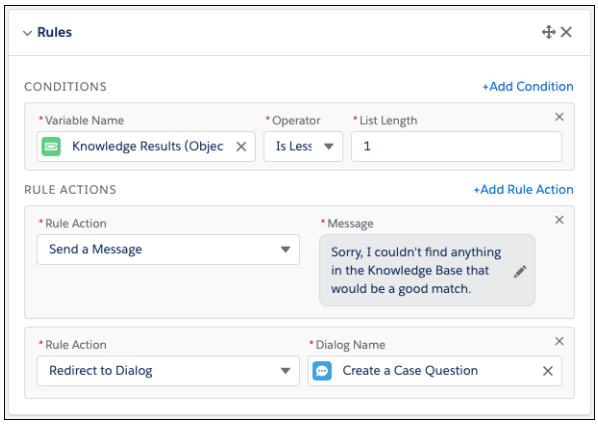



Knowledge search let you deliver up to three results, so feel free to select up to three articles. Make sure that your input variable is set to **Knowledge Question**, and create an Object List variable `Knowledge Results`.

Note: To create an Object List Variable, select **Object** as the Data Type and make sure that you check **Contains a List**.

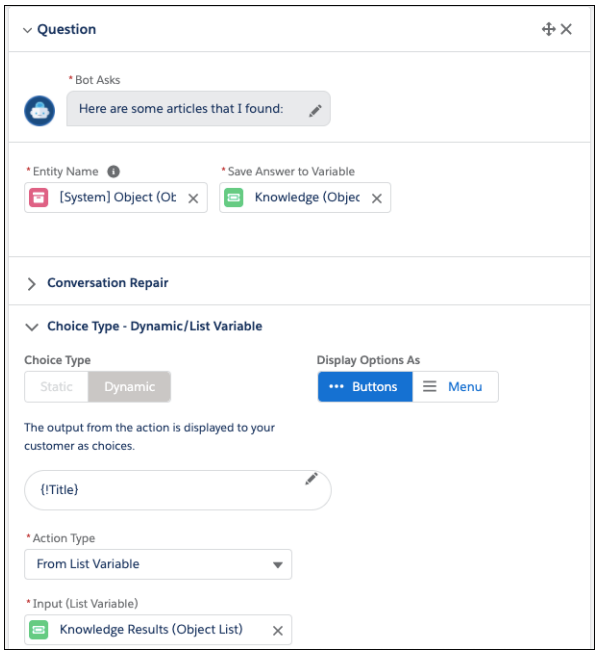
5. Next we plan for two scenarios: the bot doesn't find anything and the bot finds results. To start with the bot not finding anything, add a rule element with conditions to check if the Knowledge Results variable is less than 1. To keep the customer informed, add a Message dialog before you redirect them to ask whether they want to create a case anyway with **Create a Case Question** dialog.

We select **Sorry, I couldn't find anything in the Knowledge Base that would be a good match.**, but you can use whatever works for your company.

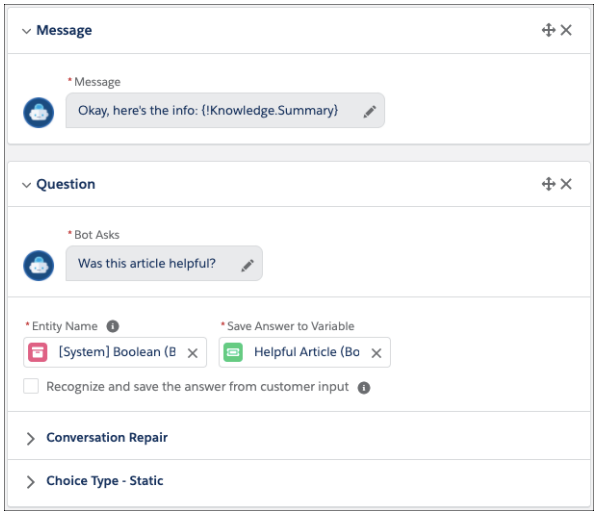



6. We handled the null use case, so let's continue to the results. Add a question element, and enter the text `Here are some articles that I found:` in the message. Click **Choice Type** to deliver the results to the end user. Change the Choice Type to **Dynamic**, and then set Action Type to **From List Variable**. The input is the object list variable used in the knowledge search completed in the Action dialog — in this example, Knowledge Results.

   This is a dynamic choice, so we identify the field that is placed within the choice buttons. For knowledge, we recommend the **{!Title}** field. To make sure that the selection is saved into a new variable, create an Object variable and name it **Knowledge**. Set the Entity to **Object**. This setting delivers multiple fields within the selected article. Because the Error Rule is required, let's add a redirect to **Create a Case Question**.

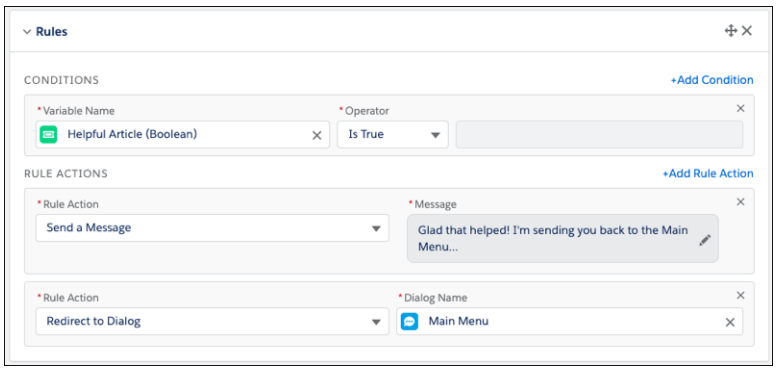


7. Now let's deliver the selected article to the customer. Add a Message dialog step to deliver the selected article summary, in this case, `{!Knowledge.Summary}`. Since the Knowledge variable we created is an Object variable, we can call any field inside the object by following the format: `{!VariableName.FieldName}`.

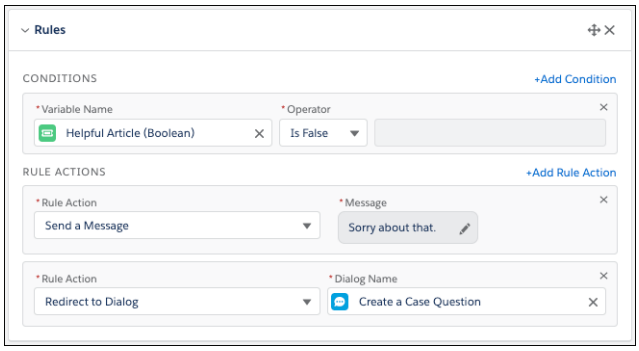



We're now going to check in with the customer to see if that article was helpful. Add a question dialog step with a new boolean variable to ask if the article was helpful. We use Conditional Messaging built into the Rules dialog step to make sure that the customer found what they wanted.

If it was helpful, then our bot's job is done here and we return the customer to the Main Menu. It's good practice to let the customer know where you plan to transfer them before you do it, so add a conditional message as part of the Rule dialog step before you redirect.

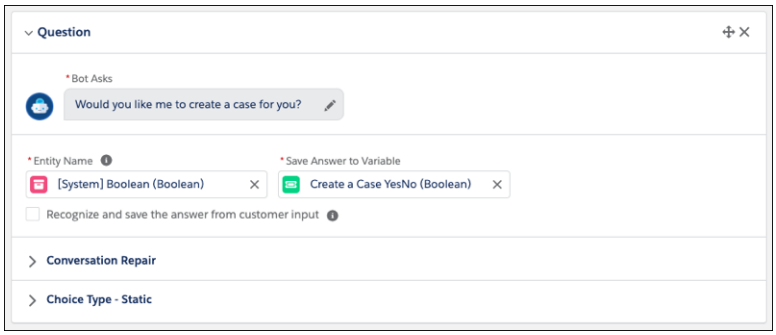


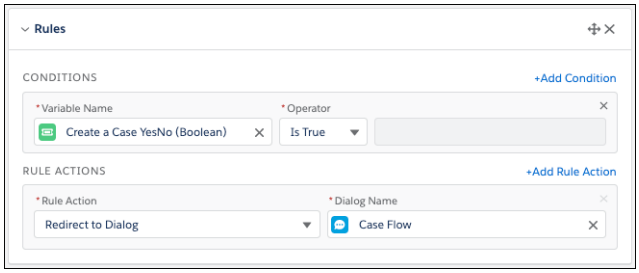

If the article wasn't helpful, we want to give the customer the option to create a case, so we add a transfer to the Create a Case dialog.

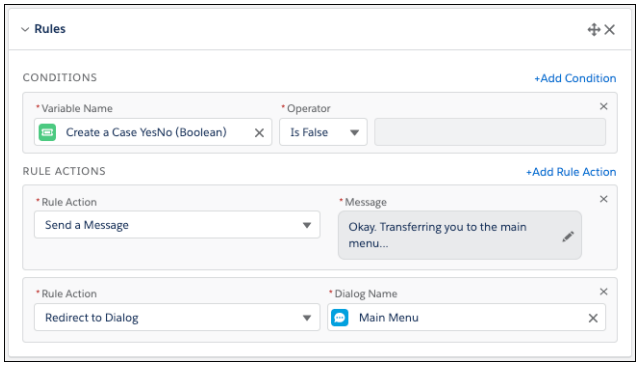



8. Next we build a flow to create a case for when a knowledge base article isn't helpful to our customer.

   Switch over to the Create a Case Question dialog. We use a similar flow with a question dialog step that leads to two rule dialog steps. And we use conditional messages that depend on the answer. Create a boolean variable to store the answer. If true, the bot redirects to the Case Flow dialog, and if not, then the bot moves back to the Main Menu dialog.

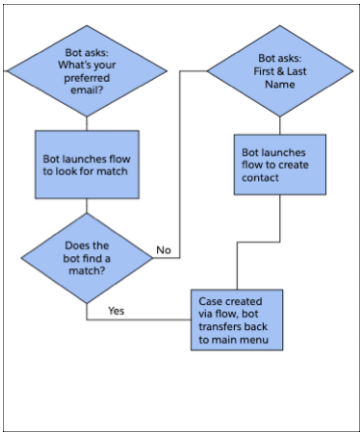


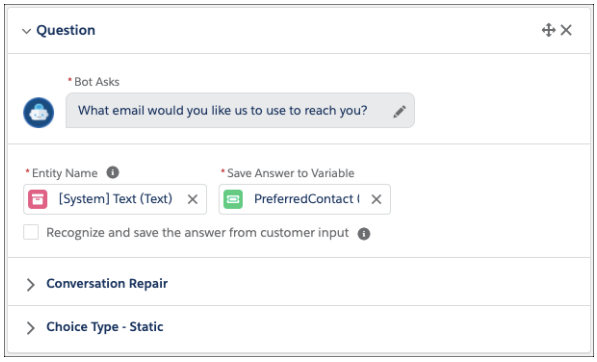



9. Now let's add flows to our bot. To keep things simple for this flow, we ask the user for an email address. To enhance this flow, you can ask the customer for either an email or a phone number. Here, we search for a contact based on their contact information. If the flow is unable to find a match, the flow is complete. If the bot finds a match, then the flow should create a case attached to that contact, and deliver the case number back to the bot.

   On the flowchart, the next section refers to the second half of the chart:




   Before we move to the Flow Builder, let's open the Case Flow dialog and enter a new Question dialog step to ask the customer for their preferred contact information:

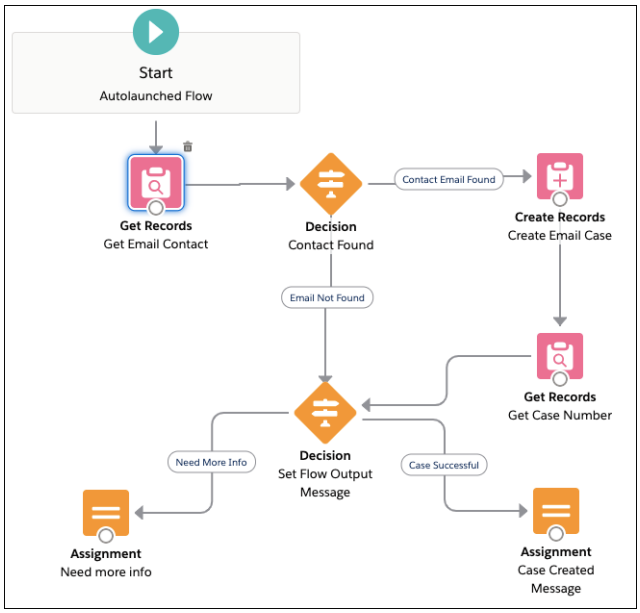



   Click **Save** on the Bot Builder menu.

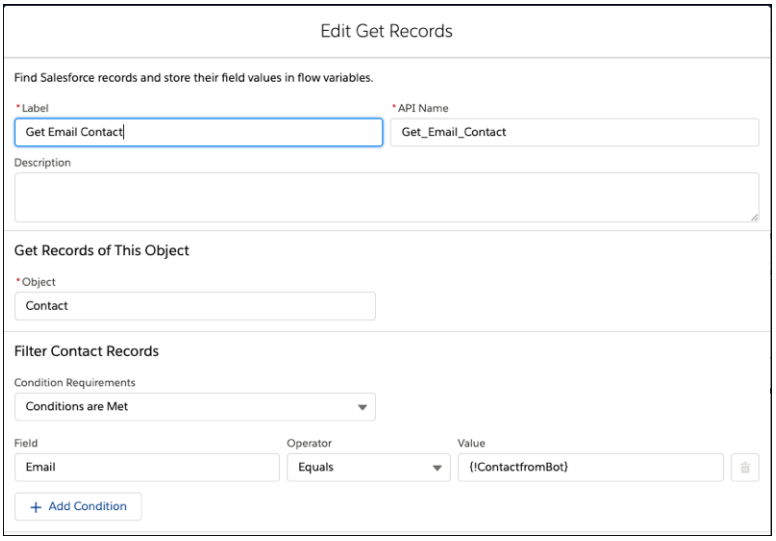
   When you work with bots and flows, it's important to identify the connections between the two—the input and output variables that the bot uses to pass and receive information from the flow. The variable settings in Flows, called Availability Outside the Flow, define whether they show up in the Bot Builder as inputs or outputs.

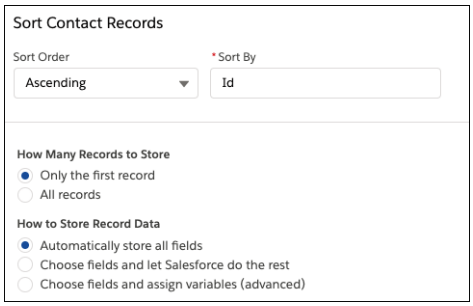
   For the inputs, the wiring between Flows and Bots includes two new variables in Flows to capture the preferred contact information and the original question that the end user asked the knowledge base. The variables link up in the builder to match Preferred Contact and Knowledge Question, respectively.

**10.** We split the process into two autolaunched flows—one if there's a contact match in the system and another to create a contact and case at once. Because we need more information from the customer, it makes sense to move this process into a separate flow. The dialog can launch one flow at a time, so calling two flows works with the conversation written in the dialog. We create this flow with email only, but you can use this base as a launching point for other channels.

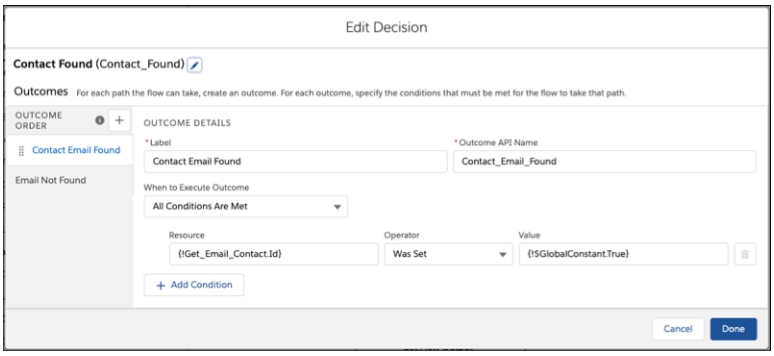



Let's create the lookup part of the flow. We add a Get Records element to the workspace to look up a contact based on the email:

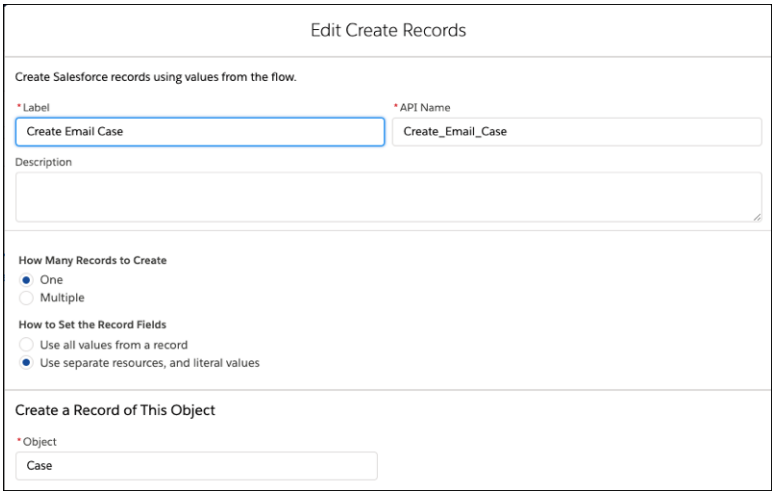


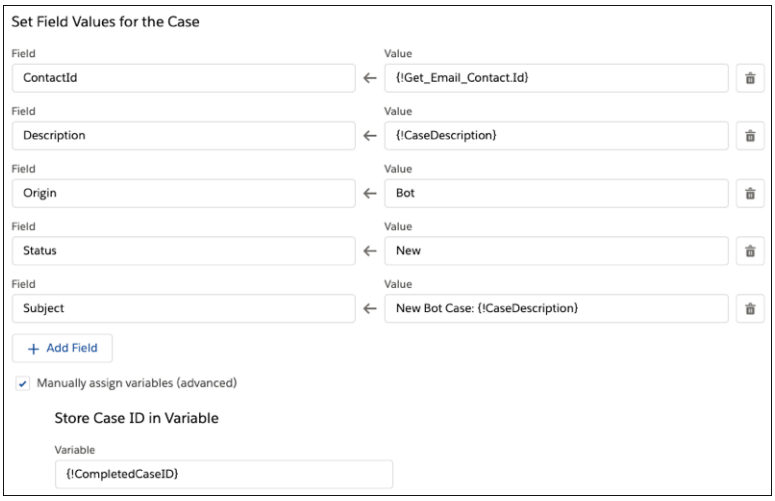



**11.** Next, we add a Decision element to check whether a match was found:



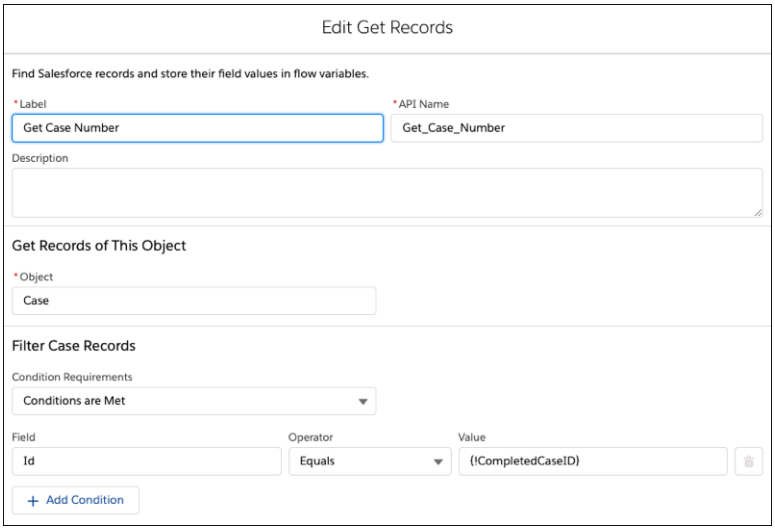

If found, then the flow moves to a Create Records element that defines the case record:

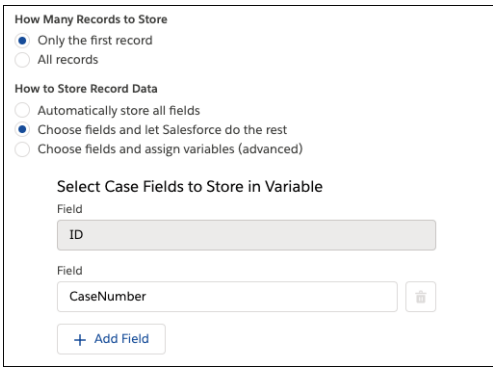




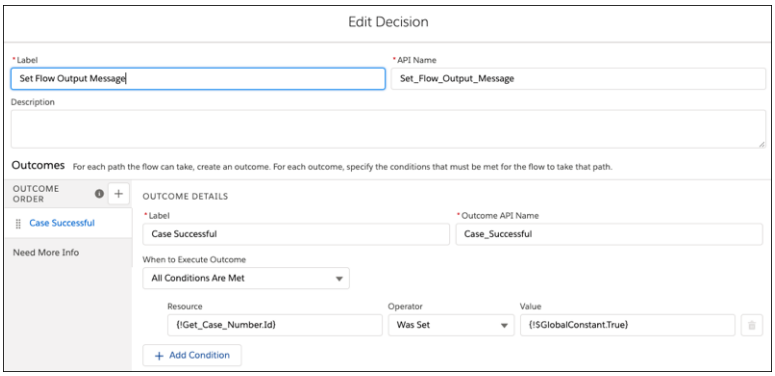

Note: We added `Bot` as a value on the Case Origin picklist to run reports on how many cases the bot created.

**12.** Next, we use Get Records and the Case ID to get the case number to pass along to the customer in the final message:

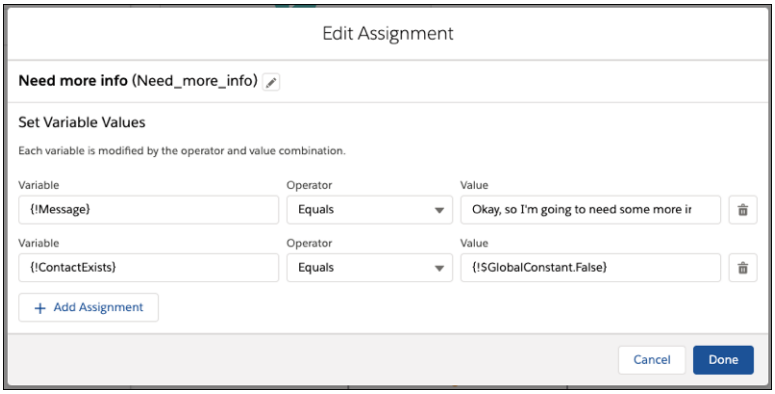


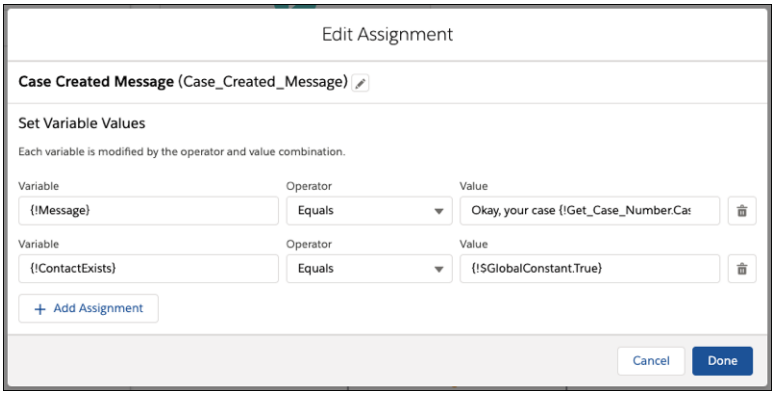



**13.** Now, let's add a Decision element to define which output message is brought back to the bot. Whether it's that a case was successful or that we must create a contact, we can feed both results to the same variable back to the bot.




We finish with two Assignment elements that point to the same variables:

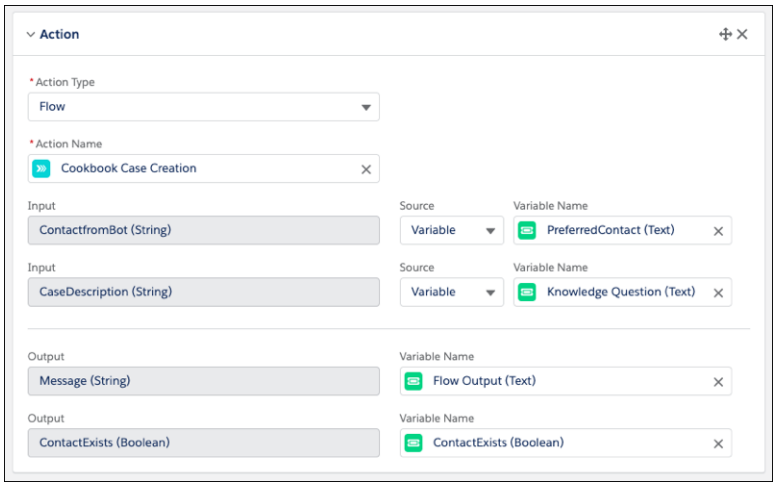


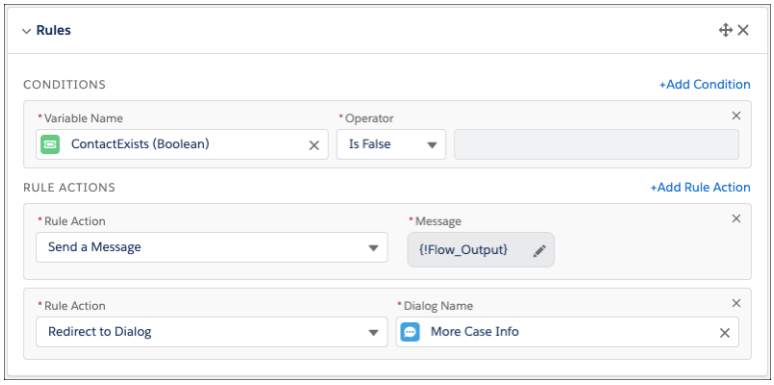

You can use variables such as the case number to personalize the message to the end user, for instance: `Okay, your case {!Get_Case_Number.CaseNumber} has been created.`

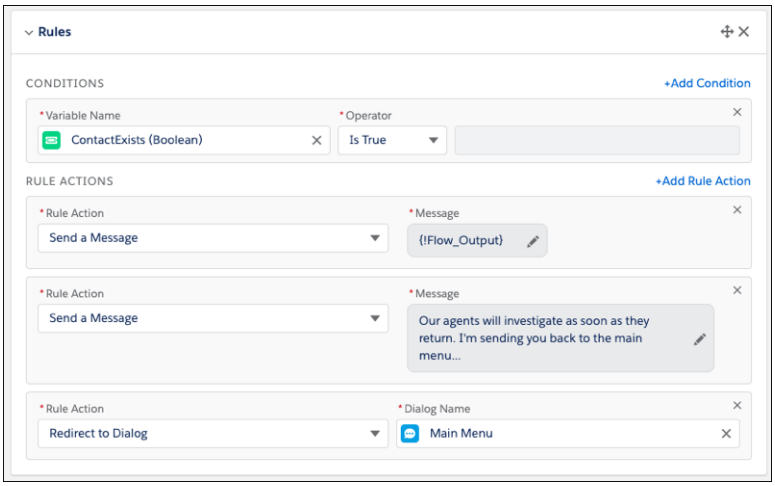
Save the flow and activate it. To validate that the flow is working, use the Debug button to work through all the potential use cases.

**14.** Let's head back to the Bot Builder to connect the bot to the flow. Select a new Action element and enter the flow details. If your flow is missing, check to ensure that it is activated, and if variables are missing, return to the Flow builder and confirm that the variables are set as available for input or output.




**15.** Finally, let's build out the dialog steps to handle the output of the flow. If a contact does not exist in the system we want to move the bot to the second flow, which asks the customer for more information and then creates both records. Otherwise, the flow includes the success message with the case number and then sends them back to the main menu. Add two Rule dialog steps to act on the boolean variable of whether a contact exists in the system: **ContactExists**. This information is passed through the flow back to the bot.

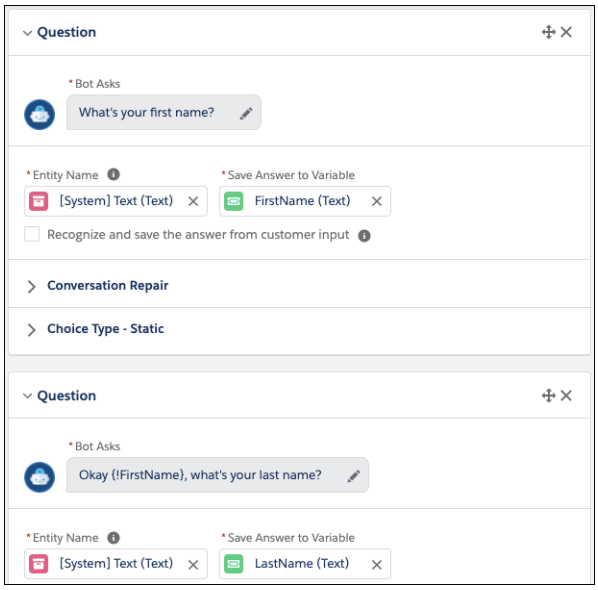


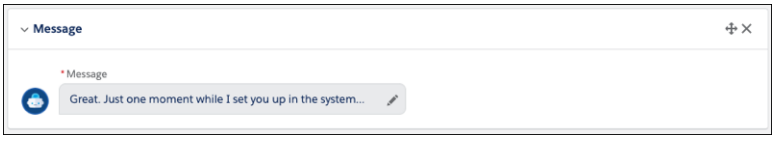



Because we built the output message into the flow, we can use the same **{!Flow_Output}** variable in both situations.

**16.** Let's move on to our final dialog step, which asks the customer for more information and then introduces a flow to create a contact and a case at once.

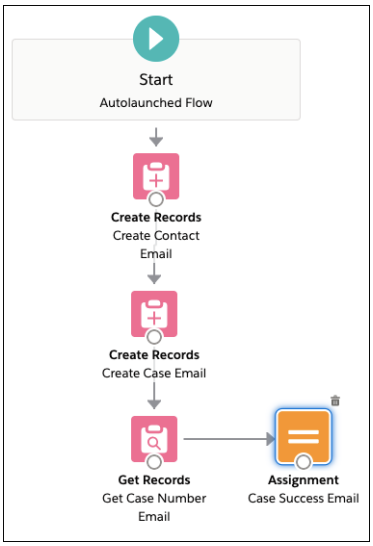
In the Bot Builder, let's add two question dialog steps to ask for the customer name followed by a Message dialog step to tell the customer what happens on the backend:



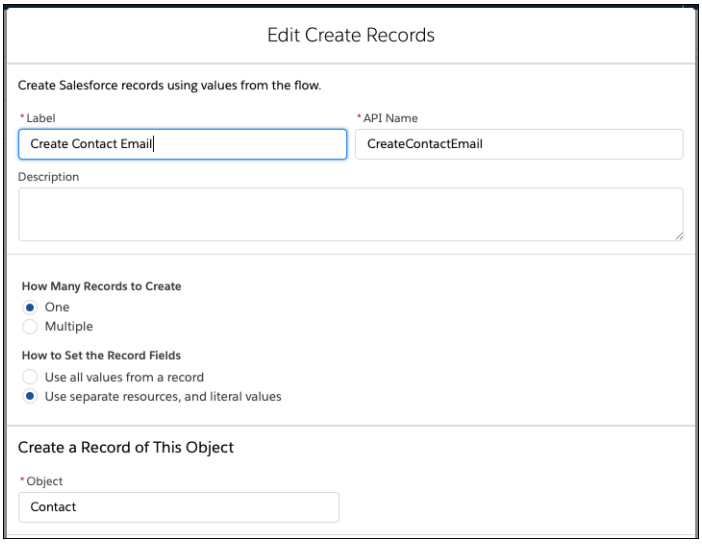



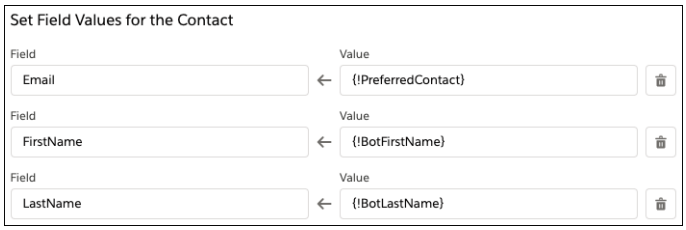

Click **Save**.

**17.** Returning to the Flow Builder, we can build out an autolaunched flow to create a contact, then a case, and output a success message:

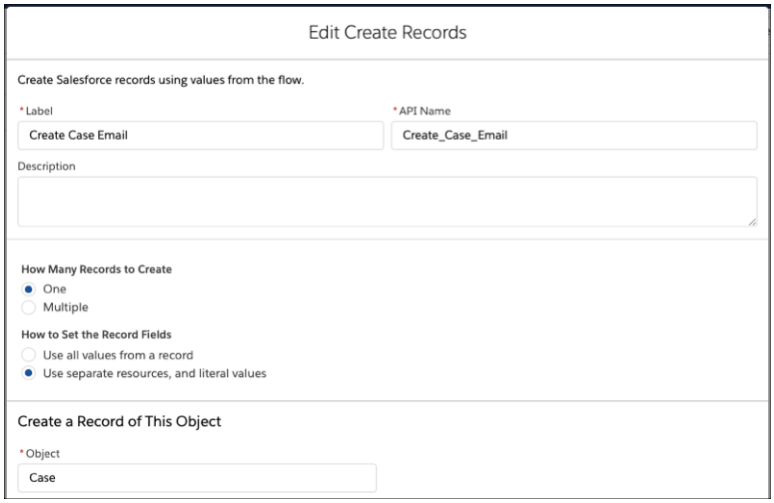


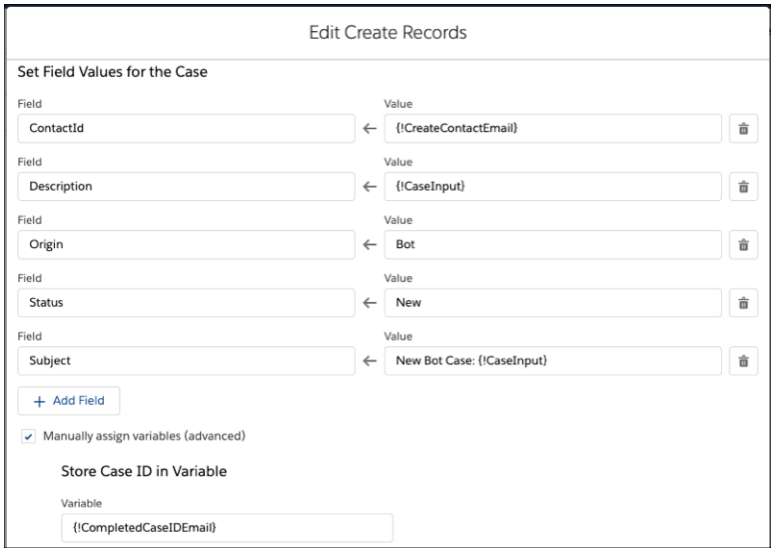

We use the Create Records element to create a contact:



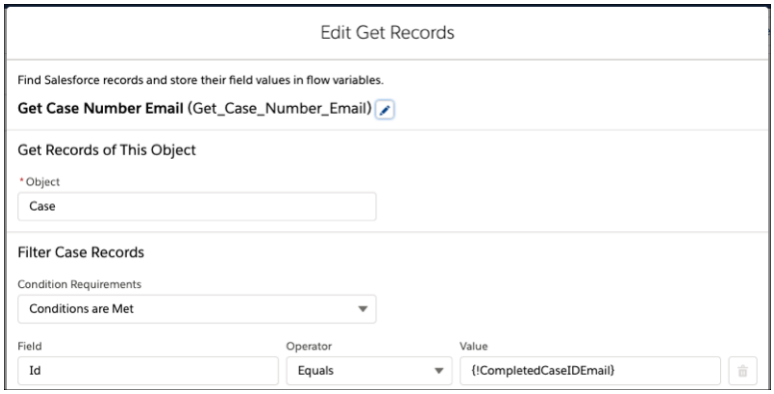




Then, we create a case record using this new contact:

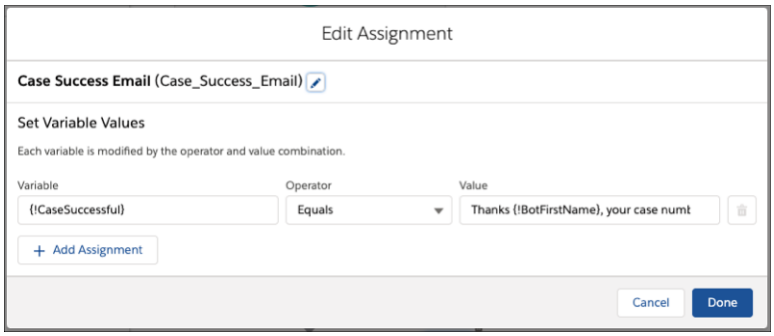



Then, we use the Get Records element to get the case number:

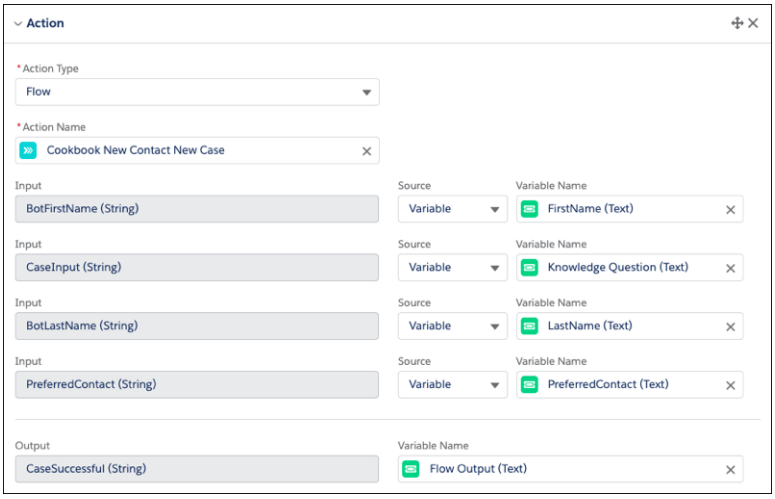


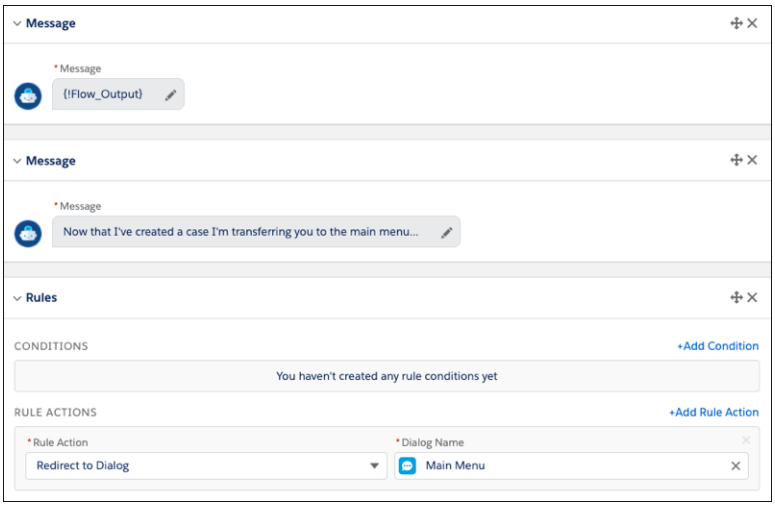
And finally, we set the output variable to the success message including the Case Number using the message `Thanks {!BotFirstName}, your case number is {!Get_Case_Number_Email.CaseNumber}.`




Set up the flow to match the diagram shown at the beginning of this step. Make sure you save and activate this new flow.

**18.** Return to the Bot Builder to set up this second flow and send the end user back to the main menu:



### Fit and Finish

You can use this recipe as a foundation to build on in many ways. For instance, you can add complexity to ensure that the email comes in through a specific format, and you can use Conversation Repair to let the customer know the appropriate format to enter.

When you test the bot and walk through the flow multiple times, or if you expect the customer to search the knowledge base multiple times in one session, clear out variables at the top of each dialog step so that the bot doesn't use the previous values. Simply add a Rule Dialog Step at the beginning of the dialog and add the Clear Variable Value for each variable that is used in the dialog.


It's amazing how much you can automate with clicks! Bots built with flows paired with a well-considered conversation design can create powerful, sustainable solutions for your business.

# Advanced Bot Recipes

Use these advanced recipes to learn how to take your bot to the next level.

Before working through these recipes, go through the recipes in Get Started with Bots and Beginner Bot Recipes.

Create Dynamic Menus
In this recipe, we learn how to create menus that change based on the context of a conversation.

Get Context Info from the Web
In this recipe, we pass along some context information to the bot so that it can transfer serious issues directly to the agent.

Handle Intent Detection Failures
Confused? We've got your back! In this recipe, we handle intent detection failures more gracefully.

# Create Dynamic Menus

In this recipe, we learn how to create menus that change based on the context of a conversation.

Smarter bots ease the load on your service agents. There are two common questions about how to build smarter bots:

- How does a bot perform advanced business logic or integrate with external systems? In an earlier recipe, we revealed that the secret is Apex invocable methods, on page 38 which do the heavy lifting.
- How can you create dynamic menus? In this recipe, we'll show how to use an Apex invocable method to do just that.

The main menu and submenu configuration from Bot Builder is easy to understand—you create new dialogs and point your menu options to them. When you ask a question, you enter fixed menu options that the Einstein Bot presents. But these approaches create static menus. In a real world use case, you want the menu to change based on data, such as customer information.

Let's think about our "Order Status" inquiry experience again. Traditionally, one reason that a business wants to add an order number on a pre-chat form is to make sure that customers have that information ready before the chat starts. Otherwise, the agent has to look up the customer record and search for the customer's orders. If there are multiple orders, the agent goes through the list with the customer. A lot of time can be wasted.

With the help of Einstein Bots, a business can provide the experience that a customer expects, without worrying about agent productivity. The bot can find recent orders and present the list to the customer in a split second. And with the enhanced chat interface, customers can simply select the right order from the menu, instead of typing the order number in chat.

In this recipe, we'll start to build out our "Appointment Related" conversations. Let's pick the transaction type "Reschedule an Appointment". The first step is to confirm which appointment to update.

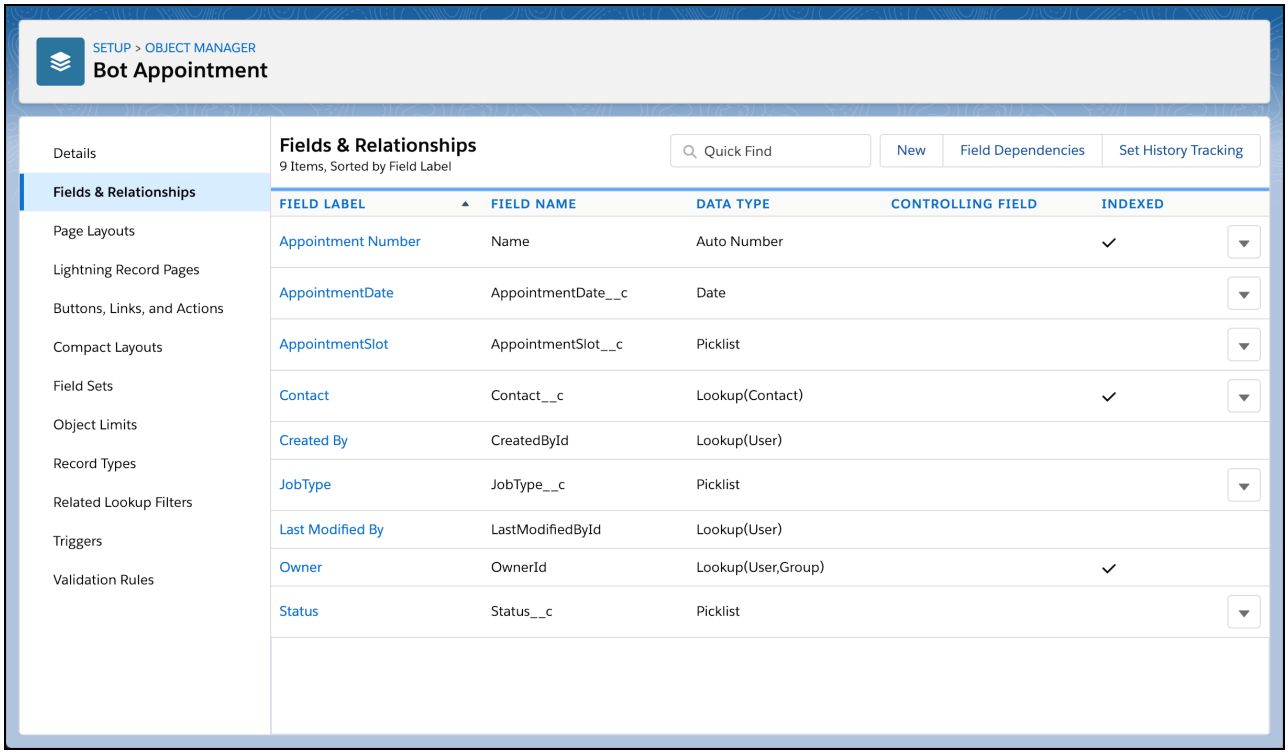
1. Create a custom object to store appointment information.

   Similar to the "Order Status" use case (that we began in Gather Customer Information Using the Bot), let's assume that our appointment data is stored in a custom object named `Bot_Appointment__c`. Create this custom object now. The fields in this object are:

   - `Name` (Text)
   - `Status__c` (Picklist): `New`, `Scheduled`, `Completed`, `Cancelled`
   - `AppointmentDate__c` (Date)
   - `AppointmentSlot__c` (Picklist): `Morning`, `Afternoon`, `Evening`
   - `JobType__c` (Picklist): `Installation`, `BreakAndFix`, `Maintenance`
   - `Contact__c` (lookup to `Contact`)

   The actual appointment number is stored in the `Name` field.

   When you build this `Bot_Appointment` object in your org, it should look like this screenshot.



**Important:** As mentioned in Set Up Your First Einstein Bot, you need to give the bot permission to access this custom object and these fields. From Setup, use the Quick Find box to find **Permission Sets**. Select the **sfdc.chatbot.service.permset** permission set. From the permission set options, select **Object Settings**. Select your custom object and then make sure that this permission set has read and edit permissions to the object and all the relevant fields.

Create a few test records in this object, including a few records with the `Status` value of `New` or `Scheduled`, so that we can reschedule an appointment later on. Also note the email address for the contact associated with the appointments. That's what you'll be using to test this functionality.

2. Create an invocable method in Apex to get the list of appointments.

   Now we can build our invocable Apex method using the following code. If you need some guidance on how to create and use invocable Apex code, review the earlier recipe on Apex: Call an Apex Action.

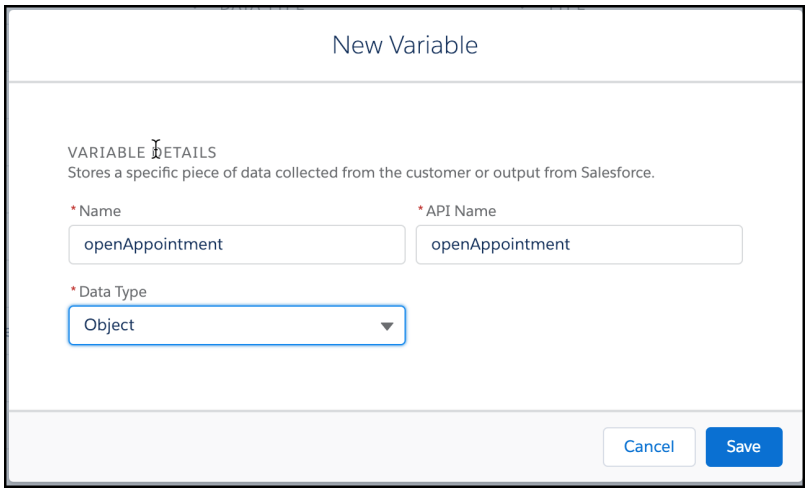
```
public with sharing class CookbookBot_GetOpenAppointments {

  @InvocableMethod(label='Get Open Appointments')
  public static List<List<Bot_Appointment__c>> getOpenAppointments(List<String> sEmails)
 {

    String sEmail = sEmails[0];

    // Get the list of new and scheduled appointments
    List<Bot_Appointment__c> appointments = [SELECT Id, Name, JobType__c,
                                                AppointmentDate__c, AppointmentSlot__c

                                            FROM Bot_Appointment__c
                                            WHERE Contact__r.Email =:sEmail
                                            AND Status__c IN ('New','Scheduled')];

    // Create a list for the appointments
    // NOTE: This is a list of lists in order to handle bulk calls...
   List<List<Bot_Appointment__c>> appointmentList = new List<List<Bot_Appointment__c>>();


    // Add all the new and scheduled appointments to the list
    appointmentList.add(appointments);

    return appointmentList;
  }
}
```

A few comments about this code:

- The `getOpenAppointments()` method performs a SOQL query to find the open appointment records (i.e., where status is `New` or `Scheduled`) that are associated with this contact's email address. The method returns the list of `Bot_Appointment__c` records.
- There is no `@InvocableVariable` declared in this class. Since we only have one input parameter and one output parameter, we skipped that part.
- The return variable of the `getOpenAppointments()` method needs to be `List<List<SObject>>`. The inner `List` has all the open appointment records associated with this email address. The outer `List` is required by the `@InvocableMethod` annotation to allow this function to be bulk ready (in case you want to find open appointments for multiple contacts). For our use case, we only need to operate on one contact record, so there's only one element in the outer `List`.

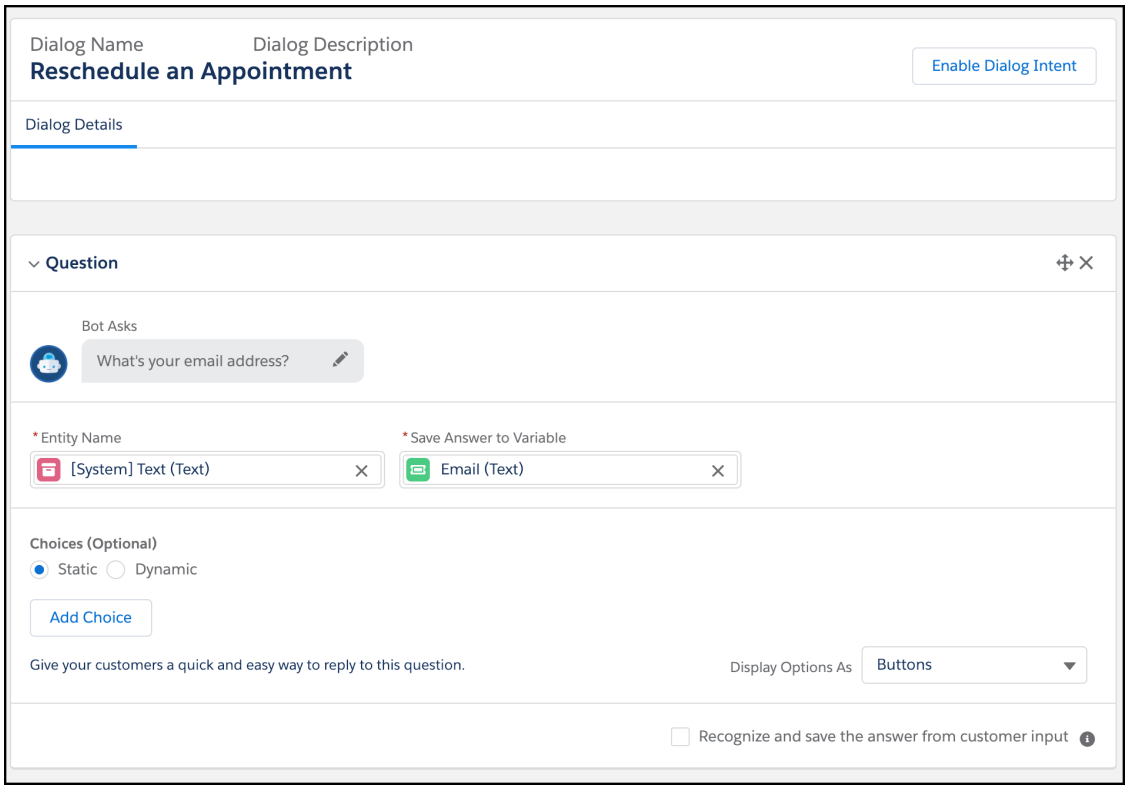
3. Create an object variable.

   Now let's present the list of records from the previous step and their values in a menu. The menu option that's selected needs to be stored in a variable so that we can use it to further drive the interaction. Let's create a variable named `openAppointment`. The data type of the variable should be set to **Object**.

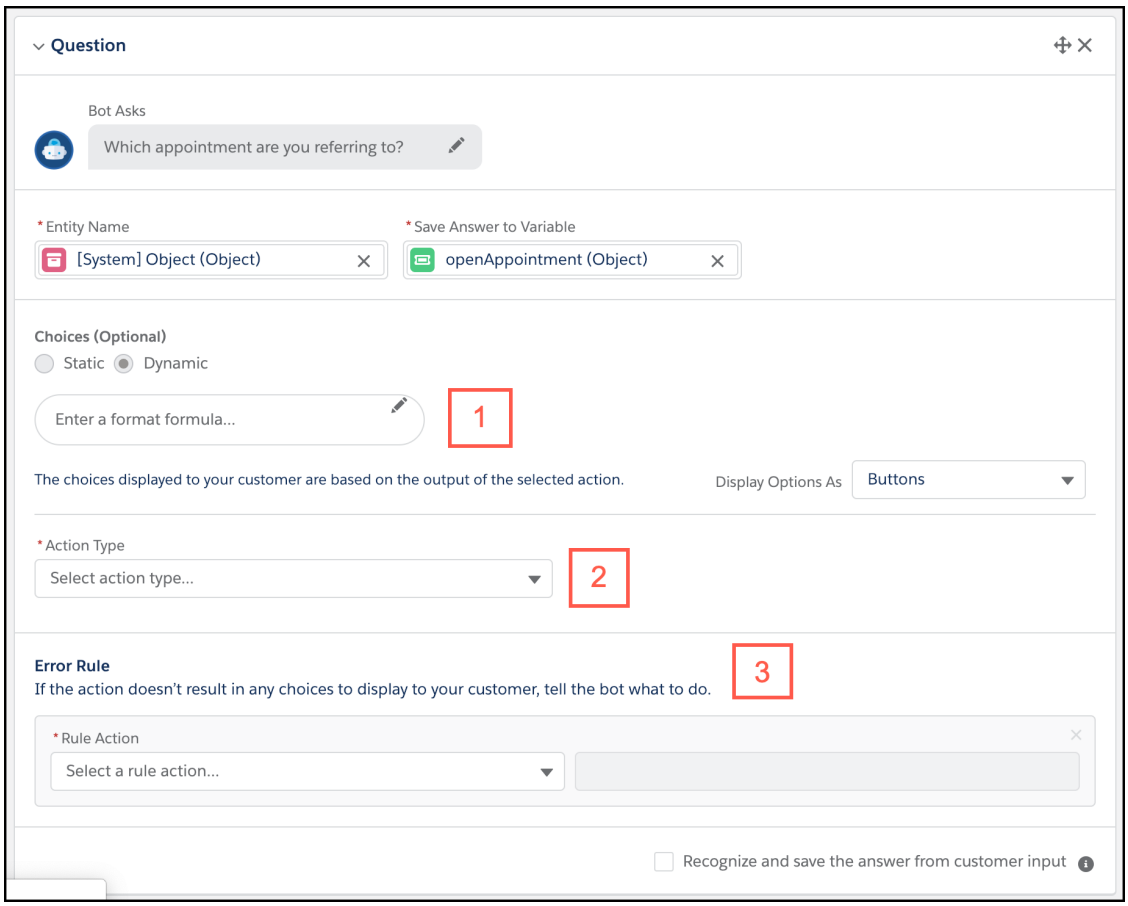



4. Call the "Get Open Appointments" Apex method and present dynamic options using a question.

   Let's build the "Reschedule an Appointment" dialog. First, we'll ask for the email address used to book the appointment. We can reuse the same variable **Email** to capture the customer answer. As we already learned, the bot skips this question if the customer entered the email address previously, for example, in the pre-chat form, as long as the address is passed to the same variable (see Optimize Bot Flow with Pre-Chat Data).



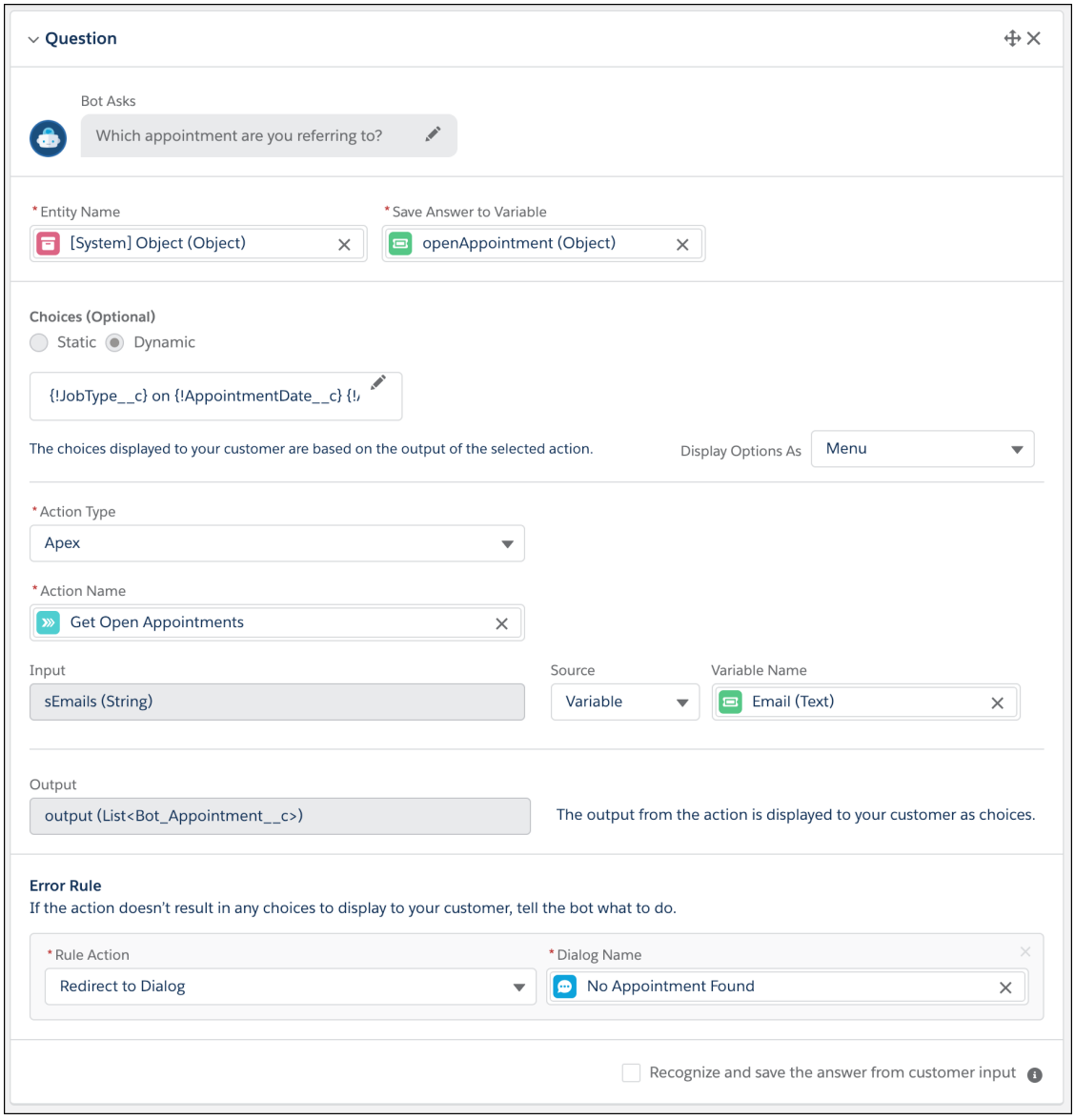
Next, we'll add another **Question** element that presents the list of options. When we set the answer variable to an **Object** entity, such as `openAppointment`, and select the **Choices** option as **Dynamic**, the **Question** element configuration area expands with additional settings for a formula (1), an action (2), and an error handling mechanism (3).



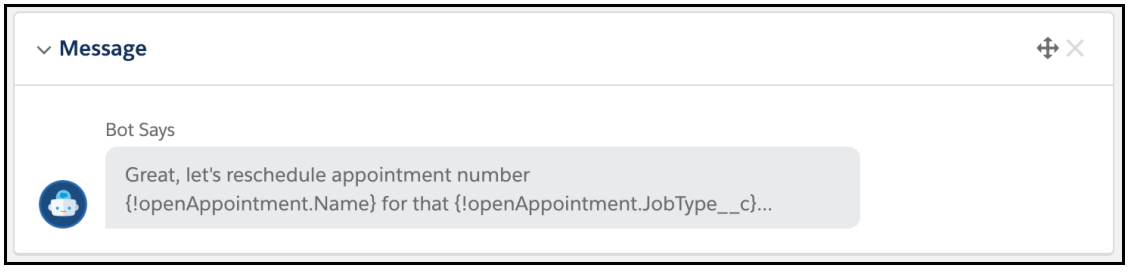

The format formula gives administrators flexibility in determining the format of menu options. We can merge fields from the Salesforce object using their field API names. In this case, set the formula to: `{!JobType__c} on {!AppointmentDate__c} {!AppointmentSlot__c}`. You don't have to prefix the variable names with the object type because this entry is scoped to the Salesforce object itself. You can try different fields, but make sure they are included in the SOQL query in the `getOpenAppointments()` Apex method. And make sure that the field level security is enabled in the **sfdc.chatbot.service.permset** permission set.

Select Apex for the **Action Type**. In the **Action Name** field, set it to the new Apex class that we created ("Get Open Appointments"). When you select the action name, the input and output parameter sections also expand based on the signature of the Apex method. Set the **Email** variable as the input for this Apex action.

The **Error Rule** handles the situation when Apex returns no records. In that case, we redirect the customer to a new dialog named "No Appointment Found". We can leave the this new dialog empty for now.



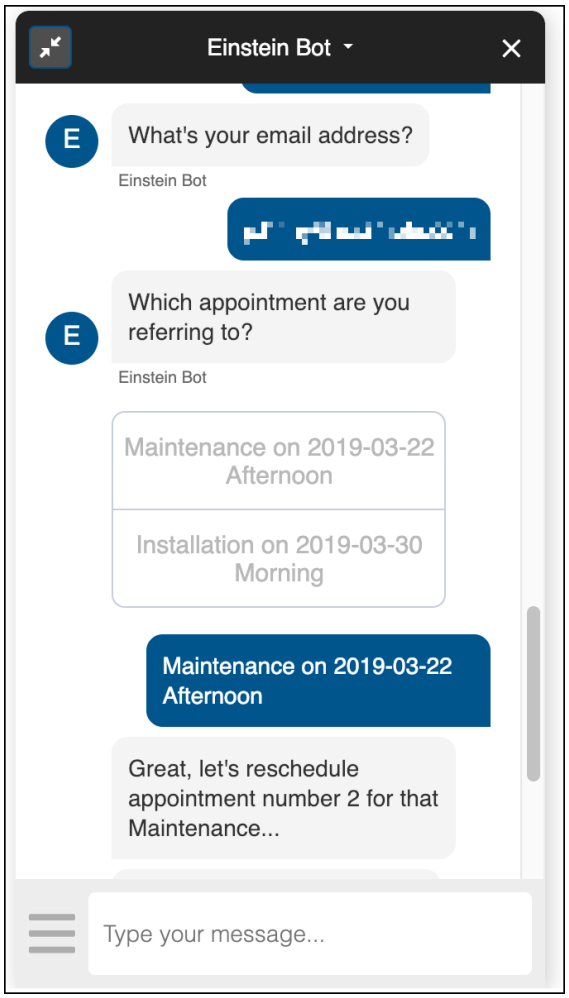

After the customer selects an option from the menu, the variable is set to the selected record. You can use this info however you want.




In our example, the bot acknowledges the customer selection in this response: `Great, let's reschedule appointment number {!openAppointment.Name} for that {!openAppointment.JobType__c}...`

5. Test!

Let's see how our dynamic menu looks now. Make sure you enter a valid email address that's associated with one of the contacts on a test appointment record in the org.




### Best Practices and Additional Notes

- What happens when you enter an email address that's not associated with a contact on your test appointment records? Not much, because we didn't build out the "No Appointment Found" dialog in the **Error Rule** yet. Add a Yes/No question to confirm the email address. If the email is wrong, send them back to the "Reschedule an Appointment" dialog so that the customer can try again. Don't forget that you have to clear the Email variable so the bot won't skip the question the second time.
- Selecting an appointment to update from a list is only the first step of a rescheduling transaction. How many Apex actions do we need to complete this inquiry type? There are roughly three potential actions: `getOpenAppointments()`, `getAvailableSlots()`, and `saveUpdatedAppointment()`. Can you complete the other two Apex actions?
- Looking back at our user experience for the "Reschedule an Appointment" conversation, once the chat starts, the customer is greeted and presented with the main menu. They can select **Appointment Related** and then **Reschedule an Appointment** in the submenu to start this transaction. From this dialog, they select an open appointment, select an available time slot, and get a confirmation message. Four taps on a mobile device. In a happy path scenario, they don't have to type anything at all. See how a bot can change the way we interact with our customers through the traditional chat channel?

# Get Context Info from the Web

In this recipe, we pass along some context information to the bot so that it can transfer serious issues directly to the agent.

In Optimize Bot Flow with Pre-Chat Data, we discussed how to pass customer input from the pre-chat form to the bot. What about information we already know about the customer and their activities on the website? Are they authenticated? Which knowledge articles have they seen before coming to the chat for help? Do they have $1000 worth of stuff in their shopping cart, but couldn't finish the payment process?

Typically a business tries their best to save a high value cart from being abandoned. If that happens, you may want to offer an invitation to chat.

Let's build out this user story so that all customers who get stuck on the payment page with $500 or more worth of items in their shopping cart are transferred to an agent.

Get Context Info with Embedded Chat
If you're using Snap-ins Chat, follow these steps to get context information from outside the chat session.

Get Context Info in Salesforce Classic
If you're using Salesforce Classic, follow these steps to get context information from outside the chat session.

## Get Context Info with Embedded Chat

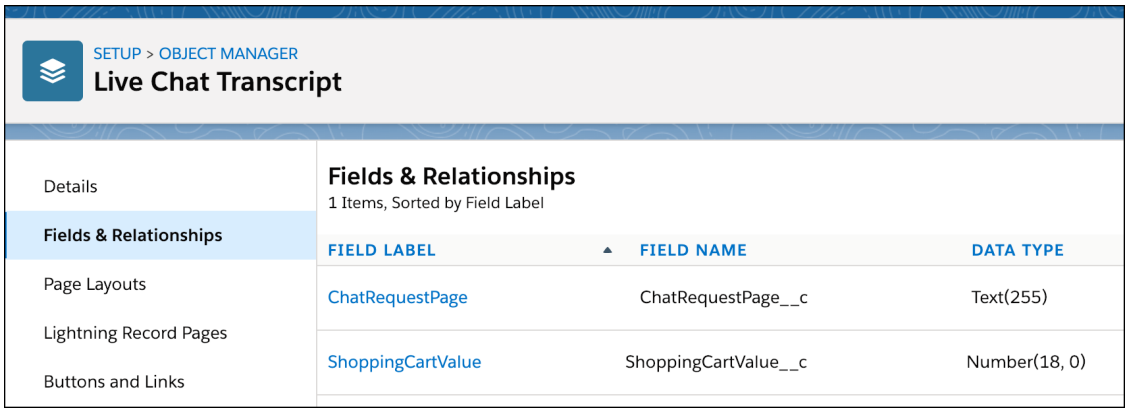
If you're using Snap-ins Chat, follow these steps to get context information from outside the chat session.

1. Add custom fields to the `LiveChatTranscript` object.

   Let's use two variables to identify customers on the payment page that have shopping carts valued at more than $500:

   - **ChatRequestPage**: The page from which the customer is requesting help.
   - **ShoppingCartValue**: The total dollar value of the customer's online shopping cart.

   Since the bot pre-chat form action uses the `LiveChatTranscript` object to pull pre-chat data, we create a custom **Text** field `ChatRequestPage__c` and a custom **Number** field `ShoppingCartValue__c` on this object.




2. Modify the chat code snippet and add these new fields.

   We created the `extraPrechatFormDetails` array in our JavaScript file in Optimize Bot Flow with Embedded Chat. Add the variables `ChatRequestPage` and `ShoppingCartValue` to this array. For testing, to indicate that the customer is on the payment page and the shopping cart has $600 worth of items, hard code the values of "Payment" and "600" in these variables.

   Note the difference between these and other pre-chat fields, such as `Name`, `Email`, and `OrderNumber`. For the `ChatRequestPage` and `ShoppingCartValue` variables, we are actually setting the value attribute in the array. Data for

other fields would actually come from whatever the customer entered on the pre-chat form. Also, specify the `transcriptFields` attribute so that data is saved to the mapped field in the `LiveChatTranscript` record.

**a.** Code change for a customer site:

```
embedded_svc.settings.extraPrechatFormDetails = [{
  "label":"First Name",
  "transcriptFields": ["FirstName__c"]
},{
  "label":"Last Name",
  "transcriptFields": ["LastName__c"]
},{
  "label":"Email",
  "transcriptFields": ["Email__c"]
},{
  "label":"OrderNumber",
  "transcriptFields": ["OrderNumber__c"]
},{
  "label": "ChatRequestPage",
  "value": "Payment",
  "displayToAgent": true,
  "transcriptFields": ["ChatRequestPage__c"]
},{
  "label": "ShoppingCartValue",
  "value": "600",
  "displayToAgent": true,
  "transcriptFields": ["ShoppingCartValue__c"]
}];
```

**b.** Code change for an Experience Cloud site:

```
embedded_svc.snippetSettingsFile.extraPrechatFormDetails = [{
  "label":"First Name",
  "transcriptFields": ["FirstName__c"]
},{
  "label":"Last Name",
  "transcriptFields": ["LastName__c"]
},{
  "label":"Email",
  "transcriptFields": ["Email__c"]
},{
  "label":"OrderNumber",
  "transcriptFields": ["OrderNumber__c"]
},{
  "label": "ChatRequestPage",
  "value": "Payment",
  "displayToAgent": true,
  "transcriptFields": ["ChatRequestPage__c"]
},{
  "label": "ShoppingCartValue",
  "value": "600",
  "displayToAgent": true,
  "transcriptFields": ["ShoppingCartValue__c"]
}];
```

3. Populate the bot variables with pre-chat form actions.

   In Bot Builder, create two variables, **ChatRequestPage** and **ShoppingCartValue**. Since the bot can properly cast a pre-chat form field to the correct data type, we can use **Text** as the data type for the **ChatRequestPage** variable, and **Number** as the data type for the **ShoppingCartValue** variable.

   In the same "Bot Initialization" dialog where we set first name, last name, email, and order number variables (in Optimize Bot Flow with Pre-Chat Data), we also want to populate the two new variables. As we discussed in Greet the Customer with Embedded Service Chat (and then again in Optimize Bot Flow with Embedded Chat), you can access pre-chat fields using the Metadata API or by writing some Apex code. Using the example in Greet the Customer with Embedded Service Chat, extend that functionality and include these two variables. Use the following code snippets to check your work.

   a. (Option 2 of 2: **Metadata API Solution**) Use this XML to get you going with the Metadata API approach. If you need more help, revisit the metadata solution in Greet the Customer with Snap-Ins Chat on page 13.

```
<contextVariables>
    <contextVariableMappings>
        <SObjectType>LiveChatTranscript</SObjectType>
        <fieldName>LiveChatTranscript.ChatRequestPage__c</fieldName>
        <messageType>WebChat</messageType>
    </contextVariableMappings>
    <dataType>Text</dataType>
    <developerName>ChatRequestPage</developerName>
    <label>ChatRequestPage</label>
</contextVariables>
<contextVariables>
    <contextVariableMappings>
        <SObjectType>LiveChatTranscript</SObjectType>
        <fieldName>LiveChatTranscript.ShoppingCartValue__c</fieldName>
        <messageType>WebChat</messageType>
    </contextVariableMappings>
    <dataType>Number</dataType>
    <developerName>ShoppingCartValue</developerName>
    <label>ShoppingCartValue</label>
</contextVariables>
```

   b. (Option 1 of 2: **Apex Code Solution**) Use this code to get you going with the Apex solution. Notice that the data type for the `ShoppingCartValue` variable is `Decimal`. If you need more help, revisit the Apex solution in Greet the Customer with Snap-Ins Chat on page 12.

```
public with sharing class CookbookBot_GetTranscriptVariables {

  public class TranscriptInput {
    @InvocableVariable(required=true)
    public ID routableID;
  }

  public class TranscriptOutput {
    @InvocableVariable(required=true)
    public String sFirstName;

    @InvocableVariable(required=true)
    public String sOrderNumber;

    @InvocableVariable(required=true)
```

```
    public String sChatRequestPage;

    @InvocableVariable(required=true)
    public Decimal nShoppingCartValue;
  }

  @InvocableMethod(label='Get Transcript Variables')
  public static List<TranscriptOutput> getUserName(List<TranscriptInput> transcripts)
 {

    List<TranscriptOutput> outputList = new List<TranscriptOutput>();

    for (TranscriptInput transcript : transcripts) {

      // Query for the transcript record based on the ID
     LiveChatTranscript transcriptRecord = [SELECT Name, FirstName__c, OrderNumber__c,

                                                    ChatRequestPage__c,
ShoppingCartValue__c
                                            FROM LiveChatTranscript
                                            WHERE Id = :transcript.routableID
                                            LIMIT 1];

      TranscriptOutput output = new TranscriptOutput();

      // Store the FirstName field in an output variable
      output.sFirstName = transcriptRecord.FirstName__c;

      // Store the OrderNumber field in an output variable
      output.sOrderNumber = transcriptRecord.OrderNumber__c;

      // Store the the ChatRequestPage field in an output variable
      output.sChatRequestPage = transcriptRecord.ChatRequestPage__c;

      // Store the the ShoppingCartValue field in an output variable
      output.nShoppingCartValue = transcriptRecord.ShoppingCartValue__c;

      // Add the values to the list of outputs
      outputList.add(output);
    }

    return outputList;
  }
}
```

4. Verify that we correctly extracted the values from our new fields.

   Temporarily add the following text to the welcome message: `{!ShoppingCartValue} and {!ChatRequestPage}`. If the changes are correct, you see "600 and Payment" when you run the bot. If not, confirm that your bot has access to the new `LiveChatTranscript` fields and any new Apex code. To learn more, review setting up permissions in Set Up Your First Einstein Bot. Also, verify that the API field names are correct in all your code snippets.

   Now let's look at how we can use this context information to improve the user experience. Our goal is to transfer a customer that's stuck on the payment page to an agent.

5. Add a new "Transfer Required" dialog.

   In Optimize Bot Flow with Pre-Chat Data, we optimized the conversation for customers that provided an order number on the pre-chat form. Let's do it again now that we have more context when a customer requests a chat.

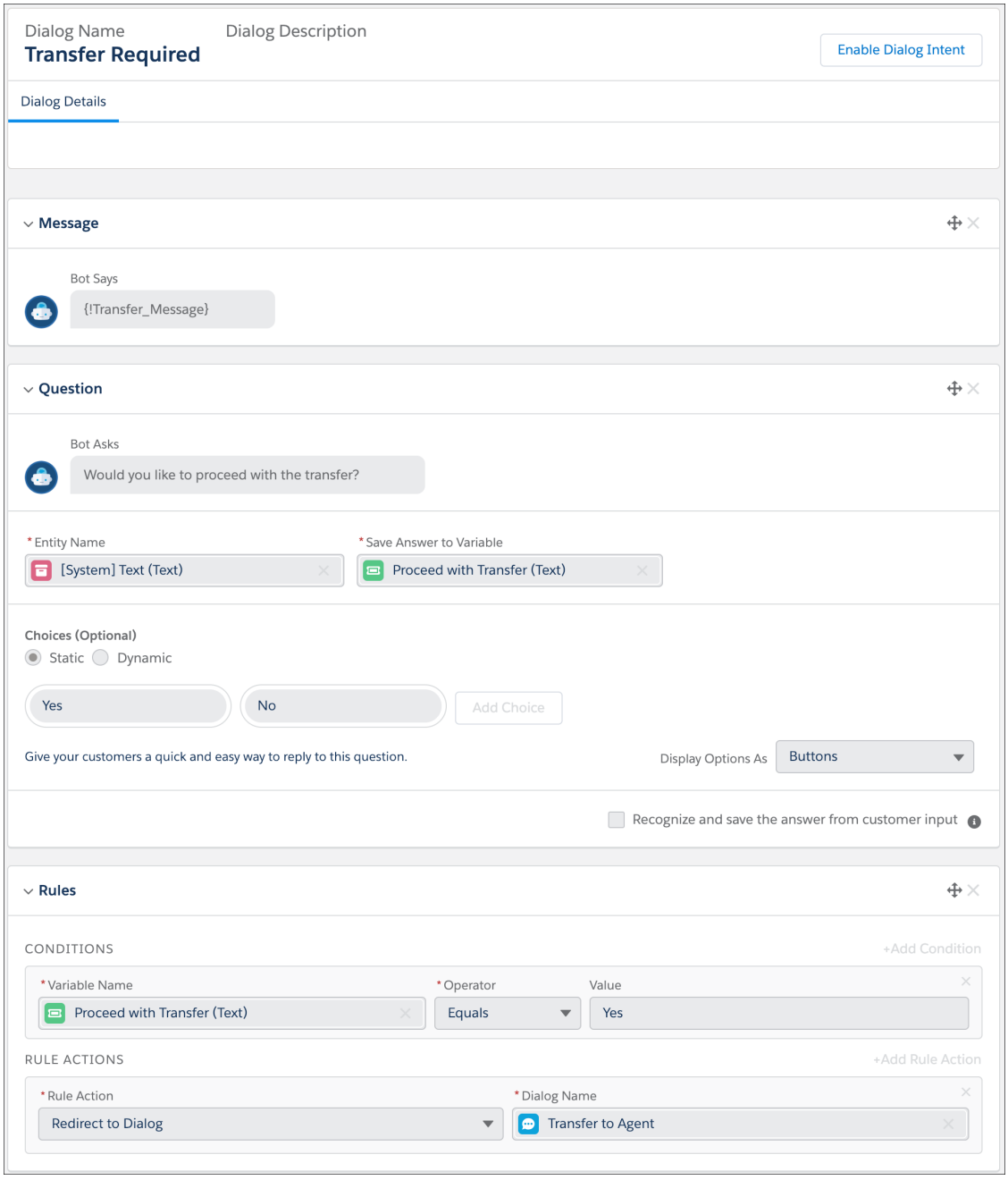
   Do we build another dialog called "Welcome customer with a high value shopping cart"? We can. But as we introduce more context variables and rules, the number of dialogs can grow quickly for greeting the customer. Another approach is to build a utility or helper dialog that uses variables to serve different situations and deliver different messages. This approach is more scalable. Let's try it.

   Add a "Transfer Required" dialog that handles when the bot recommends agent assistance.

   a. Set the first message element to include just a new variable, "{!Transfer_Message}".

   b. Ask if the customer agrees to the transfer.

   c. If they agree, redirect them to the "Transfer to Agent" dialog.




By configuring the message within a variable, we can use an Apex action, for example, to populate message variables for different situations.

6. Create a "Bot Initialization" Apex action.

   Set up some rules in the original welcome screen that conditionally redirects the customer. The rule follows this logic:

```
Redirect if ChatRequestPage = 'Payment' and ShoppingCartNumber > $500
```

   Let's write some Apex code to handle this logic. The code takes two input parameters (`sChatRequestPage` and `sShoppingCartValue`), and returns two `@InvocableVariables`, `bInitialTransferRequired` and `sInitialTransferMessage`, in the wrapper class. The code does two things.

**a.** Sets boolean `bInitialTransferRequired` to true when `sChatRequestPage='Payment'` and `nShoppingCartNumber > 500`.

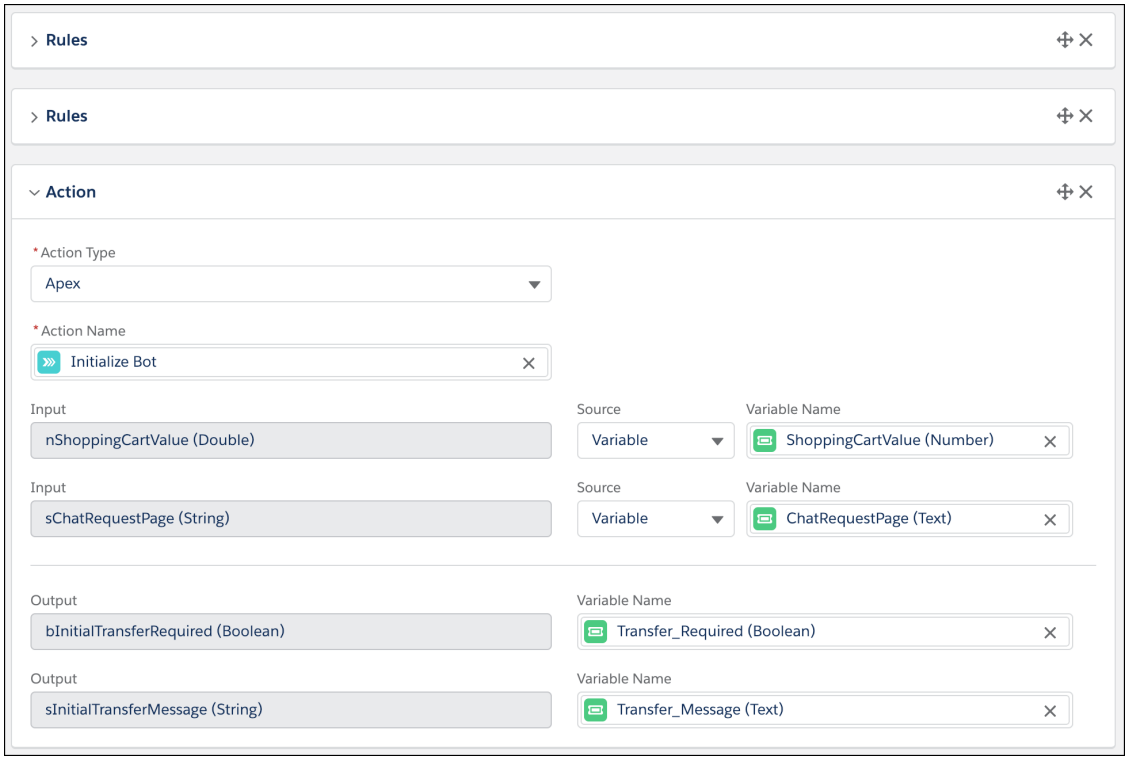
**b.** Sets the message in the text variable `sInitialTransferMessage`.

```
public with sharing class CookbookBot_InitializeBot {
   public class InitializationOutput{
       @InvocableVariable(required=true)
       public String sInitialTransferMessage;
       @InvocableVariable(required=true)
       public Boolean bInitialTransferRequired;
   }

   public class InitializationInput{
       @InvocableVariable(required=false)
       public String sChatRequestPage;
       @InvocableVariable(required=false)
       public Decimal nShoppingCartValue;
   }

   @InvocableMethod(label='Initialize Bot')
   public static List<InitializationOutput> initializeBot(List<InitializationInput>
inputParameters)
   {
       // Grab the input variables
       String sChatRequestPage = inputParameters[0].sChatRequestPage;
       Decimal nShoppingCartValue = inputParameters[0].nShoppingCartValue;

       // Initialize output variables
      List<InitializationOutput> outputParameters = new List<InitializationOutput>();

       InitializationOutput outputParameter = new InitializationOutput();

       // Is this a condition where we want to transfer to an agent?
       if (sChatRequestPage=='Payment' && nShoppingCartValue > 500)
       {
           // Set the output variables for a transfer
           outputParameter.bInitialTransferRequired = true;
           outputParameter.sInitialTransferMessage = 'Looks like you may have a
payment-related question. I can find one of our specialists to help you...';
       }
       else
       {
           // Set the output variables for no transfer required
           outputParameter.bInitialTransferRequired = false;
           outputParameter.sInitialTransferMessage = '';
       }

       // Return result
       outputParameters.add(outputParameter);
       return outputParameters;
   }
}
```

By now, you know how to add an Apex class and make sure that your bot has permission to access the class. If not, see Call an Apex Action.

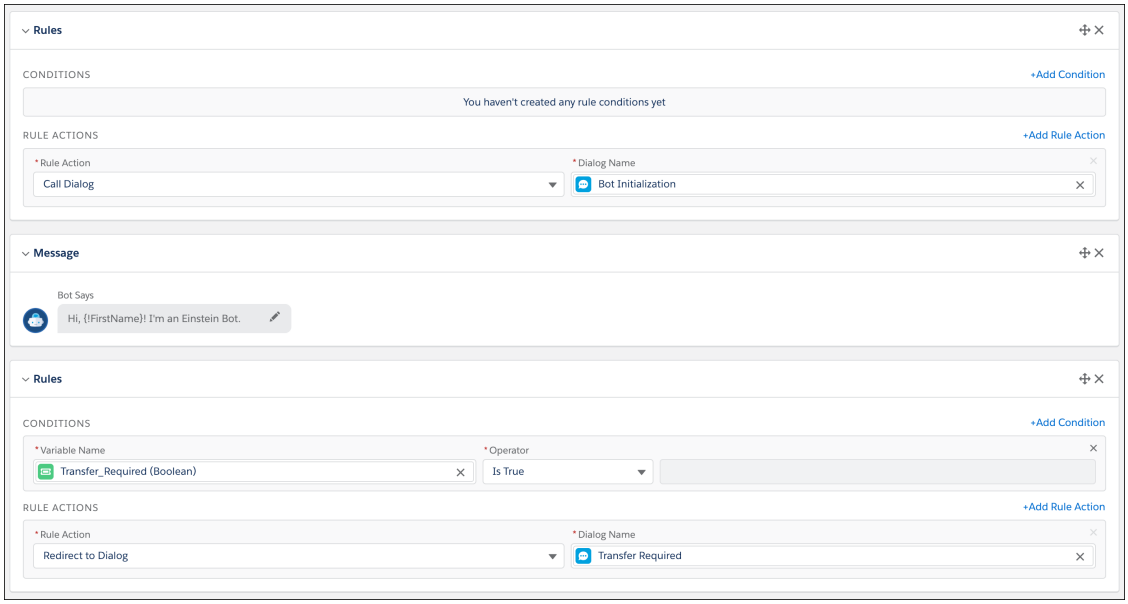
7. Call the "Initialize Bot" Apex action from the "Bot Initialization" dialog.

   In the "Bot Initialization" dialog, *after* we set the `ShoppingCartValue` and `ChatRequestPage` variables with pre-chat form actions, we can call the new Apex action that we built in the previous step. Create two new variables, `Transfer_Required` and `Transfer_Message`, to take the output value of the Apex action, which drives the bot experience.




8. Update the "Welcome" dialog with the new conversation design.

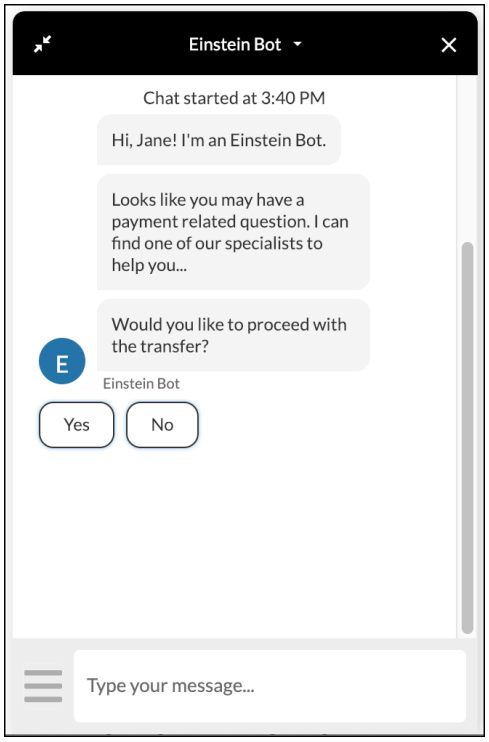
   In the Optimize Bot Flow with Pre-Chat Data recipe, you created a call to the "Bot Initialization" dialog as the first step in the Welcome screen, followed by a greeting message. Let's add a rule to redirect customers to the "Transfer Required" dialog when our initialization action shows that it's warranted.



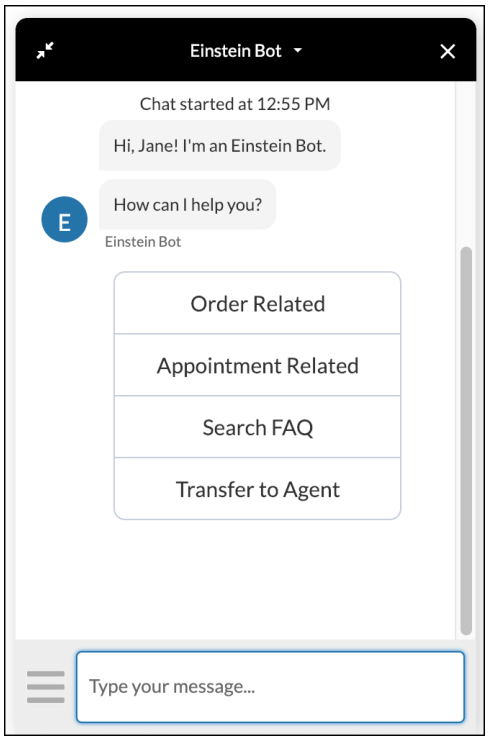

If you look at the available **Rule Action** options, there's a **Call Dialog** action and a **Redirect to Dialog** action. Here's the difference: with **Redirect to Dialog**, the destination dialog takes over the conversation, while **Call Dialog** returns the conversation back to the originating dialog. Hence, we use **Call Dialog** for "Bot Initialization" (because we want the flow to return back to this dialog), and we use **Redirect to Dialog** for the "Transfer Required" dialog (where we want to hop over to the other dialog and not return). Note that when we use **Redirect to Dialog**, whatever configuration we selected in the **Next Dialog** section of the originating dialog is ignored.

9. Test!

Let's do two rounds of testing. First, let's use the original chat snippet that had the `ShoppingCartValue` set to 600.



Now let's change the JavaScript that we wrote earlier so that the `ShoppingCartValue` is 300.




**Best Practices and Additional Notes**

- In this use case, one disadvantage is the hard-coded business logic in our "Bot Initialization" Apex action. For example, if you change the `ShoppingCartValue` that triggers a transfer, you must change that value in Apex. A more flexible approach is to create a custom setting or a record in a custom object.
- Can a business user change the bot behavior or are admins responsible for all configuration? It may not be a good idea to have everyone become an administrator and change the bot how they want. Instead, take advantage of custom settings or custom objects that make the bot configuration more data-driven. You can then give access to users so that they can change configuration-related data records. In this way, you give business users flexibility, but in a controlled manner.

## Get Context Info in Salesforce Classic

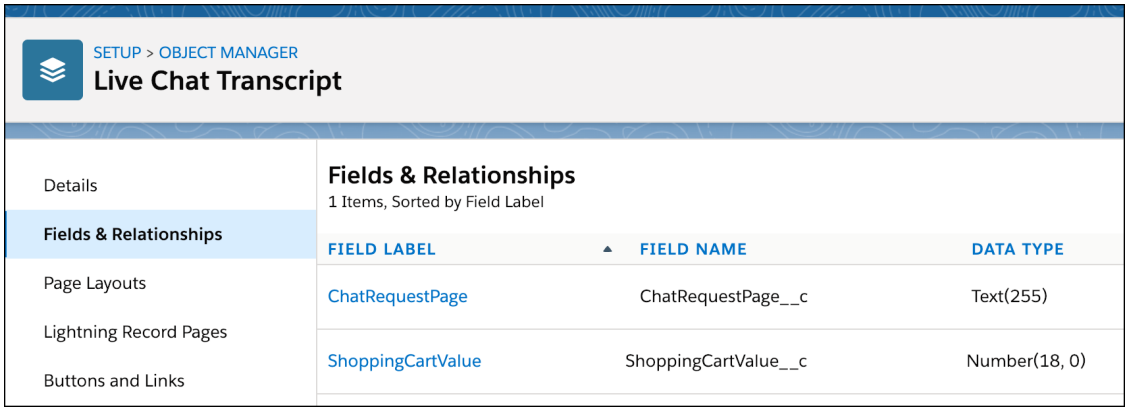
If you're using Salesforce Classic, follow these steps to get context information from outside the chat session.

1. Add custom fields to the `LiveChatTranscript` object.

   Let's use two variables to identify customers on the payment page that have shopping carts valued at more than $500:

   - **ChatRequestPage**: The page from which the customer is requesting help.
   - **ShoppingCartValue**: The total dollar value of the customer's online shopping cart.

   Since the bot pre-chat form action uses the `LiveChatTranscript` object to pull pre-chat data, we create a custom **Text** field `ChatRequestPage__c` and a custom **Number** field `ShoppingCartValue__c` on this object.




2. Modify the chat request page to include custom details.

   Open the HTML landing page that we created when we first set up the bot in Set Up Your First Einstein Bot.

   Right before the `liveagent.init()` method, add two JavaScript variables and two `addCustomDetail()` functions to make them available as part of custom chat detail variables. For testing, to indicate that the customer is on the payment page and the shopping cart has $600 worth of items, hard code the values of "Payment" and "600" in these variables.

```
<script type='text/javascript'
src='https://c.gla3-phx.MyDomainName.salesforce.com/content/g/js/42.0/deployment.js'></script>
<script type='text/javascript'>

<!-- NEW CODE GOES HERE -->
var chatRequestPage = 'Payment';
var shoppingCartValue = 600;
liveagent.addCustomDetail('ChatRequestPage',
chatRequestPage).saveToTranscript('ChatRequestPage__c');
liveagent.addCustomDetail('ShoppingCartValue',
shoppingCartValue).saveToTranscript('ShoppingCartValue__c');
<!-- END OF NEW CODE -->

liveagent.init('https://d.gla3-phx.MyDomainName.salesforce.com/chat', '572B0000000513C',
 '00DB0000000LgOe');
</script>
```

   The other thing to note is that the `saveToTranscript()` method maps the `ChatRequestPage` chat variable to the custom field `LiveChatTranscript.ChatRequestPage__c` and the `ShoppingCartValue` chat variable to the custom field `LiveChatTranscript.ShoppingCartValue__c`. This way, these two pieces of information are saved to the `LiveChatTranscript` when the record is created.

3. Modify the pre-chat form to pass on the custom details.

   If you do have a custom pre-chat form for a Classic Live Agent client (see Greet the Customer in Salesforce Classic Chat), you'll need to pass the custom detail variable from the landing page to the pre-chat form, which in turn passes the data to the bot as pre-chat variables. Open the pre-chat page and make the following updates.

```
First name: <input type='text' name='liveagent.prechat:FirstName' id='firstName' /><br
 />
Last name: <input type='text' name='liveagent.prechat:LastName' id='lastName' /><br />
Email: <input type='text' name='liveagent.prechat:Email' id='email' /><br />

<!-- NEW CODE GOES HERE -->
```

```
<input type='hidden' name='liveagent.prechat:ShoppingCartValue' id='ShoppingCartValue'
 />
<input type='hidden' name='liveagent.prechat:ChatRequestPage' id='ChatRequestPage' />
<!-- END OF NEW CODE -->

<input type='hidden' name= 'liveagent.prechat.save:FirstName' value= 'FirstName__c' />
<input type='hidden' name= 'liveagent.prechat.save:LastName' value= 'LastName__c' />
<input type='hidden' name= 'liveagent.prechat.save:Email' value= 'Email__c' />

<!-- NEW CODE GOES HERE -->
<input type='hidden' name= 'liveagent.prechat.save:ShoppingCartValue' value=
'ShoppingCartValue__c' />
<input type='hidden' name= 'liveagent.prechat.save:ChatRequestPage' value=
'ChatRequestPage__c' />
<!-- END OF NEW CODE -->

<input type='submit' value='Chat Now' id='prechat_submit' onclick="setName()"/>
```

The first two new lines of code indicate that we have two hidden chat variables with the name specified as `liveagent.prechat:ShoppingCartValue` and `liveagent.prechat:ChatRequestPage`. The second two lines of code map these two chat detail variables to the `LiveChatTranscript` fields by their API names.

Use the `preChatInit` callback function to populate these two hidden fields. Add the following section to the pre-chat page. The values are coming from the input parameter `details.customDetails` array, which includes data passed from the landing page where the customer initiated the chat.

```
<script type='text/javascript'
src='https://MyDomainName.my.salesforcescrt.com/content/g/js/32.0/prechat.js'></script>
<script type="text/javascript">
var detailCallback = function (details){
   for (var i = 0; i < details.customDetails.length; i++) {
       if(details.customDetails[i].label == 'ShoppingCartValue'){
           document.getElementById('ShoppingCartValue').value =
details.customDetails[i].value;
       }
       if(details.customDetails[i].label == 'ChatRequestPage'){
           document.getElementById('ChatRequestPage').value =
details.customDetails[i].value;
       }
   }
};

//First parameter is Chat URL. This is same as generated in Live Chat deployment code
and can be used here
liveagent.details.preChatInit('https://d.gla3-phx.MyDomainName.salesforce.com/chat','detailCallback');
</script>
```

There are two URLs in the above code. The first line references the `prechat.js` library. In your code, change the domain and path to be the same as how the `deployment.js` file is referenced on the chat request page. One of the parameters for the `preChatInit()` method is also a URL. You should set it the same as the first parameter in the `liveagent.init()` method in the deployment code. It's generated from **Deployments** in Setup. While you are on the deployment configuration page, verify that the Live Agent deployment has **Allow Access to Pre-Chat API** enabled.

4. Populate the bot variables with pre-chat form actions.

In Bot Builder, create two variables, **ChatRequestPage** and **ShoppingCartValue**. Since the bot can properly cast a pre-chat form field to the correct data type, we can use **Text** as the data type for the **ChatRequestPage** variable, and **Number** as the data type for the **ShoppingCartValue** variable.

In the same "Bot Initialization" dialog where we set first name, last name, email, and order number variables (in Optimize Bot Flow with Pre-Chat Data), we also want to populate the two new variables. As we discussed in Greet the Customer with Embedded Service Chat (and then again in Optimize Bot Flow with Embedded Chat), you can access pre-chat fields using the Metadata API or by writing some Apex code. Using the example in Greet the Customer with Embedded Service Chat, extend that functionality and include these two variables. Use the following code snippets to check your work.

**a.** (Option 2 of 2: **Metadata API Solution**) Use this XML to get you going with the Metadata API approach. If you need more help, revisit the metadata solution in Greet the Customer with Snap-Ins Chat on page 13.

```
<contextVariables>
    <contextVariableMappings>
        <SObjectType>LiveChatTranscript</SObjectType>
        <fieldName>LiveChatTranscript.ChatRequestPage__c</fieldName>
        <messageType>WebChat</messageType>
    </contextVariableMappings>
    <dataType>Text</dataType>
    <developerName>ChatRequestPage</developerName>
    <label>ChatRequestPage</label>
</contextVariables>
<contextVariables>
    <contextVariableMappings>
        <SObjectType>LiveChatTranscript</SObjectType>
        <fieldName>LiveChatTranscript.ShoppingCartValue__c</fieldName>
        <messageType>WebChat</messageType>
    </contextVariableMappings>
    <dataType>Number</dataType>
    <developerName>ShoppingCartValue</developerName>
    <label>ShoppingCartValue</label>
</contextVariables>
```

**b.** (Option 1 of 2: **Apex Code Solution**) Use this code to get you going with the Apex solution. Notice that the data type for the `ShoppingCartValue` variable is `Decimal`. If you need more help, revisit the Apex solution in Greet the Customer with Snap-Ins Chat on page 12.

```
public with sharing class CookbookBot_GetTranscriptVariables {

  public class TranscriptInput {
    @InvocableVariable(required=true)
    public ID routableID;
  }

  public class TranscriptOutput {
    @InvocableVariable(required=true)
    public String sFirstName;

    @InvocableVariable(required=true)
    public String sOrderNumber;

    @InvocableVariable(required=true)
    public String sChatRequestPage;
```

```
    @InvocableVariable(required=true)
    public Decimal nShoppingCartValue;
  }

  @InvocableMethod(label='Get Transcript Variables')
  public static List<TranscriptOutput> getUserName(List<TranscriptInput> transcripts)
 {

    List<TranscriptOutput> outputList = new List<TranscriptOutput>();

    for (TranscriptInput transcript : transcripts) {

      // Query for the transcript record based on the ID
     LiveChatTranscript transcriptRecord = [SELECT Name, FirstName__c, OrderNumber__c,

                                                    ChatRequestPage__c,
ShoppingCartValue__c
                                            FROM LiveChatTranscript
                                            WHERE Id = :transcript.routableID
                                            LIMIT 1];

      TranscriptOutput output = new TranscriptOutput();

      // Store the FirstName field in an output variable
      output.sFirstName = transcriptRecord.FirstName__c;

      // Store the OrderNumber field in an output variable
      output.sOrderNumber = transcriptRecord.OrderNumber__c;

      // Store the the ChatRequestPage field in an output variable
      output.sChatRequestPage = transcriptRecord.ChatRequestPage__c;

      // Store the the ShoppingCartValue field in an output variable
      output.nShoppingCartValue = transcriptRecord.ShoppingCartValue__c;

      // Add the values to the list of outputs
      outputList.add(output);
    }

    return outputList;
  }
}
```

5. Verify that we correctly extracted the values from our new fields.

   Temporarily add the following text to the welcome message: `{!ShoppingCartValue} and {!ChatRequestPage}`. If the changes are correct, you see "600 and Payment" when you run the bot. If not, confirm that your bot has access to the new `LiveChatTranscript` fields and any new Apex code. To learn more, review setting up permissions in Set Up Your First Einstein Bot. Also, verify that the API field names are correct in all your code snippets.

   Now let's look at how we can use this context information to improve the user experience. Our goal is to transfer a customer that's stuck on the payment page to an agent.

6. Add a new "Transfer Required" dialog.

In Optimize Bot Flow with Pre-Chat Data, we optimized the conversation for customers that provided an order number on the pre-chat form. Let's do it again now that we have more context when a customer requests a chat.

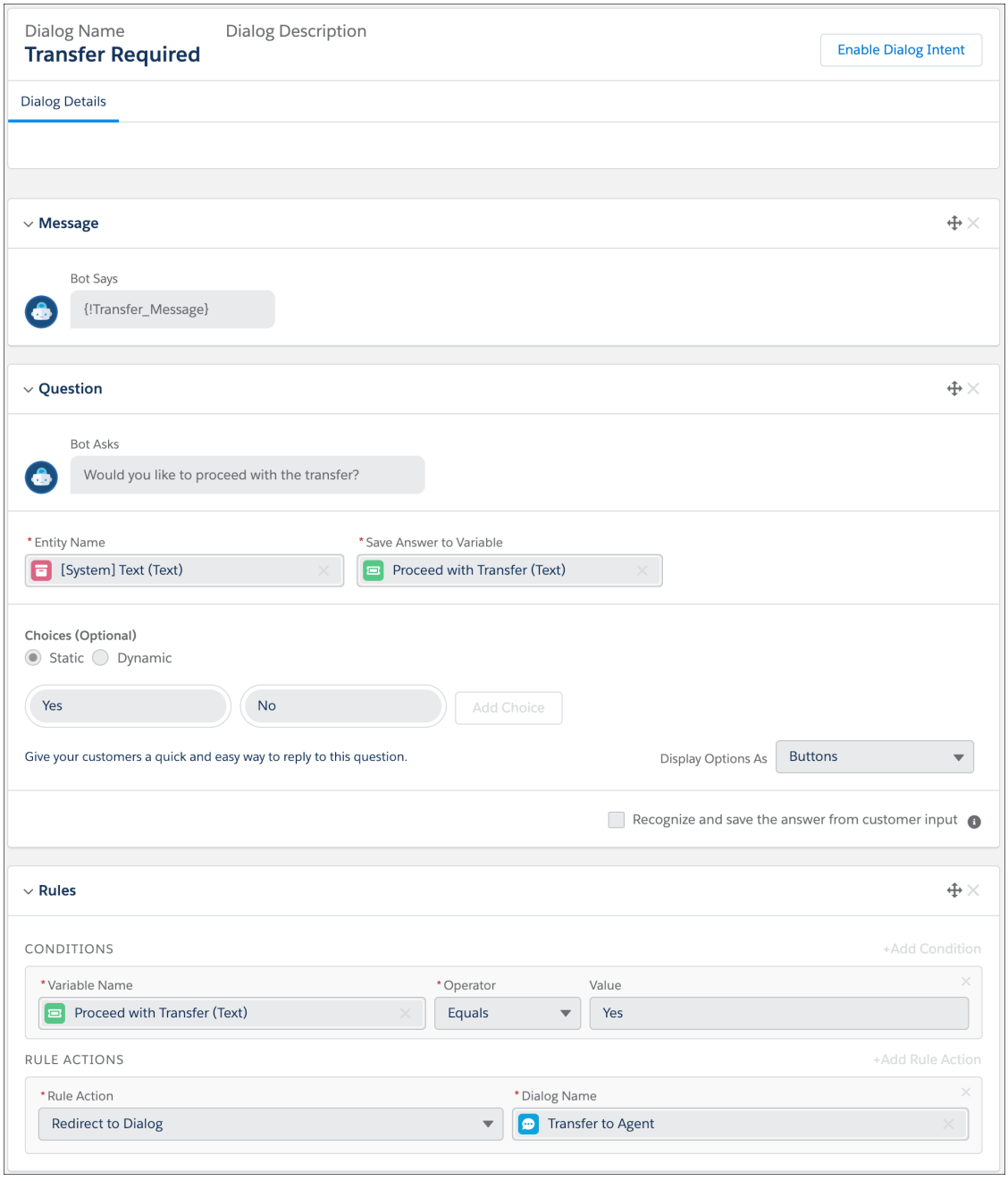
Do we build another dialog called "Welcome customer with a high value shopping cart"? We can. But as we introduce more context variables and rules, the number of dialogs can grow quickly for greeting the customer. Another approach is to build a utility or helper dialog that uses variables to serve different situations and deliver different messages. This approach is more scalable. Let's try it.

Add a "Transfer Required" dialog that handles when the bot recommends agent assistance.

**a.** Set the first message element to include just a new variable, "{!Transfer_Message}".

**b.** Ask if the customer agrees to the transfer.

**c.** If they agree, redirect them to the "Transfer to Agent" dialog.



By configuring the message within a variable, we can use an Apex action, for example, to populate message variables for different situations.

7. Create a "Bot Initialization" Apex action.

   Set up some rules in the original welcome screen that conditionally redirects the customer. The rule follows this logic:

   ```
   Redirect if ChatRequestPage = 'Payment' and ShoppingCartNumber > $500
   ```

   Let's write some Apex code to handle this logic. The code takes two input parameters (`sChatRequestPage` and `sShoppingCartValue`), and returns two `@InvocableVariables`, `bInitialTransferRequired` and `sInitialTransferMessage`, in the wrapper class. The code does two things.

   a. Sets boolean `bInitialTransferRequired` to true when `sChatRequestPage='Payment'` and `nShoppingCartNumber > 500`.

   b. Sets the message in the text variable `sInitialTransferMessage`.

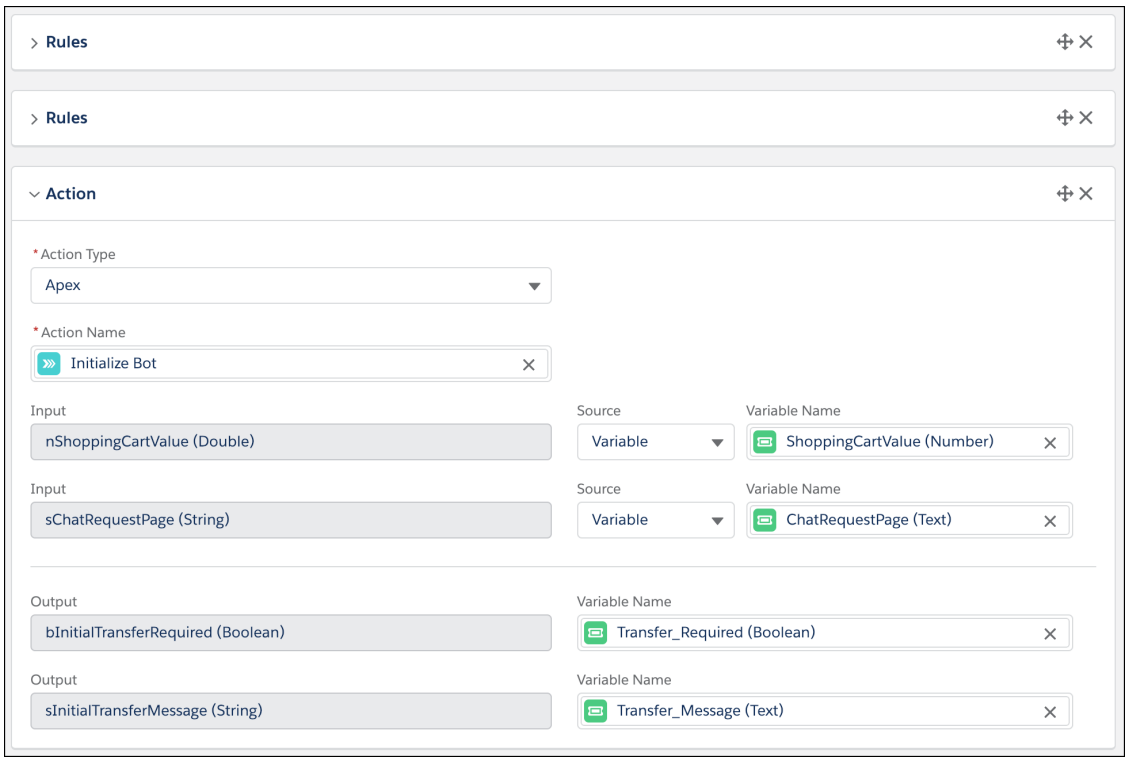
```
public with sharing class CookbookBot_InitializeBot {
   public class InitializationOutput{
        @InvocableVariable(required=true)
        public String sInitialTransferMessage;
        @InvocableVariable(required=true)
        public Boolean bInitialTransferRequired;
   }

   public class InitializationInput{
        @InvocableVariable(required=false)
        public String sChatRequestPage;
        @InvocableVariable(required=false)
        public Decimal nShoppingCartValue;
   }

   @InvocableMethod(label='Initialize Bot')
   public static List<InitializationOutput> initializeBot(List<InitializationInput>
inputParameters)
   {
        // Grab the input variables
        String sChatRequestPage = inputParameters[0].sChatRequestPage;
        Decimal nShoppingCartValue = inputParameters[0].nShoppingCartValue;

        // Initialize output variables
       List<InitializationOutput> outputParameters = new List<InitializationOutput>();

        InitializationOutput outputParameter = new InitializationOutput();

        // Is this a condition where we want to transfer to an agent?
        if (sChatRequestPage=='Payment' && nShoppingCartValue > 500)
        {
            // Set the output variables for a transfer
            outputParameter.bInitialTransferRequired = true;
            outputParameter.sInitialTransferMessage = 'Looks like you may have a
payment-related question. I can find one of our specialists to help you...';
        }
        else
        {
```

```
            // Set the output variables for no transfer required
            outputParameter.bInitialTransferRequired = false;
            outputParameter.sInitialTransferMessage = '';
        }

        // Return result
        outputParameters.add(outputParameter);
        return outputParameters;
    }
}
```

By now, you know how to add an Apex class and make sure that your bot has permission to access the class. If not, see Call an Apex Action.

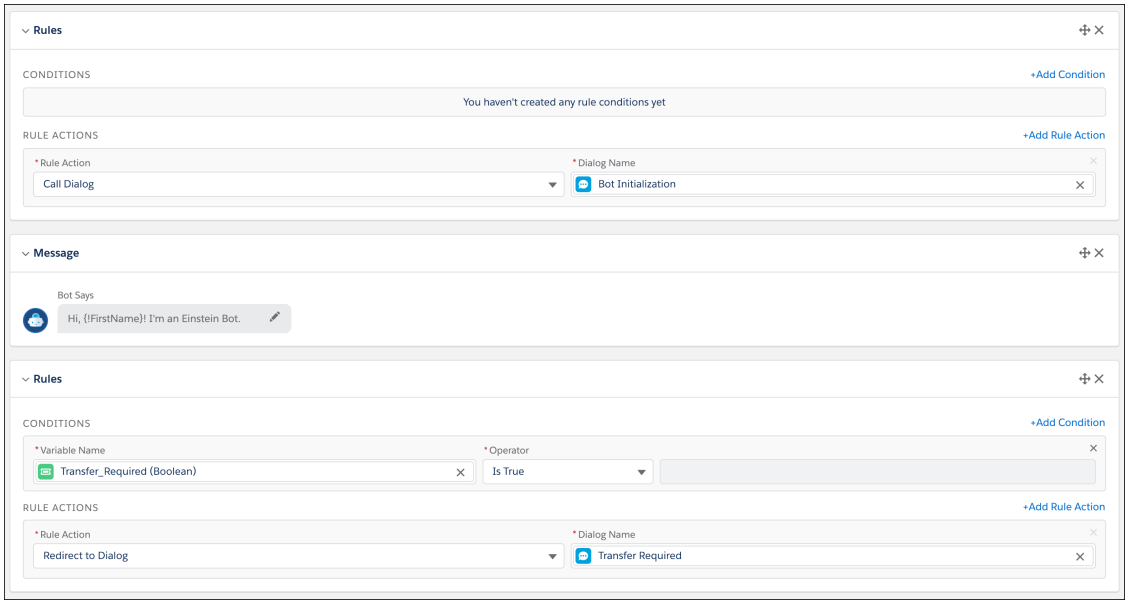
8. Call the "Initialize Bot" Apex action from the "Bot Initialization" dialog.

   In the "Bot Initialization" dialog, *after* we set the `ShoppingCartValue` and `ChatRequestPage` variables with pre-chat form actions, we can call the new Apex action that we built in the previous step. Create two new variables, `Transfer_Required` and `Transfer_Message`, to take the output value of the Apex action, which drives the bot experience.




9. Update the "Welcome" dialog with the new conversation design.

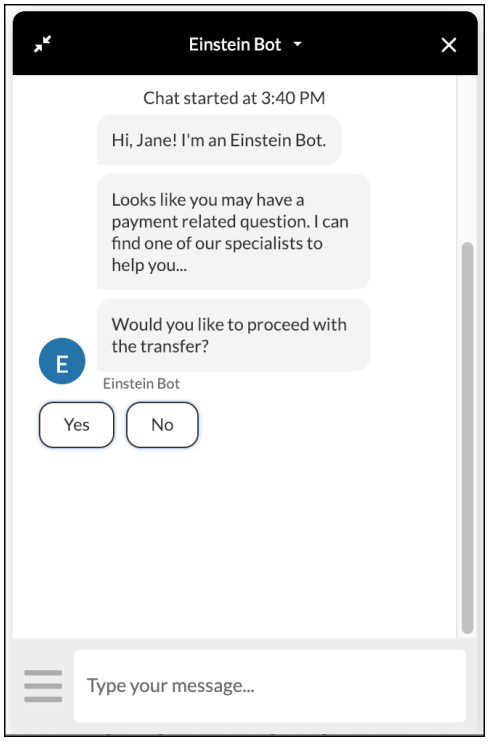
   In the Optimize Bot Flow with Pre-Chat Data recipe, you created a call to the "Bot Initialization" dialog as the first step in the Welcome screen, followed by a greeting message. Let's add a rule to redirect customers to the "Transfer Required" dialog when our initialization action shows that it's warranted.



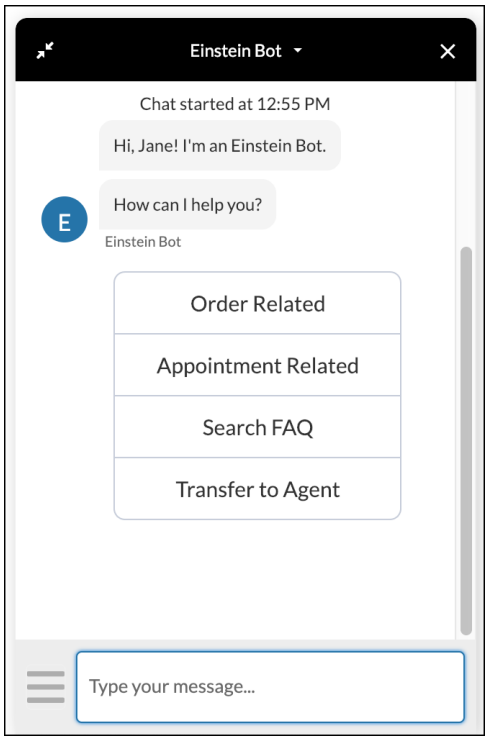

If you look at the available **Rule Action** options, there's a **Call Dialog** action and a **Redirect to Dialog** action. Here's the difference: with **Redirect to Dialog**, the destination dialog takes over the conversation, while **Call Dialog** returns the conversation back to the originating dialog. Hence, we use **Call Dialog** for "Bot Initialization" (because we want the flow to return back to this dialog), and we use **Redirect to Dialog** for the "Transfer Required" dialog (where we want to hop over to the other dialog and not return). Note that when we use **Redirect to Dialog**, whatever configuration we selected in the **Next Dialog** section of the originating dialog is ignored.

**10.** Test!

Let's do two rounds of testing. First, let's use the original chat snippet that had the `ShoppingCartValue` set to 600.



Now let's change the JavaScript that we wrote earlier so that the `ShoppingCartValue` is 300.




**Best Practices and Additional Notes**

- In this use case, one disadvantage is the hard-coded business logic in our "Bot Initialization" Apex action. For example, if you change the `ShoppingCartValue` that triggers a transfer, you must change that value in Apex. A more flexible approach is to create a custom setting or a record in a custom object.
- Can a business user change the bot behavior or are admins responsible for all configuration? It may not be a good idea to have everyone become an administrator and change the bot how they want. Instead, take advantage of custom settings or custom objects that make the bot configuration more data-driven. You can then give access to users so that they can change configuration-related data records. In this way, you give business users flexibility, but in a controlled manner.

# Handle Intent Detection Failures

Confused? We've got your back! In this recipe, we handle intent detection failures more gracefully.

We closed out our beginner section on page 33 with the popular topic of Natural Language Processing (NLP) and intent management on page 71. Let's revisit the topic in this advanced section.

It takes many iterations of learning and training to improve a bot's intent detection accuracy. So how do we do that?

Unlike other parts of solution design, NLP depends largely on confidence level, which means that you can't reliably predict the outcome. Fortunately, the Einstein Bot Builder provides a system dialog, the "Confused" dialog, to help us handle intent detection failures. When we create a new bot, a few dialogs are automatically generated in the initial setup, including the "Welcome", "Main Menu", and "Confused" dialogs. When a bot is confused, such as when it fails to identify the customer intent to start a dialog, or fails to identify the answer variable when asking a question, the bot calls this dialog. By default, the "Confused" dialog shows the message: `Sorry I don't understand.` We can update the "Confused" dialog to provide a different user experience. For starters, you can create your own custom message like `Sorry {!FirstName}, I don't quite understand. Can you try again?`

In addition, Apex code and system variables are available to improve the user experience. In this exercise, we explain two ways to improve intent detection accuracy:

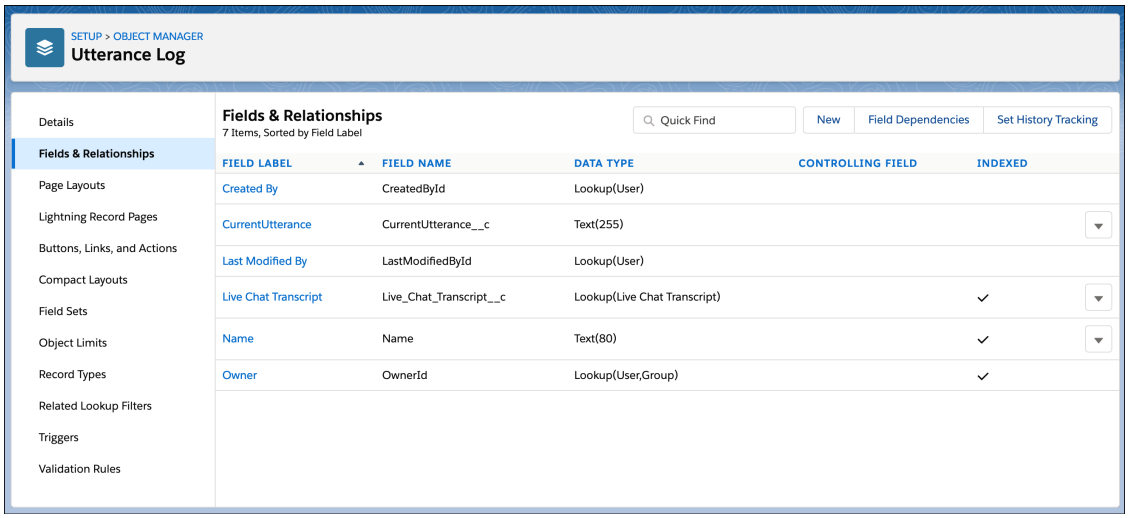
- Log the phrases that the customer enters but that the bot can't identify. We can review these "missing phrase" logs to understand what the customer is asking and add utterances to the bot that improve accuracy.
- If the customer tries several different phrases, we can propose speaking with an agent, and set the next dialog to "Transfer to Agent". However, we want to use advanced logic so that transfers occur only after there are multiple attempts.

Let's get started.

1. Create a custom object for utterances.

   Let's create a log object that stores utterance data. Create a custom object `Utterance Log` (with the API name `Utterance_Log__c`). Fields in this object are:

   - `Name` (Text)
   - `CurrentUtterance__c` (Text)
   - `Live_Chat_Transcript__c` (lookup to LiveChatTranscript)




2. Build an invocable Apex method to log any utterances that confuse us.

```
public with sharing class CookbookBot_LogUtterance {
   public class LogInput{
       @InvocableVariable(required=false)
       public String sCurrentUtterance;
       @InvocableVariable(required=true)
       public String sLiveAgentTranscriptId;
   }

   @InvocableMethod(label='Log Utterance')
   public static void logUtterance(List<LogInput> inputParameters)
   {
       String sCurrentUtterance = inputParameters[0].sCurrentUtterance;
       String sLiveAgentTranscriptId = inputParameters[0].sLiveAgentTranscriptId;

       // Create a new Utterance Log record
       Utterance_Log__c logRecord = new Utterance_Log__c();
```

```
        // Store the utterance
        logRecord.CurrentUtterance__c = sCurrentUtterance;

        logRecord.Live_Chat_Transcript__c = sLiveAgentTranscriptId;

        // Save the log utterance record to our org
        insert logRecord;
    }
}
```

This code takes input parameters—the system variables we pass from the "Confused" dialog—and saves them into a new `Utterance_Log__c` record. With the `Live_Chat_Transcript__c` field, we can associate the utterance with the chat transcript.

3. Build an invocable Apex class and method to handle the transfer required logic.

   This method looks at how many logs we have in the same chat session, and decides whether the bot suggests a transfer. The code uses two rules to determine whether we should redirect to an agent.

   a. There are more than five intent detection failures in the entire session.

   b. The last two failed attempts happened within twenty seconds.

   The output parameters include a `boolean` value for when a transfer is recommended, and a text variable for the transfer message.

```
public with sharing class CookbookBot_HandleIntentDetectFailure {

    public class HandleInput {
        @InvocableVariable(required=false)
        public String sCurrentUtterance;
        @InvocableVariable(required=false)
        public String sLiveChatTranscriptId;
    }

    public class HandleOutput {
        @InvocableVariable(required=true)
        public String sTransferMessage;
        @InvocableVariable(required=true)
        public Boolean bTransferRequired;
    }

    @InvocableMethod(label='Handle Intent Detection Failure')
    public static List<handleOutput> handleIntentDetectionFailure(List<HandleInput>
inputParameters) {
        String sCurrentUtterance = inputParameters[0].sCurrentUtterance;
        String sLiveChatTranscriptId = inputParameters[0].sLiveChatTranscriptId;

        // Create default output values
        List<HandleOutput> outputParameters = new List<HandleOutput>();
        HandleOutput outputParameter = new HandleOutput();
        outputParameter.bTransferRequired = false;
        outputParameter.sTransferMessage = '';

        // Find undetected utterances from this session
        List<Utterance_Log__c> undetectedUtterances = [SELECT CurrentUtterance__c,
CreatedDate
                                                       FROM Utterance_Log__c
```

```
                                                    WHERE Live_Chat_Transcript__c =
:sLiveChatTranscriptId
                                                    ORDER BY CreatedDate Desc];

        // Have we had more than two undetected utterances?
        if (undetectedUtterances.size() >= 2)
        {
            // Have we had more than five?
            if (undetectedUtterances.size() > 5)
            {
                // If so, then automatically suggest a transfer...
                outputParameter.bTransferRequired = true;
                outputParameter.sTransferMessage = 'Let me find one of our specialists
to help you.';
            }
            // If less than five...
            else
            {
                // Grab the most recent two utterances
                Utterance_Log__c thisUtterance = undetectedUtterances[0];
                Utterance_Log__c lastUtterance = undetectedUtterances[1];

                // If they happened within the past 20 seconds...
              if (thisUtterance.CreatedDate < lastUtterance.CreatedDate.addSeconds(20))

                {
                    // Then suggest a transfer...
                    outputParameter.bTransferRequired = true;
                    outputParameter.sTransferMessage =
                      'I\'m having troubles understanding what you need. Let me find a
specialist.';
                }
            }
        }
        outputParameters.add(outputParameter);
        return outputParameters;
    }
}
```

4. Update the "Confused" dialog.

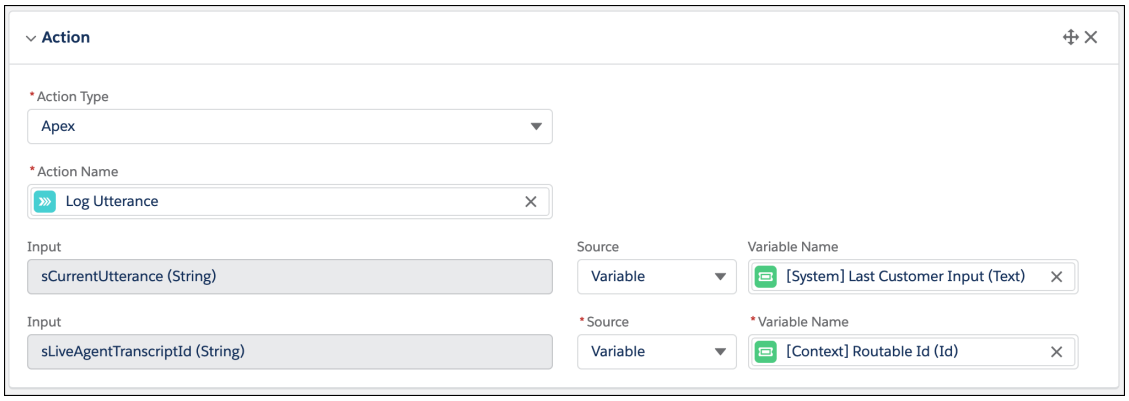
   This dialog should call the two Apex actions that we just created.

   Important: As mentioned in Set Up Your First Einstein Bot, you need to give the bot permission to access these Apex classes. From Setup, use the Quick Find box to find **Permission Sets**. Select the **sfdc.chatbot.service.permset** permission set. From the permission set options, select **Apex Class Access**. Now you can edit the access settings so that your new class is accessible to the bot. Always make sure that the bot has permission to access objects and classes as required.

   Here are the steps for this dialog.

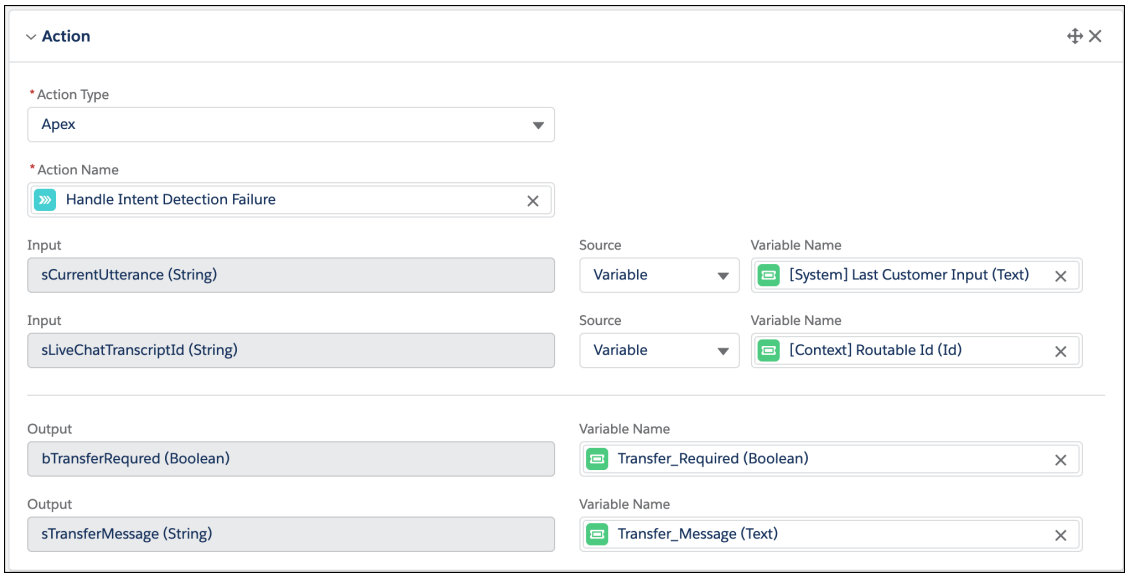
   a. Keep the original message, such as `Sorry, I didn't understand that.`


**b.** Run our `Log Utterance` Apex method.



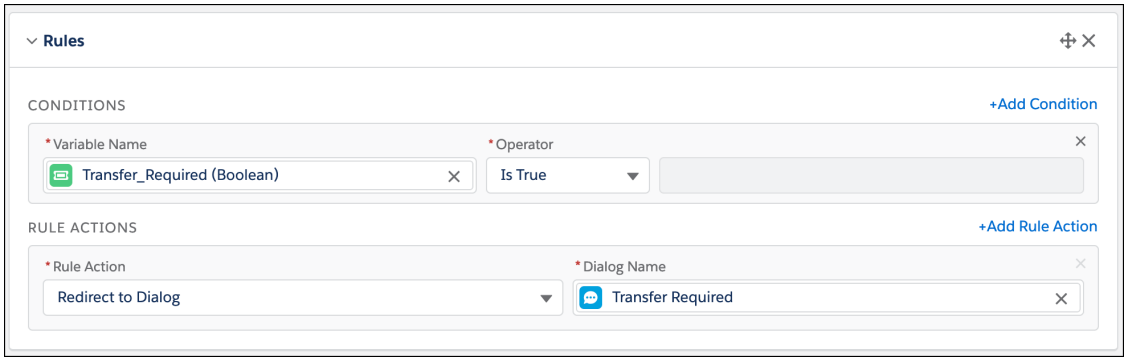

We use a built-in system variable called `Last Customer Input`, which passes in what the customer last typed. We also use the context variable `Routable Id` to pass in the `Live Chat Transcript` ID. To review the list of context variables, see the Context Variable Table on page 12 from Greet the Customer with Embedded Service Chat.

**c.** After logging the utterance, run our `Handle Intent Detection Failure` Apex method.




Again, we pass in `Last Customer Input` and `Routable Id`. As for the outputs, we store these values in `Transfer Required` and `Transfer Message`. We're using the same variables as in the Get Context Info from the Web recipe so that we can leverage the transfer utility dialog that we already created.

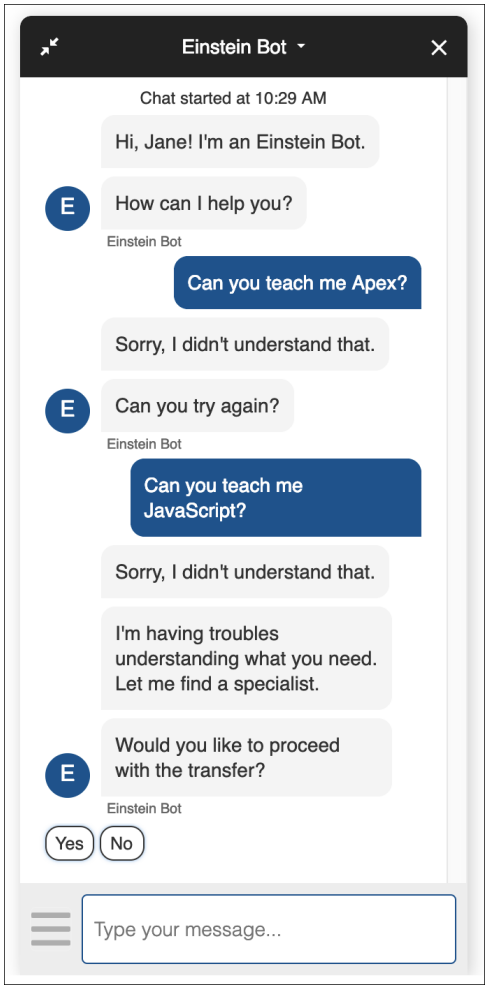
**d.** After recording the output from the previous Apex method, create a rule to check the `Transfer_Required` variable. If true, redirect to the "Transfer Required" dialog.




**e.** If the `Transfer_Required` variable is false, respond with a generic message, such as `Can you try again?`


**5.** Test!

OK, let's see what the user experience looks like.




You can also inspect the `Utterance_Log` records to see what you captured from the failures.

**Best Practices and Additional Notes**

- The sample code in this example is for demonstration purposes and is NOT production ready. Depending on the chat volume, a log object can grow quickly. In addition, you should take privacy considerations into account when storing these utterances. Consider data storage, performance, and a data archive plan.
- We separated the `logUtterance()` method from `handleIntentDetectionFailure()` method to create a generic approach that works from any dialog. For example, to see how customer phrases a certain intent, call the logging method from that dialog. In that situation, it may help to update the Apex method to pass in information about the current dialog to store alongside the utterance.

# Mobile App Bot Recipes

Use these recipes to build native mobile apps that take advantage of bots.

Did you know that you can add a chatbot to your iOS or Android app with very little effort? Yep. It's easy to do using the Embedded Service SDK for Mobile Apps. After grabbing a few pieces of config info from your org, your app can give customers access to chat, and your chatbot, within minutes. Check out the iOS and Android recipes in this section to learn more.

Before working through these recipes, go through the recipes in Get Started with Bots.

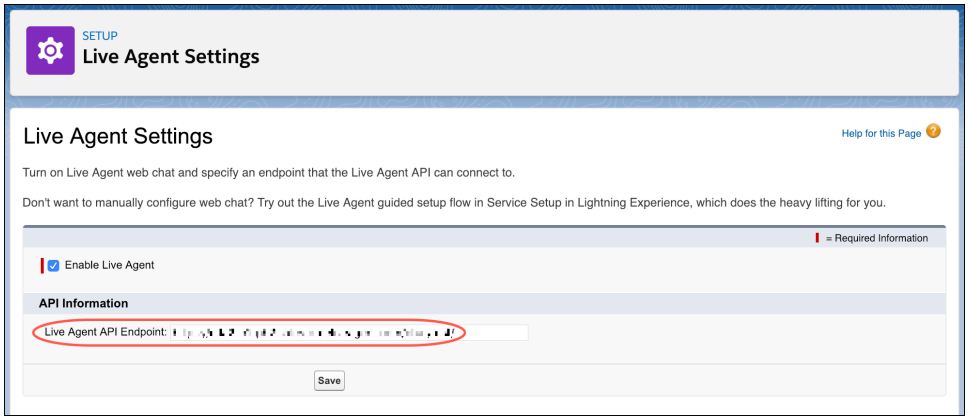
# Get Config Info Before Building a Mobile App

This section describes how to get configuration information from your org that you'll need before building a chatbot-enabled mobile app.

Before we get into the mobile app development process, we need to get four pieces of information from our chat implementation: the hostname for the chat endpoint, the org ID, the deployment ID, and the button ID. If you need help getting this information, follow these steps.

1. Get the hostname for the chat endpoint that your organization has been assigned.

   To get this value, from Setup, search for **Chat Settings** and copy the hostname from the **API Endpoint**.




   Be sure not to include the protocol or the path. For instance, if your API Endpoint is:

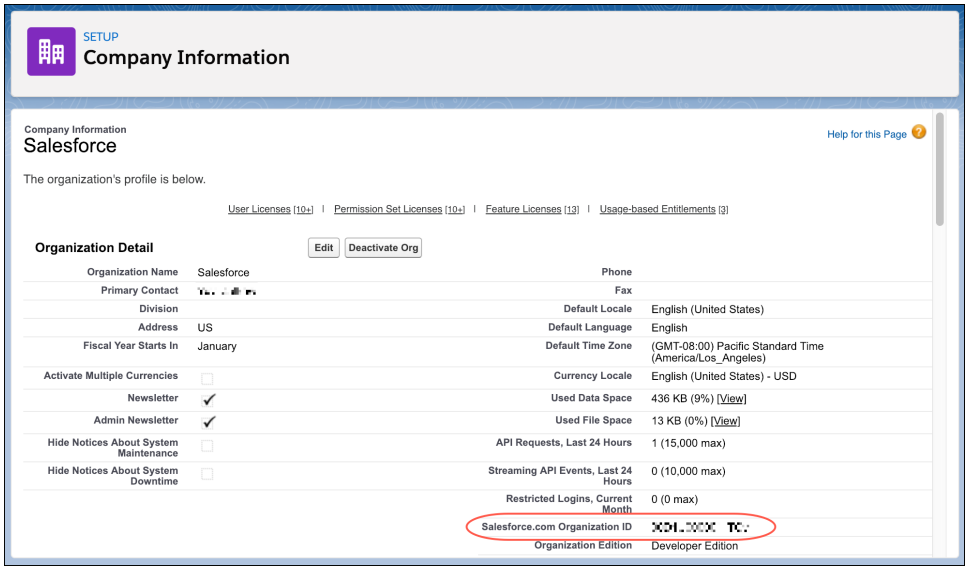
```
https://d.gla5.gus.salesforce.com/chat/rest/
```

   Your endpoint hostname is:

```
d.gla5.gus.salesforce.com
```

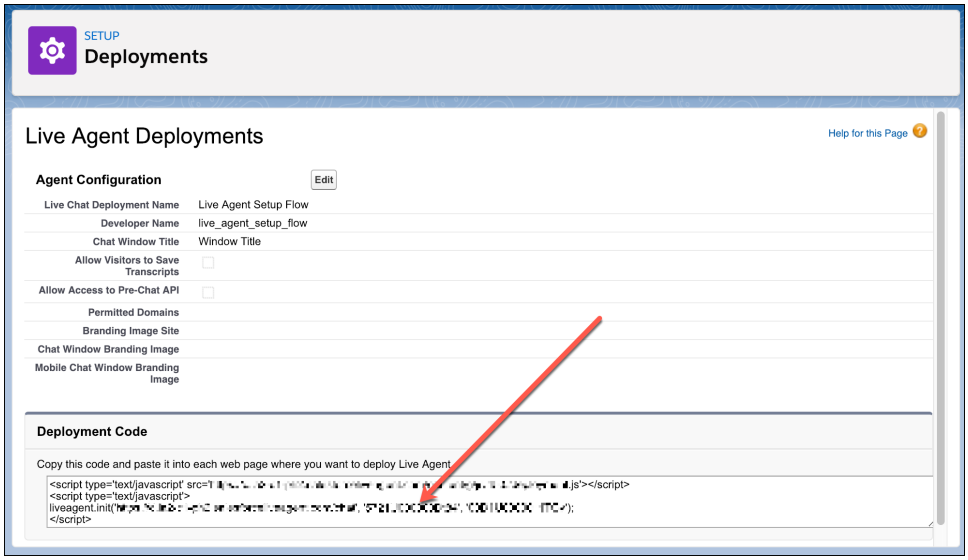
2. Get the Salesforce org ID.

   To get this value, from Setup, search for **Company Information** and copy the **Salesforce Organization ID**.




3. Get the unique ID of your Chat deployment.

   To get this value, from Setup, select **Chat** > **Deployments**. The script at the bottom of the page contains a call to the `liveagent.init` function with the **pod**, the **deploymentId**, and **orgId** as arguments. Copy the **deploymentId** value.




For instance, if the deployment code contains the following information:

```
<script type='text/javascript'
       src='https://d.gla3.gus.salesforce.com/content/g/js/44.0/deployment.js'></script>
<script type='text/javascript'>
liveagent.init('https://d.gla5.gus.salesforce.com/chat', '573B00000005KXz',
'00DB00000003Rxz');
</script>
```

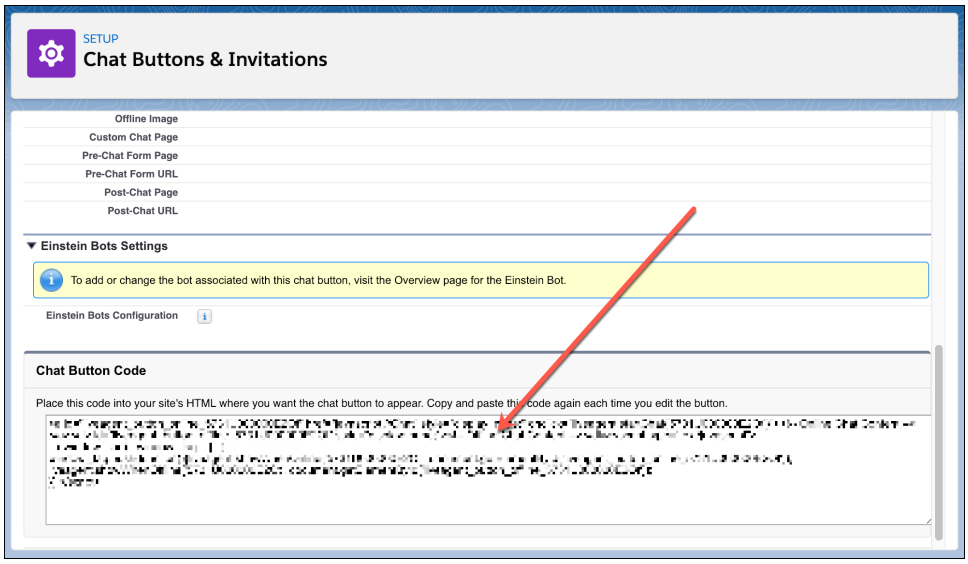
The deployment ID value is:

```
573B00000005KXz
```

Be sure not to use the org ID value (which is also in this deployment code) for the deployment ID.

4. Get the unique button ID for your chat configuration.

   To get this value, from Setup, search for **Chat Buttons** and select **Chat Buttons & Invitations**. Copy the `id` for the button from the JavaScript snippet.




   For instance, if your chat button code contains the following information:

```
<a id="liveagent_button_online_575C00000004h3m"
   href="javascript://Chat"
   style="display: none;"
   onclick="liveagent.startChat('575C00000004h3m')">
   <!-- Online Chat Content -->
</a>
<div id="liveagent_button_offline_575C00000004h3m"
     style="display: none;">
     <!-- Offline Chat Content -->
</div>
<script type="text/javascript">
  if (!window._laq) { window._laq = []; }
  window._laq.push(function() { liveagent.showWhenOnline('575C00000004h3m',
    document.getElementById('liveagent_button_online_575C00000004h3m'));
    liveagent.showWhenOffline('575C00000004h3m',
    document.getElementById('liveagent_button_offline_575C00000004h3m'));
  });
</script>
```

   The button ID value is:

```
575C00000004h3m
```

   Be sure to omit the `liveagent_button_online_` text from the ID when using it in the SDK.

OK. That's it! You're ready to build a mobile app. Which platform do you want to work on first?

- Add a Chatbot to Your iOS App
- Add a Chatbot to Your Android App

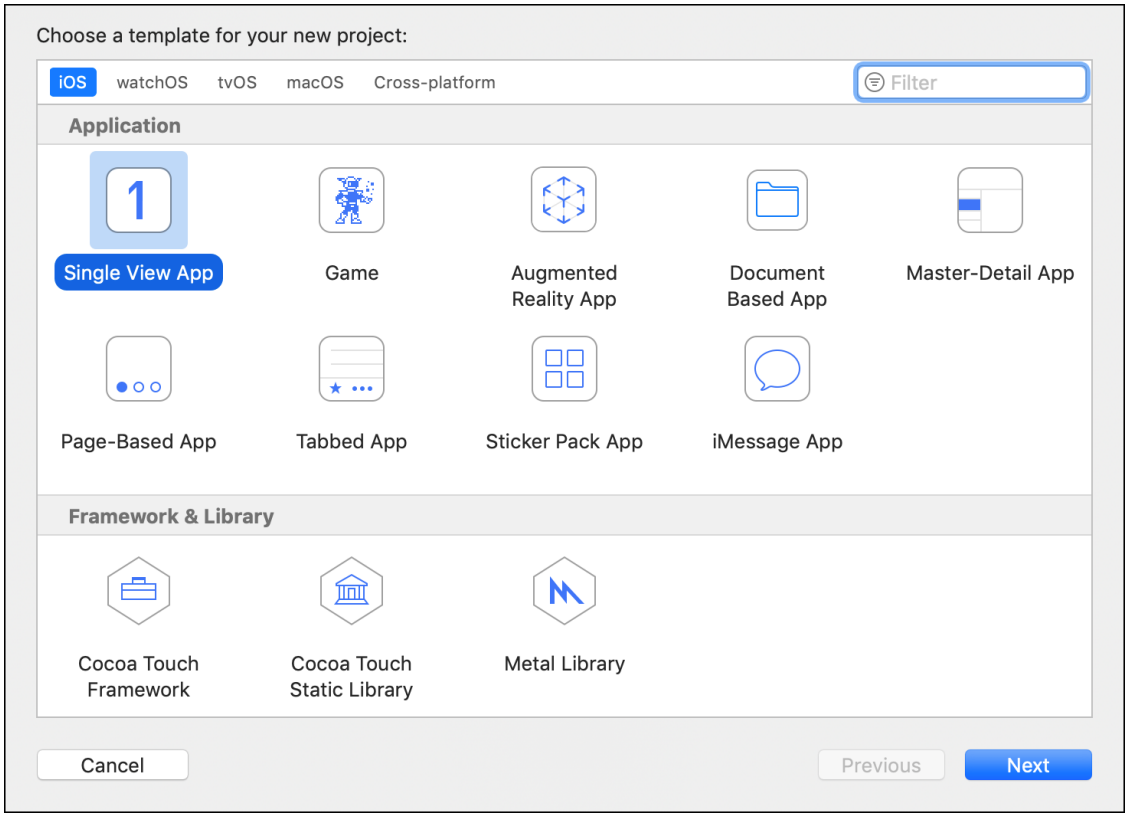
# Add a Chatbot to Your iOS App

In this recipe, we learn how to use the Service SDK for iOS to get a chatbot up and running in our iOS app.

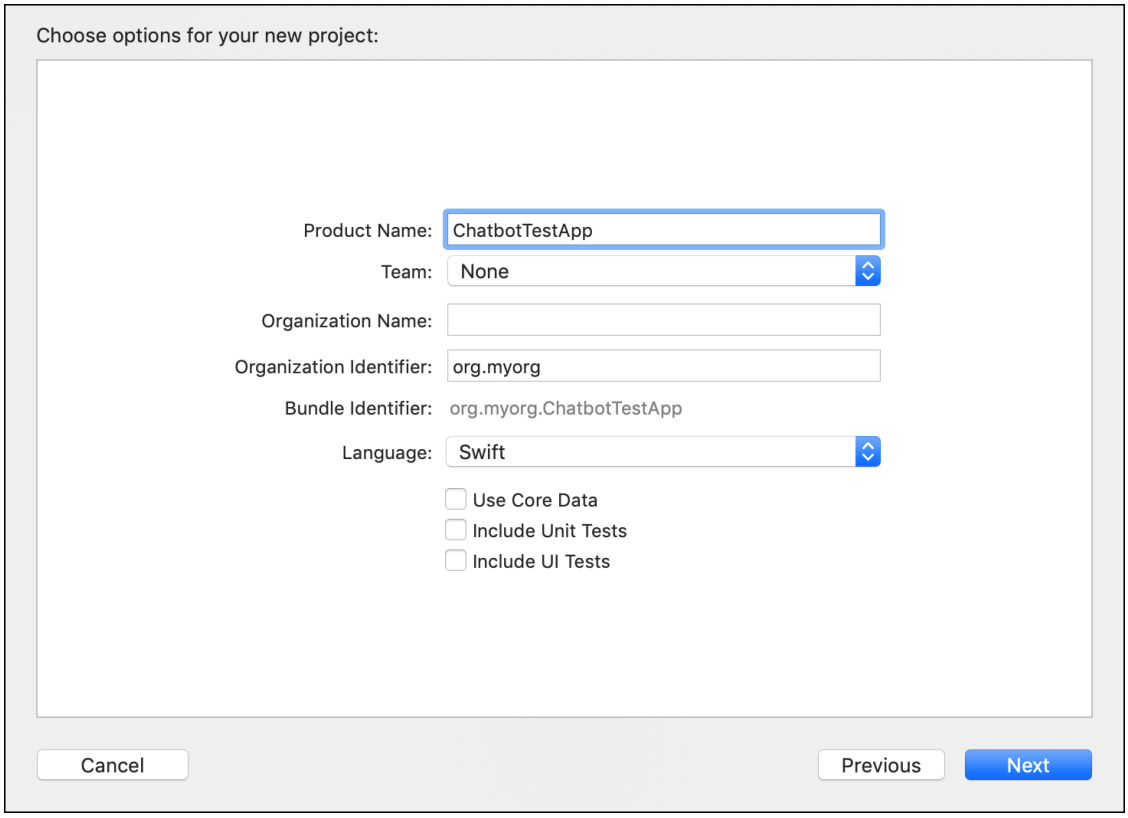
Even though this recipe walks you through the whole process of adding a chatbot to a mobile app, we assume that you've built at least one iOS app before starting this section. If you've never tried your hand at iOS development, run through a beginner tutorial to get a feeling for how things work and then come back. We are using the Swift language to write this app, but you could also write it in Objective-C.

1. Before adding a chatbot to our mobile app, we need to grab some configuration information from your org. If you haven't done so, follow the steps found in Get Config Info Before Building a Mobile App.
2. Build a basic app.

   If you have an app, you can skip this step, but if not, let's create a new **Single View App** in Xcode.




Give the app a name and an organization identifier. Specify Swift for the language and if you're making a simple test app, you can uncheck the boxes to use core data and to build test code.



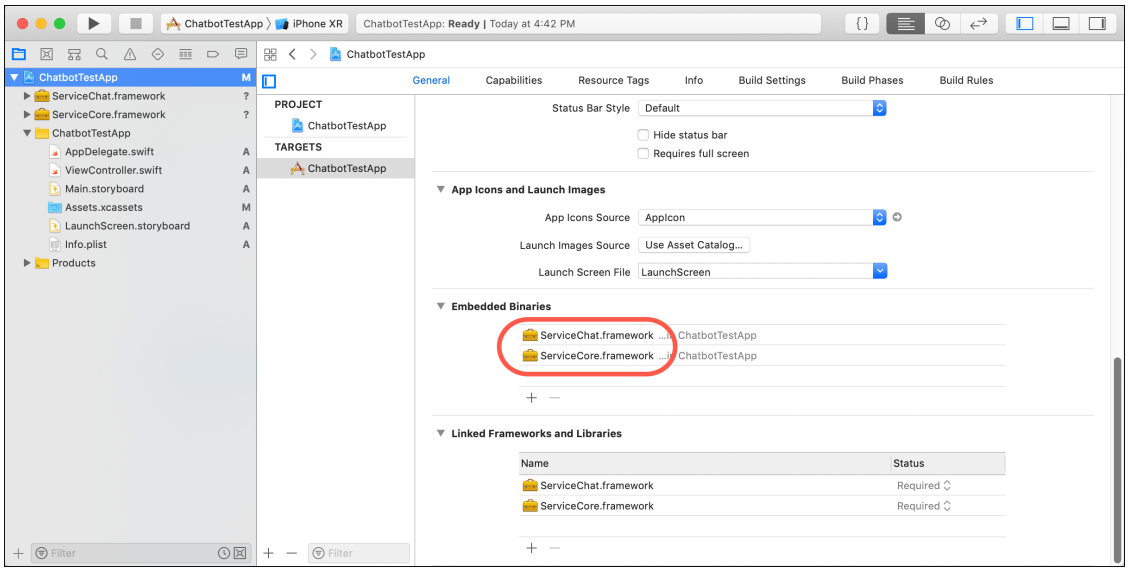

3. Install the SDK.

   Follow these instructions to install the SDK. If you want the complete details, refer to Install the Service SDK for iOS.

   a. Download the SDK framework files from the Service SDK download page.

      The zip file contains a few different frameworks. The two frameworks that we need are **ServiceCore** and **ServiceChat**.

   b. From the **Embedded Binaries** section of your Xcode project settings, add these two framework files.



c. Because the chat experience with an agent could involve transferring an image, you need to provide descriptions for why the app might access the device's camera and photo library.

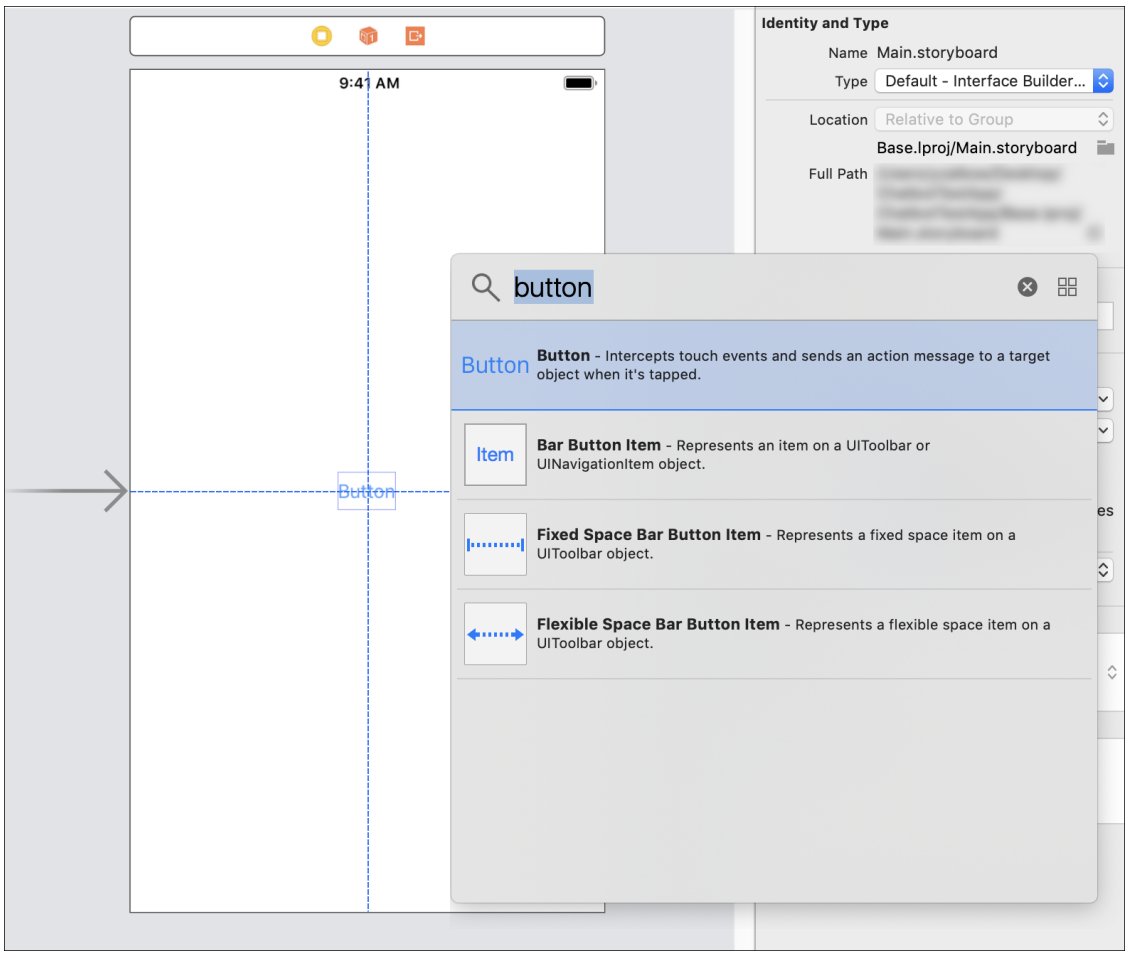
Add string values for "Privacy - Camera Usage Description" and "Privacy - Photo Library Usage Description" in your `Info.plist` file. To learn more about these properties, see Cocoa Keys in Apple's reference documentation.

Sample values for these keys:

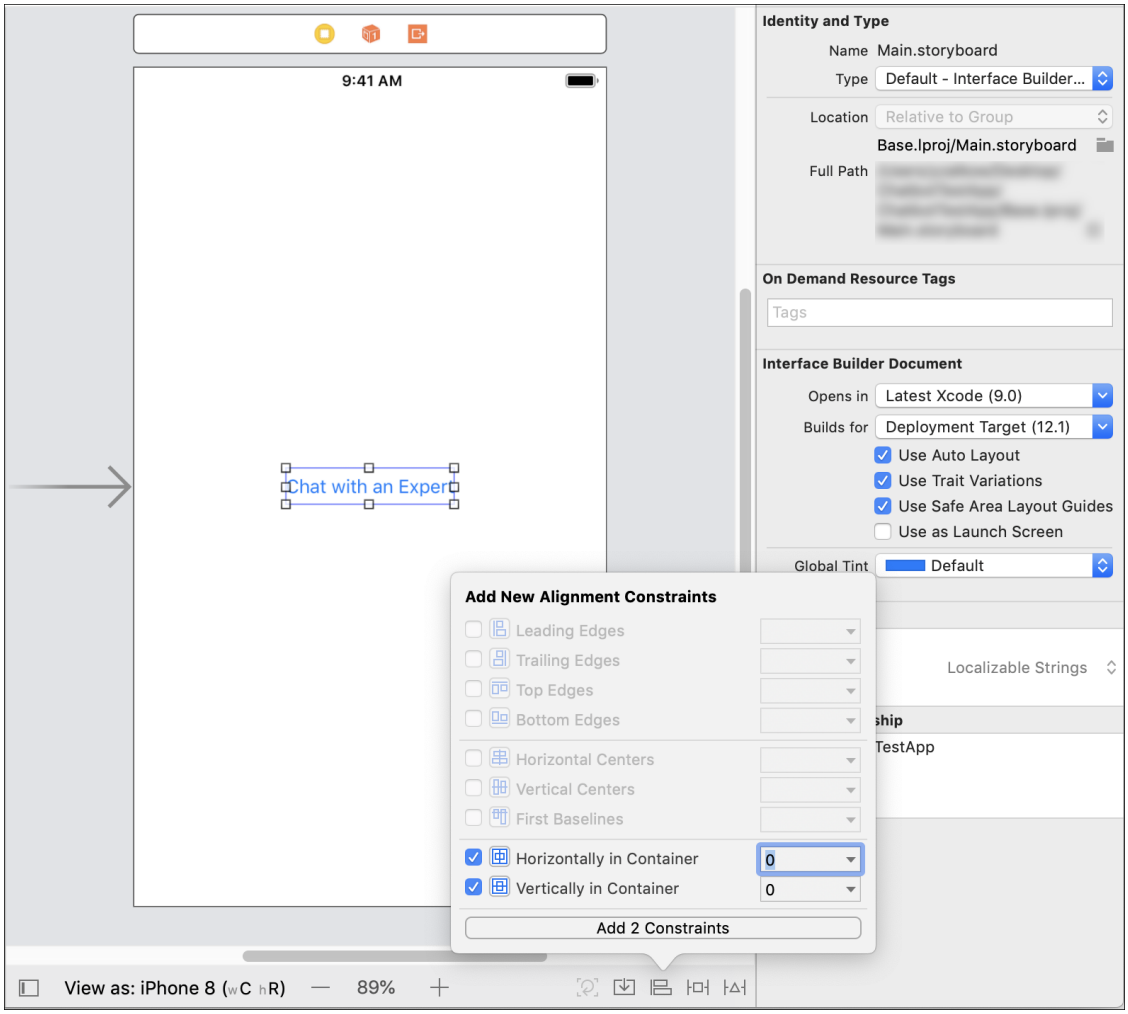
```
<key>NSCameraUsageDescription</key>
<string>Used when sending an image to an agent.</string>
<key>NSPhotoLibraryUsageDescription</key>
<string>Used when sending an image to an agent.</string>
```

4. Add a chat button to your app.

Go to your `Main.storyboard` and search for *`button`* in the **Library** and drag a button to the view.



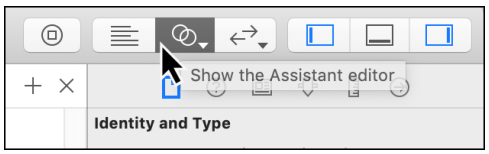

Name the button `Chat with an Expert`. Align the button so that it shows on the view no matter how the device is oriented. Once simple way to do this is by centering the button horizontally and vertically.



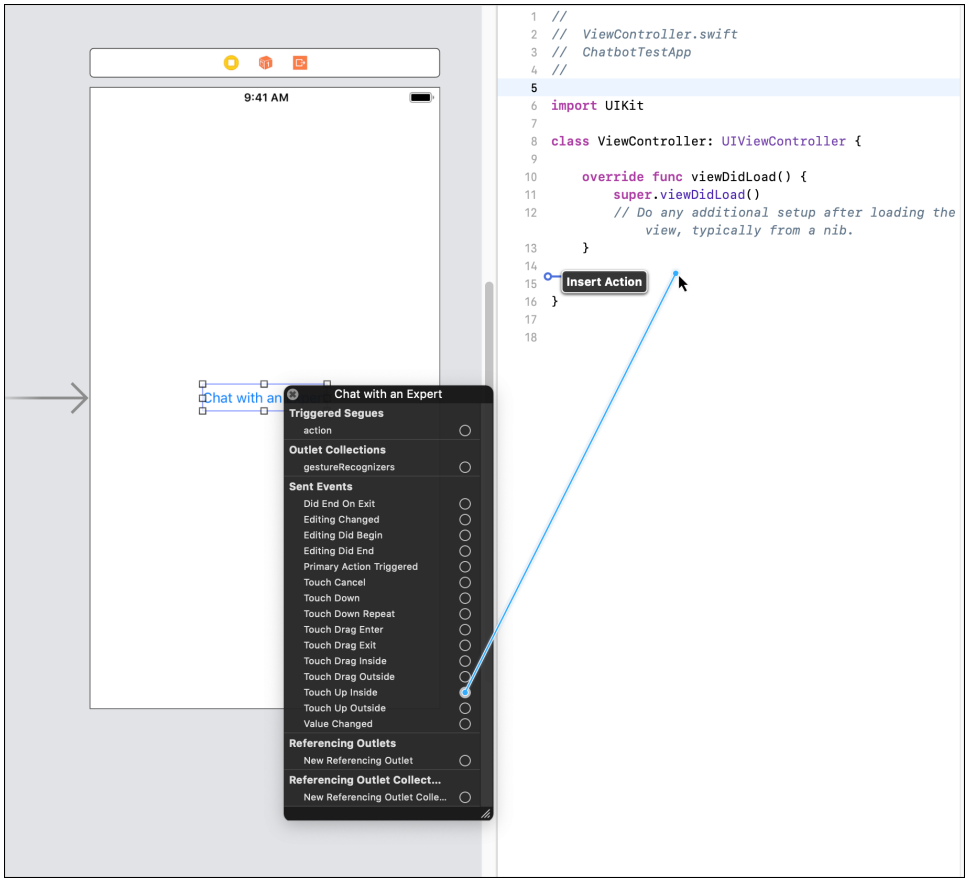

We may not have the most beautiful app in the world, but at least we have a working app with a button. If you try to run the app, you'll notice that the button doesn't do anything yet.

5. Connect the button to a method in the `ViewController` class.

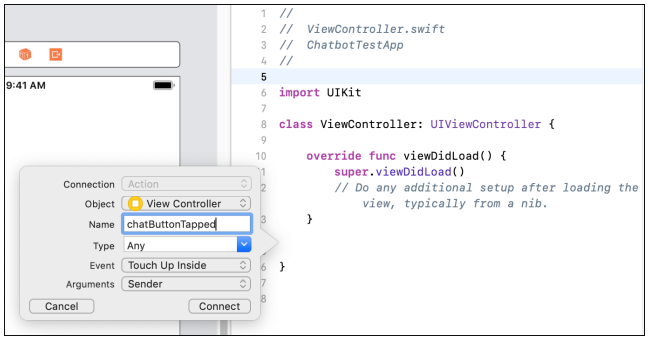
   There are many ways to map a method in our code to the button action. One way is to turn on the assistant editor while viewing the storyboard.



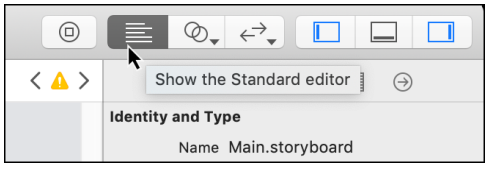

This action displays a split screen with the storyboard on the left and the `ViewController` code on the right. We want to right click (or "secondary click" in Apple parlance) the button in the storyboard. Then drag from the **Touch Up Inside** event into our `ViewController` class on the right.



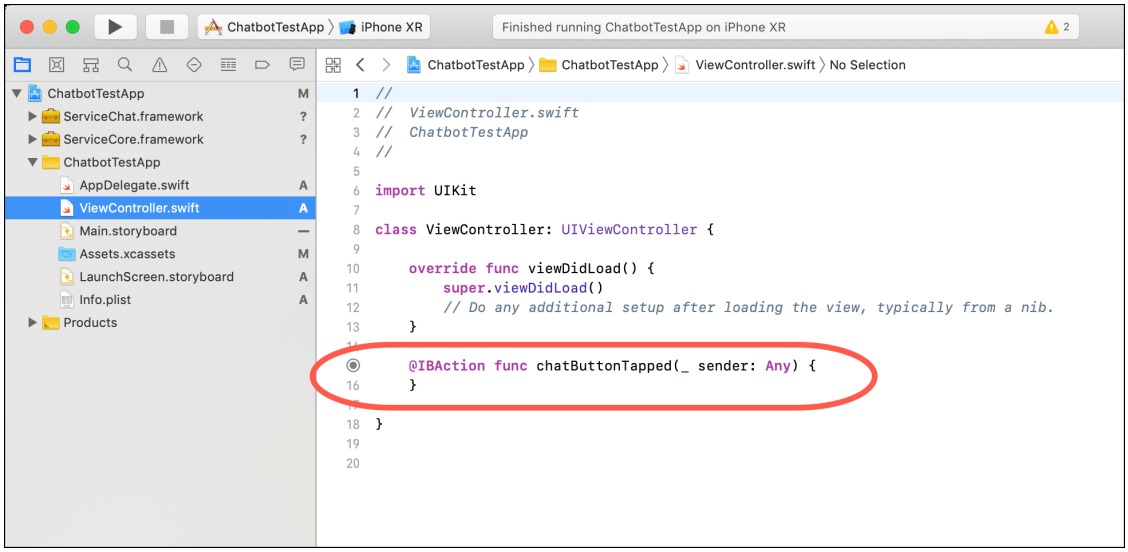

We have a chance to name our new action method. Let's name it `chatButtonTapped`.




Great, we have wired up the method to the button. Go back to the standard editor.




Select the `ViewController.swift` class from the file browser and check out our new method.




6. Configure the SDK.

   Before we get the button to start a chatbot session, let's get things configured. First, we need to import the Service SDK into our class so we can use the SDK. At the top of the `ViewController.swift` file, next to the other import statements, import `ServiceChat`:

```
import ServiceChat
```

   At the top of the class, right after the class is defined, let's add four constant values for our configuration information. If you don't have this information, go back to Get Config Info Before Building a Mobile App.

```
let POD = "YOUR_POD_NAME"
let ORG_ID = "YOUR_ORG_ID"
let DEPLOYMENT_ID = "YOUR_DEPLOYMENT_ID"
let BUTTON_ID = "YOUR_BUTTON_ID"
```

   Below those values, let's create a variable that we'll use to specify configuration information.

```
var chatConfig: SCSChatConfiguration?
```

   Let's configure our SDK in the `viewDidLoad` method, which is stubbed out for you. In a more production-ready app, you probably wouldn't want to configure the SDK in the view controller (which can be instantiated and destroyed multiple times). Typically, you only want to configure the SDK one time in the application's lifecycle, but this will work fine for this example app.

```
override func viewDidLoad() {
  super.viewDidLoad()

  chatConfig = SCSChatConfiguration(liveAgentPod: POD,
                                    orgId: ORG_ID,
                                    deploymentId: DEPLOYMENT_ID,
                                    buttonId: BUTTON_ID)
}
```

   Great, we have our app configured. We're almost ready to test it out.

7. Start the chat session.

The final step is to start a chat session, which we can do from the button handler we created earlier. Let's call the `showChat` method.

```
@IBAction func chatButtonTapped(_ sender: Any) {
        ServiceCloud.shared().chatUI.showChat(with: chatConfig!)
}
```

Since we're throwing a lot of new information your way, let's break down exactly what's going on:

**ServiceCloud**
This class is the entry point into the Service SDK.

**shared()**
This static method gets the singleton instance of the `ServiceCloud` class.

**chatUI**
This property gets the `SCSChatInterface` object, which is the entry point into the chat feature of the Service SDK.

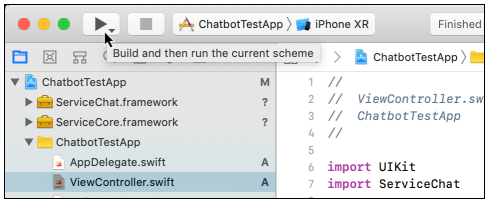
**showChat**
This method shows the chat UI. This method has a few variants. We're using the simplest method that only requires a chat configuration object (`SCSChatConfiguration`). There is another variant that is used if you want to display a pre-chat form.

If you've followed all these instructions, your `ViewController.swift` file looks like this:

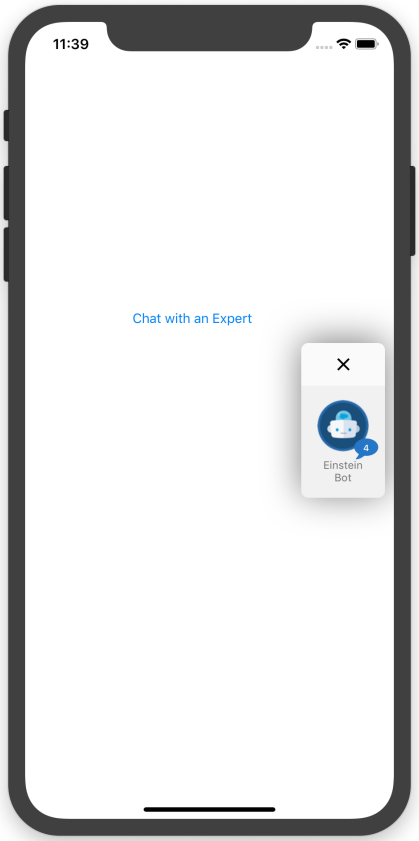
```
import UIKit
import ServiceChat

class ViewController: UIViewController {

    let POD = "YOUR_POD_NAME"
    let ORG_ID = "YOUR_ORG_ID"
    let DEPLOYMENT_ID = "YOUR_DEPLOYMENT_ID"
    let BUTTON_ID = "YOUR_BUTTON_ID"

    var chatConfig: SCSChatConfiguration?

    override func viewDidLoad() {
        super.viewDidLoad()

        chatConfig = SCSChatConfiguration(liveAgentPod: POD,
                                          orgId: ORG_ID,
                                          deploymentId: DEPLOYMENT_ID,
                                          buttonId: BUTTON_ID)
    }

    @IBAction func chatButtonTapped(_ sender: Any) {
        ServiceCloud.shared().chatUI.showChat(with: chatConfig!)
    }
}
```

8. Test!

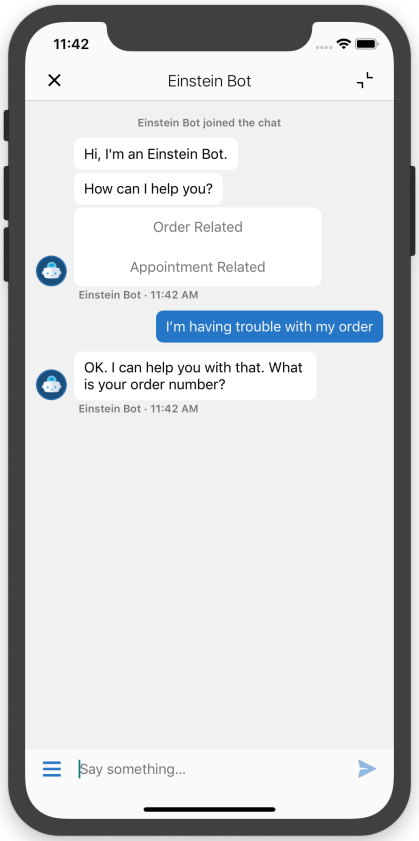
Let's see what happens when we build and run the app in the simulator.




When the app launches, click the chat button. A thumbnail of the chatbot session appears on the right side of the screen.




When you tap that thumbnail, you can have a complete interaction with your chatbot, including menus and buttons and any other smarts you've added to your bot along the way.




We've built the simplest possible chat app for iOS. Not a bad start. But there's a lot we haven't covered. For example:

**Pre-chat Forms**
: As with the web, you can display a pre-chat form before starting a session.

**State Changes and Error Handling**
: We haven't discussed how to detect state changes and how to handle error conditions.

**Branding and Customization**
: You can fully brand the look and feel of the chat UI and customize the colors, strings, and fonts.

**Roll Your Own UI**
: You can even build your own UI and simply use the Chat Core API to communicate with the underlying chat functionality.

To get the full story on using this SDK, check out all the chat-related topics in the Service SDK developer guide: Using Chat with the Service SDK for iOS.

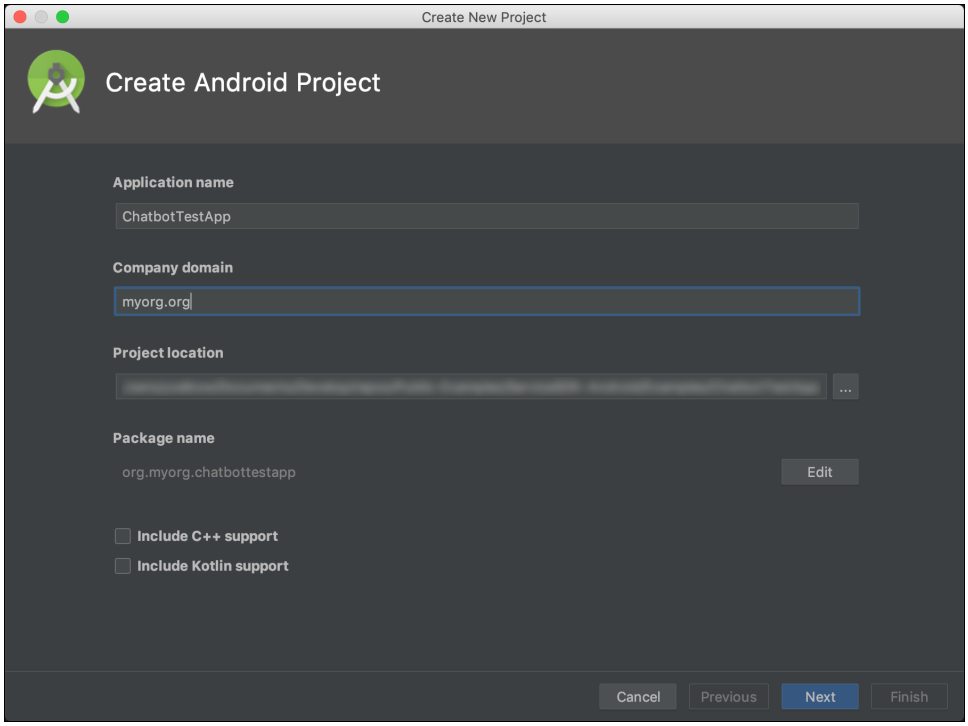
# Add a Chatbot to Your Android App

In this recipe, we learn how to use the Service SDK for Android to get a chatbot up and running in our Android app.

Even though this recipe walks you through the whole process of adding a chatbot to a mobile app, we assume that you've built at least one Android app before starting this section. If you've never tried your hand at Android development, run through a beginner tutorial to get a feeling for how things work and then come back. We are using the Java language to write this app, but you could also write it in Kotlin.

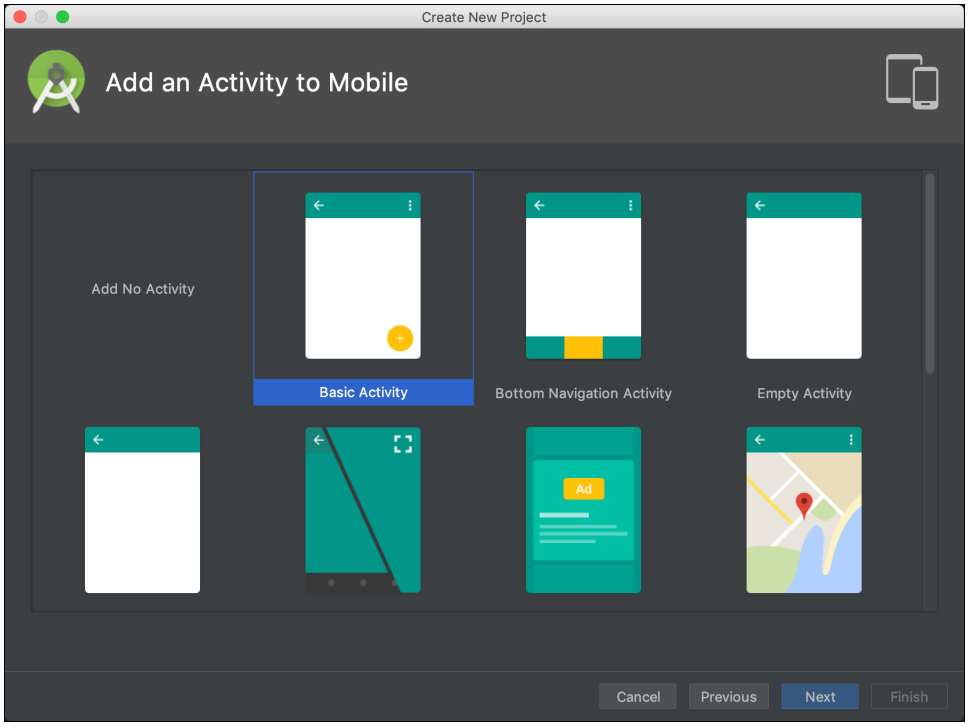
1. Before adding a chatbot to our mobile app, we need to grab some configuration information from your org. If you haven't done so, follow the steps found in Get Config Info Before Building a Mobile App.
2. Build a basic app.

   It's time to launch Android Studio and create a new project. Give your application a name and then click **Next**.




   For a target device, use **Phone and Tablet**, and pick a version of Android that is supported in Requirements for the Service SDK for Android.

   Let's select the **Basic Activity** so that we already get a button we can use for chat.




Use the default values when configuring this activity and then select **Finish**.

3. Install the SDK.

   Follow these instructions to install the SDK. If you want the complete details, refer to Install the Service SDK for Android.

   a. Add the SDK maven repository.

      In your project's `build.gradle` file, add the following maven repository: `https://s3.amazonaws.com/salesforcesos.com/android/maven/release`.

```
allprojects {
    repositories {
        google()
        jcenter()
        maven {
            url 'https://s3.amazonaws.com/salesforcesos.com/android/maven/release'
        }
    }
}
```

   b. Add the chat feature of the Service SDK.

      In your app's `build.gradle` file, add the following dependency: `com.salesforce.service:chat-ui`.

```
implementation 'com.salesforce.service:chat-ui:4.3.2'
```

   c. Add network permissions for your app.

      In your project's `AndroidManifest.xml` file, declare the following permissions:

```
<uses-permission android:name="android.permission.INTERNET"/>
<uses-permission android:name="android.permission.ACCESS_NETWORK_STATE"/>
```

4. (Optional) Change the button icon in the app.

   The basic activity already has a button included with the app, but let's change this button to show a chat bubble. From your drawable resources, open `activity_main.xml` and view it as text. Change the floating action button icon from `ic_dialog_email` to `sym_action_chat`. Your `FloatingActionButton` widget should look like this:

```
<android.support.design.widget.FloatingActionButton
  android:id="@+id/fab"
  android:layout_width="wrap_content"
  android:layout_height="wrap_content"
  android:layout_gravity="bottom|end"
  android:layout_margin="@dimen/fab_margin"
  app:srcCompat="@android:drawable/sym_action_chat" />
```

5. Configure the SDK.

   Before we get the button to start a chatbot session, let's get things configured. Open the `MainActivity` class. At the top of the class, right after the class is defined, let's add four constant values for our configuration information. If you don't already have this information, go back to Get Config Info Before Building a Mobile App.

```
public static final String ORG_ID = "YOUR_ORG_ID";
public static final String DEPLOYMENT_ID = "YOUR_DEPLOYMENT_ID";
public static final String BUTTON_ID = "YOUR_BUTTON_ID";
public static final String LIVE_AGENT_POD = "YOUR_LAC_ORG_URL";
```

   Below those values, let's build our chat configuration object.

```
ChatConfiguration chatConfiguration =
            new ChatConfiguration.Builder(ORG_ID, BUTTON_ID,
                    DEPLOYMENT_ID, LIVE_AGENT_POD)
                    .build();
```

6. Start the chat session.

   From within the `FloatingActionButton` click listener of the `onCreate` method, we want to configure and start a chat session. The code to configure and start a chat session looks like this:

```
// Configure a chat UI object
ChatUI.configure(ChatUIConfiguration.create(chatConfiguration))
  .createClient(getApplicationContext())
  .onResult(new Async.ResultHandler<ChatUIClient>() {
      @Override public void handleResult (Async<?> operation,
        ChatUIClient chatUIClient) {

            // Once configured, let's start a chat session
            chatUIClient.startChatSession(MainActivity.this);
      }
});
```

   So, putting it all together, the entire `onCreate` method looks like this:

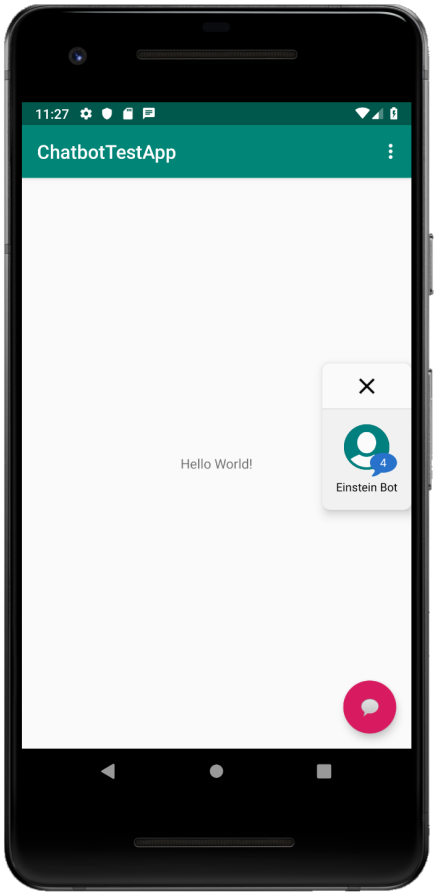
```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
    Toolbar toolbar = (Toolbar) findViewById(R.id.toolbar);
    setSupportActionBar(toolbar);
```

```
    FloatingActionButton fab = (FloatingActionButton) findViewById(R.id.fab);
    fab.setOnClickListener(new View.OnClickListener() {
        @Override
        public void onClick(View view) {

            // Configure a chat UI object
            ChatUI.configure(ChatUIConfiguration.create(chatConfiguration))
              .createClient(getApplicationContext())
              .onResult(new Async.ResultHandler<ChatUIClient>() {
                  @Override public void handleResult (Async<?> operation,
                    ChatUIClient chatUIClient) {

                      // Once configured, let's start a chat session
                      chatUIClient.startChatSession(MainActivity.this);
                  }
            });
        }
    });
}
```

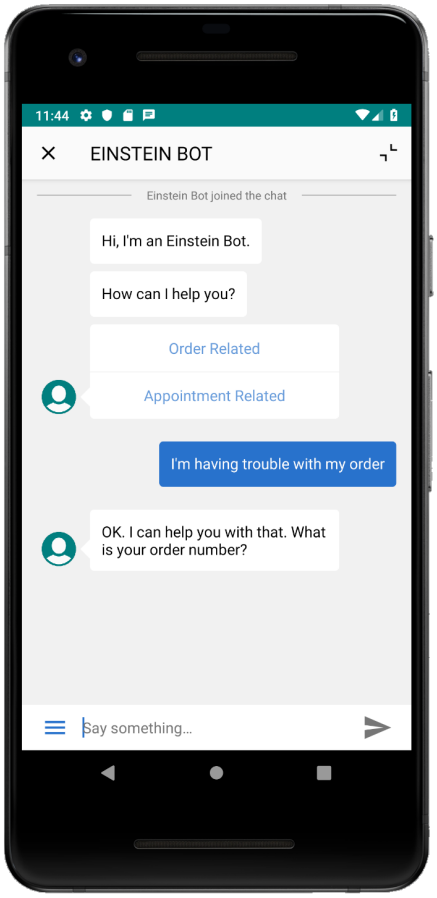
7. Test!

   Let's see what happens when we build and run the app in the emulator.

   When the app launches, click the chat button. A thumbnail of the chatbot session appears on the right side of the screen.



When you tap that thumbnail, you can have a complete interaction with your chatbot, including menus and buttons and any other smarts you've added to your bot along the way.




As with the iOS recipe on page 128, we built an basic app here. There are many ways to fine-tune this experience. For example:

**Pre-chat Forms**
As with the web, you can display a pre-chat form before starting a session.

**State Changes and Error Handling**
We haven't discussed how to detect state changes and how to handle error conditions.

**Branding and Customization**
You can fully brand the look and feel of the chat UI and customize the colors, strings, and fonts.

**Roll Your Own UI**
You can even build your own UI and simply use the Chat Core API to communicate with the underlying chat functionality.

To get the full story on using this SDK, check out all the chat-related topics in the Service SDK developer guide: Using Chat with the Service SDK for Android.

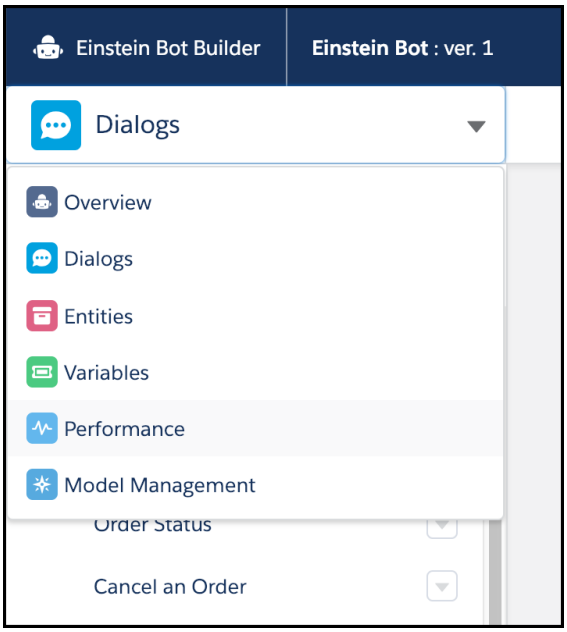
# Troubleshooting Your Bot

Get some tips on troubleshooting your bot when you run into issues.

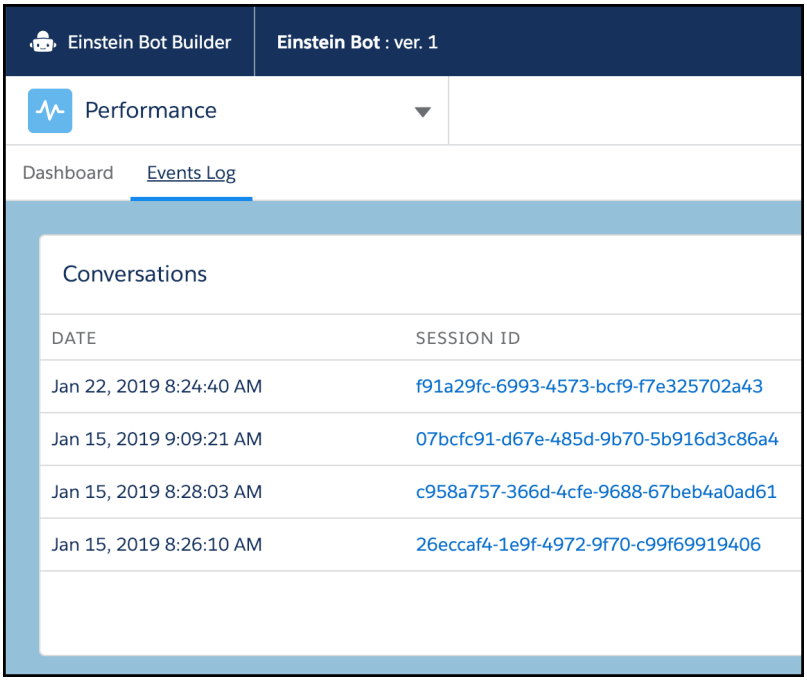
You'll probably run into a few hiccups along the way when you start building up your bot. This topic covers a few troubleshooting tactics.

# Events Log

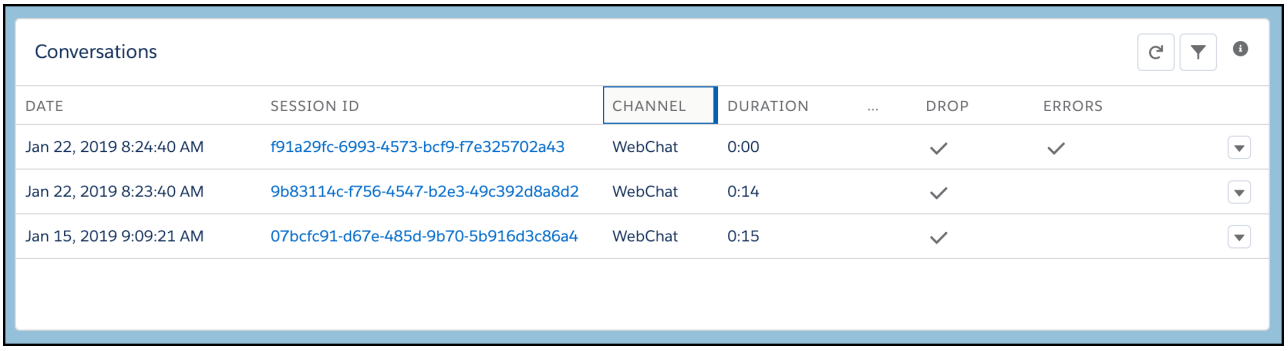
The first step in debugging a bot is to use the **Events Log**. You can get to the **Events Log** by selecting the **Performance** tab from the Einstein Bot Builder.




After selecting the **Performance** tab, select the **Events Log** subtab.

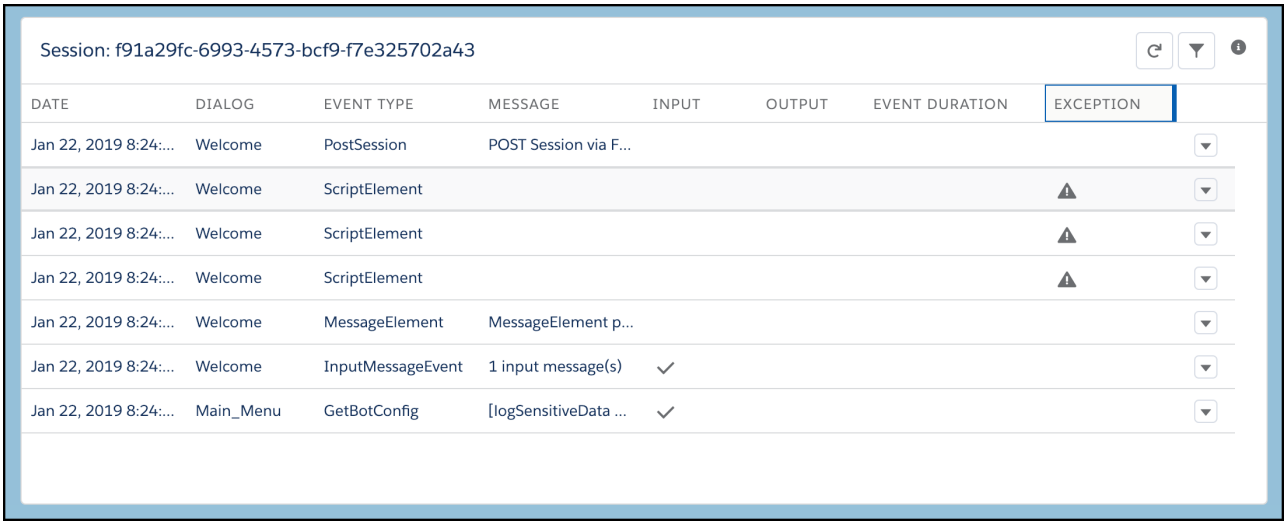



From the **Events Log** subtab, you can view a history of all your bot conversations.



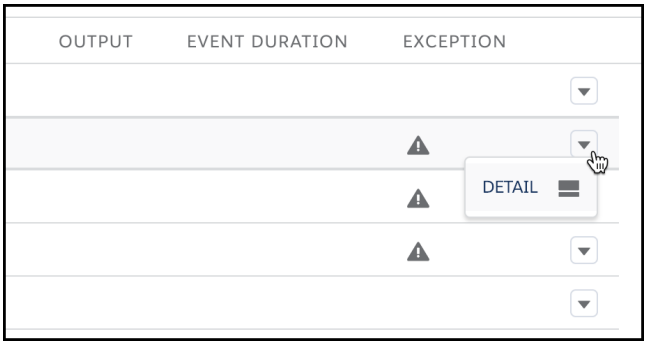

If you have too many conversations listed, you can filter the list using the filter button at the top right. The **Errors** column is useful when looking for problematic conversations.

When you want to investigate a particular conversation, click the session ID to drill into an event log.

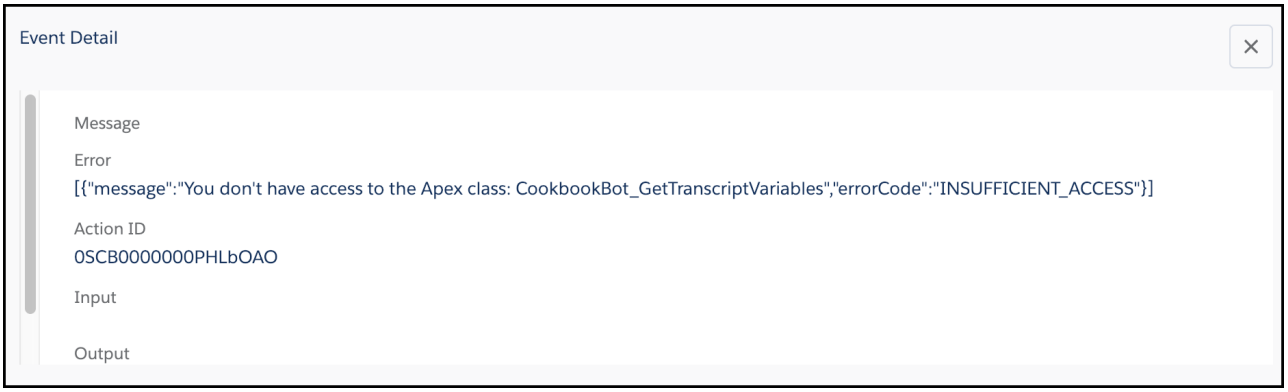



Note: It takes a few minutes after a conversation for the event log to populate with all recent events.

To get details about a specific event, click the disclosure triangle on the right side of a row and select **DETAIL**.



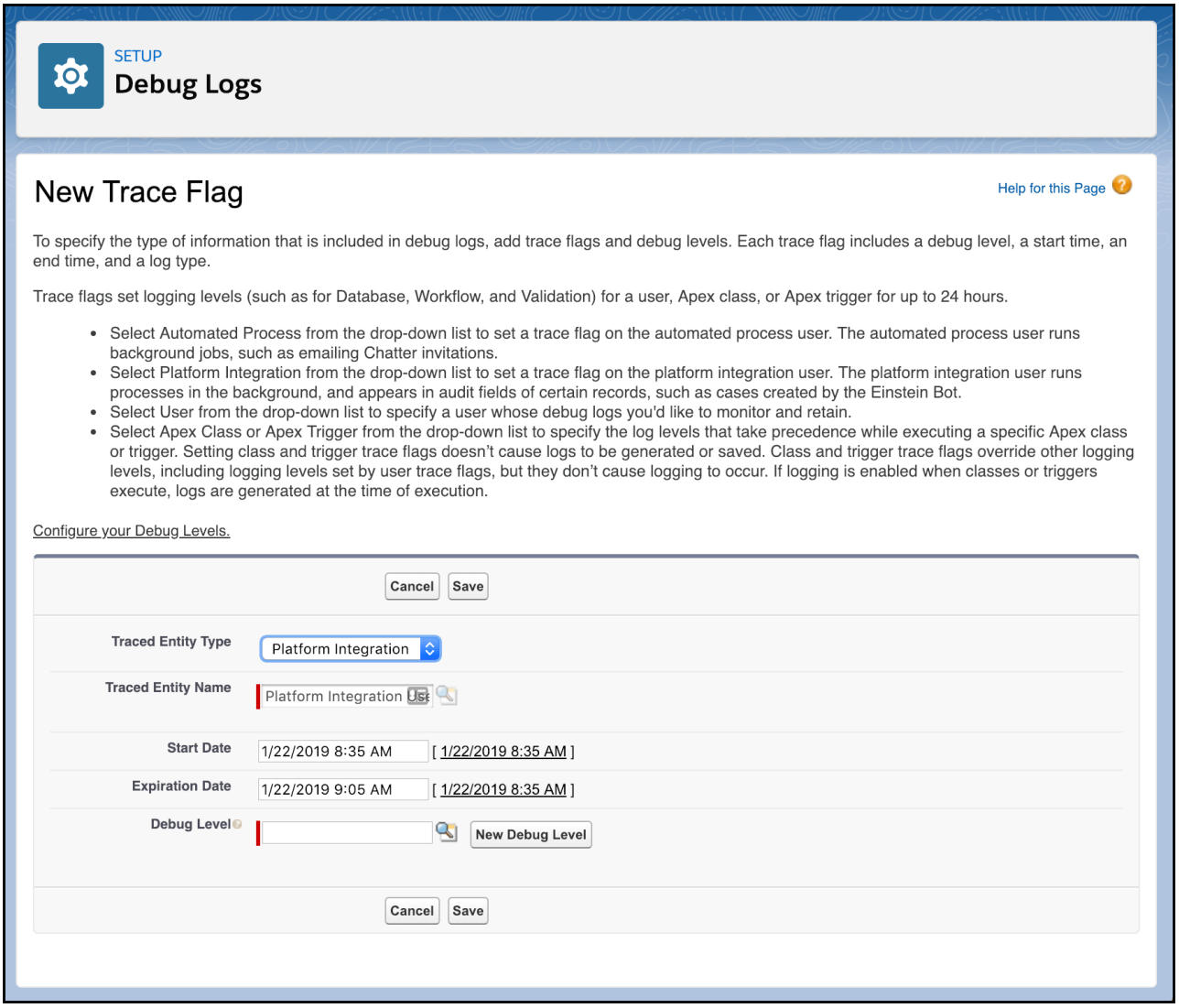

The detailed information gives you more insight into the event.




In this case, the error was caused because the bot didn't have permissions to the Apex class it was trying to access. This error can easily be resolved based on the guidance found in the Call an Apex Action section: From Setup, use the Quick Find box to find **Permission Sets**. Select the **sfdc.chatbot.service.permset** permission set. From the permission set options, select **Apex Class Access**. Now you can edit the access settings so that your new class is accessible to the bot. Always make sure that the bot has permission to access objects and classes as required.

## Debug Logs

If you didn't get any insight from the **Events Log**, you can also look into the org's debug log. From Setup, search for **Debug Logs**. Create a User Trace Flag and specify **Platform Integration** as the **Trace Entity Type**.




You can now inspect logging information from your bots. To learn more about Salesforce debug logs, see Monitor Debug Logs from Salesforce Help.