Oracle® Cloud

Creating Intelligent Bots with Oracle Mobile Cloud Enterprise

18.2.3
E80652-01
May 2018

ORACLE

Oracle Cloud Creating Intelligent Bots with Oracle Mobile Cloud Enterprise, 18.2.3

E80652-01

Primary Author: John Bassett

Contents

Preface

Audience viii
Documentation Accessibility viii
Conventions viii

1 Overview

What Are Intelligent Bots? 1-1
Why Use Oracle Bots? 1-1

2 The Basics

Bot Concepts 2-1
How Do I Create a Bot? 2-1

3 Quick Reference

Managing Your Bots 3-1

4 The Sample Bots

5 Intents

Create an Intent 5-1
- Add Entities to Intents 5-4
- Import Intents from a CSV File 5-6
- Export Intents to a CSV File 5-7
- Intent Training and Testing 5-7
  - Test Sets 5-7
  - The Intent Tester 5-7
- Which Training Model Should I Use? 5-13
- Guidelines for Building Your Training Corpus 5-14

Reference Intents in the Dialog Flow 5-16

6 Entities

Built-In Entities 6-1
Simple Entities 6-2
Complex Entities 6-2
Custom Entities 6-5
Custom Entity Types 6-5
Create Entities 6-6
Import Value List Entities from a CSV File 6-10
Export Value List Entities to a CSV File 6-10

7 The Dialog Flow Definition

The Dialog Flow Structure 7-1
How Do I Write Dialog Flows in OBotML? 7-3
Dialog Flow Syntax 7-5
Flow Navigation 7-8
Conifguring the Dialog Flow for Unexpected Actions 7-8
Accessing Variable Values with Apache FreeMarker FTL 7-10
User-Scoped Variables 7-12
Defining User-Scoped Variables 7-14
Getting the User Context 7-14
Test the Dialog Flow 7-15

8 Localization

Resource Bundles 8-1
Create Resource Bundle 8-1
Reference Resource Bundles in the Dialog Flow 8-3
Resource Bundle Entry Resolution 8-5
Autotranslation 8-5
Enable Autotranslation 8-6

9 Components

The Custom Component Service 9-1
Create a Service 9-2
How Do Custom Components Work? 9-3
The Component Service 9-4
The Shell 9-5

The Registry 9-5
Component Modules 9-6
The SDK 9-7
The Message Model 9-8
How Do I Implement the Component Service in OMCe? 9-8

10 Channels

Running Your Bot on Facebook Messenger 10-1
Step 1: Set Up Facebook Messenger 10-2
Step 2: Add the Facebook Keys 10-3
Step 3: Configure the Facebook Messenger Webhook 10-5
Step 4: Enable the Facebook Channel 10-7
Step 5: Testing Your Bot on Facebook Messenger 10-8
Running Your Bot on Other Messaging Services 10-9
Running Your Bot Within Client Messaging Apps and Web Pages 10-17
Bots Client SDKs 10-18
Bots Client SDK for Android 10-18
Bots Client SDK for iOS 10-23
Bots Client SDK for JavaScript 10-31

11 Quality Reports

How Do I Use the Data Quality Reports? 11-1
Utterances 11-1
Run an Utterance Quality Report 11-2
Suggestions 11-4
History 11-5
How Do I Run a History Report? 11-6

12 Bots Analytics

Adding Analytics to the PizzaBot Sample Bot 12-1
Setting up the PizzaBot Analytics Application 12-1
Setting up the PizzaBot Custom Component 12-1

13 Instant Apps

Creating an Instant App from Scratch 13-4
App Settings 13-4
Laying Out an Instant App 13-5
Panes 13-5

Elements 13-6
Events and Actions 13-25
App Events 13-26
Actions 13-28
Parameters 13-45
Using Brace Notation in Element and Parameter Values 13-47
Modes 13-48
Preview Mode 13-48
Test Mode 13-49
JSON 13-51
Starting an Instant App from a Template 13-51
Instant App Lifecycle 13-52
Editing 13-52
Publishing 13-53
Deactivating 13-53
Deleting and Restoring 13-53
Exporting and Importing 13-54

14 Reference

Built-In Components: Properties, Transitions, and Usage 14-1
Control Components 14-1
System.ConditionEquals 14-1
System.ConditionExists 14-4
System.Switch 14-4
Language 14-6
System.Intent 14-7
System.MatchEntity 14-11
System.DetectLanguage 14-13
System.TranslateInput 14-13
System.TranslateOutput 14-14
Security 14-14
System.OAuthAccountLink 14-15
User Interface Components 14-17
System.Text 14-17
System.List 14-19
System.Output 14-24
System.CommonResponse 14-28
System.Interactive 14-45
Transitions 14-49
Message Handling for Output Components 14-49

Limiting the Number of User Prompts 14-52
Variable Components 14-53
System.SetVariable 14-53
System.ResetVariables 14-54
System.CopyVariables 14-54
Apache FreeMarker Reference 14-55
Built-In String FreeMarker Operations 14-55
Example: Improving the Confidence Level with Casing 14-57
Example: Transforming Case with the System.Switch Component 14-58
Example: Concatenating FTL Expressions 14-58
Built-In FreeMarker Number Operations 14-58
Built-In FreeMarker Array Operations 14-60
Example: Iterating Arrays 14-63
Built-In FreeMarker Date Operations 14-63
Example: Extracting Dates from User Input 14-65
Example: Setting a Default Date (When No Date Value Is Set) 14-66
The SDK Helper Methods 14-69
Navigation with keepTurn and transition 14-72
The Custom Component Payload 14-75

Preface

Welcome to *Creating Intelligent Bots with Oracle Mobile Cloud, Enterprise*!

Audience

Creating Intelligent Bots with Oracle Mobile Cloud Enterprise is intended for developers who want to use bots to automate user interactions with backend data.

Documentation Accessibility

For information about Oracle's commitment to accessibility, visit the Oracle Accessibility Program website at http://www.oracle.com/pls/topic/lookup?ctx=acc&id=docacc.

Access to Oracle Support

Oracle customers that have purchased support have access to electronic support through My Oracle Support. For information, visit http://www.oracle.com/pls/topic/lookup?ctx=acc&id=info or visit http://www.oracle.com/pls/topic/lookup?ctx=acc&id=trs if you are hearing impaired.

Conventions

The following text conventions are used in this document:

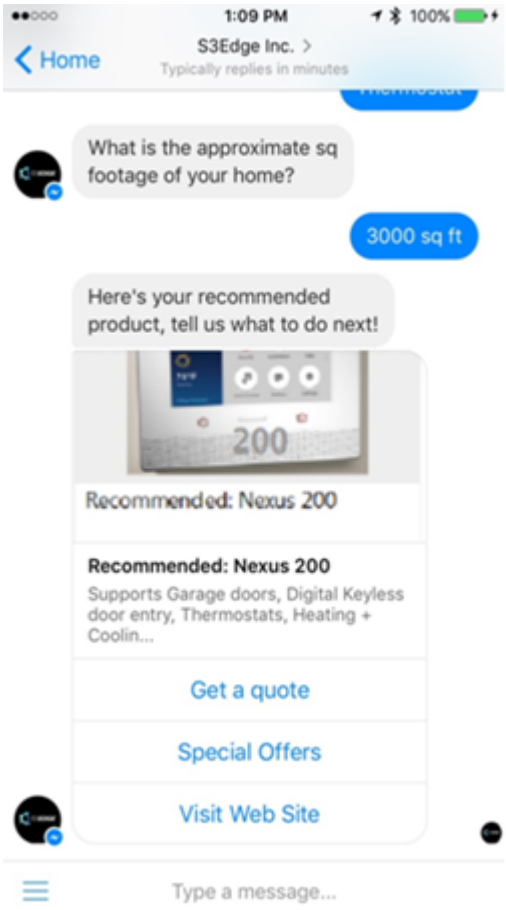
Convention	Meaning
boldface	Boldface type indicates graphical user interface elements associated with an action, or terms defined in text or the glossary.
italic	Italic type indicates book titles, emphasis, or placeholder variables for which you supply particular values.
`monospace`	Monospace type indicates commands within a paragraph, URLs, code in examples, text that appears on the screen, or text that you enter.

1

Overview

What Are Intelligent Bots?

It might be good to start off with a quick description of what a bot is. You can think of your bot as a virtual personal assistant, one that completes a task through a combination of text messages and simple UI elements like select lists. While a bot can open your enterprise to messaging, it's not a replacement for a mobile or web app. It instead provides a new channel.



Why Use Oracle Bots?

Bots enables you to build a bot that connects your users to enterprise while engaging them in naturalistic conversations. Bots manages the entire conversation. Throughout this user session (that is, the conversation, from start to finish), Bots enables your bot to keep pace with its user: it executes the functions that drive dialog all the while

keeping track of the choices that the user's made so far (the context) and where the user is within the dialog (the current state). Bots can scale to the B2C level while still managing millions of user sessions (and their states) securely. Through account linking, Bots optimizes the user experience by allowing seamless access to the bot.

While users are probably aware that they're chatting with a bot, they won't need to use (or endure) mannered, stilted language because of the language intelligence framework, which produces natural language interactions from machine learning. Because buttons or lists might provider a sleeker user experience than AI-based conversation, the framework's flexibility lets you alternate lists of options, buttons, and even forms in the dialog flow when natural language is not needed.

2
The Basics

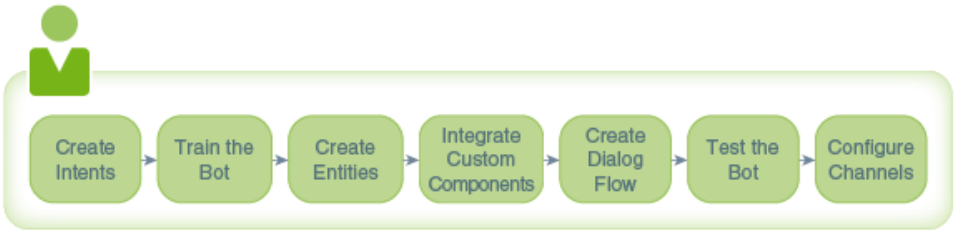
Bot Concepts

- Intents—Categories of actions or tasks users expect your bot to perform for them.
- Entities—Variables that identify key pieces of information from user input that enable the bot to fulfill a task.

  Both intents and entities are common NLP (Natural Language Processing) concepts. NLP is the science of extracting the intention of text and relevant information from text.
- Components—Provide your bot with various functions so that it can respond to users. These can be generic functions like outputting text, or they can return information from a backend and perform custom logic.
- Dialog Flow—The definition for the bot-user interaction. The Dialog flow describes how your bot responds and behaves according to user input.
- Channels—Bots aren't apps that you download from an app marketplace, like iTunes. Instead, users access them on various messaging platforms like Facebook Messenger.
- Instant Apps—Instant Apps are self-contained, wizard-liked modules that enable bot users to complete some task – a purchase that requires users to complete a form with specific field formats, for example, or an escalation.
- Bot Builder—Not only does the Bots platform provide its own A.I framework that allows your bot to parse and interpret user input, it also provides you with the Bot Builder, a UI for creating and managing all aspects of you bot, from building its cognition, to publishing it to a messaging platform.

How Do I Create a Bot?

Here's a bird's-eye view of bot development.



1. **Create Intents**—Start off by creating intents. Intents illustrate your use case by describing the various actions that your bot helps its users complete. If your bot enables users to perform various banking transactions, for example, then you could create intents like *CheckBalance* or *TransferMoney*. Intents not only

describe what your bot can do, they are also the key to your bot's intelligence: they enable it to recognize user input because each intent has a set of typical user statements known as *utterances* associated with it. While these phrases share the same meaning, they make your bot resilient because they're also varied (for example, *What's my savings account balance?* and *How much is in my checking account?*). See Intents.

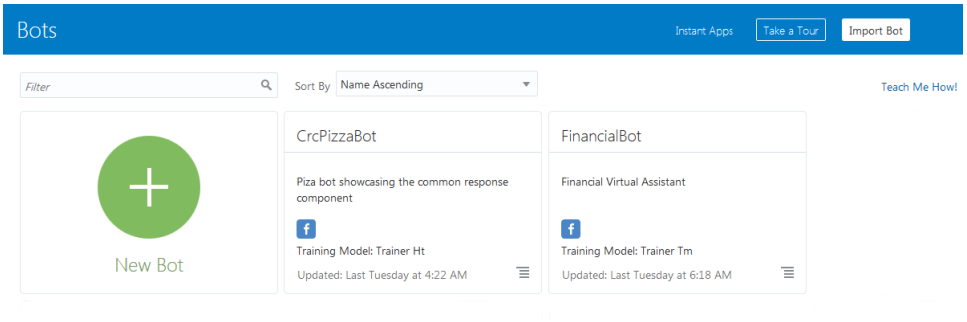
2. **Train the Bot**—To enable your bot to reference intents when it parses the user input, you need to train it. Specifically, you need to train it with the intents and their utterances (collectively, the training data), so that it can resolve the user input to one of the intents. By training your bot, you leverage the language intelligence which is at the core of the Bots platform: its algorithms enable your bot to not only recognize the sample phrases that belong to each intent, but similar phrases as well. See Intent Training and Testing.
3. **Create Entities**— In some cases, you may need to provide some context to enable your bot to complete a user request. While some user requests might resolve to the same intent (*What's my savings account balance?* and *How much is in my checking account?* would both resolve to the *CheckBalance* intent, for example), they are nonetheless asking for different things. To clarify the request, you would add an entity. Using the banking bot example, an entity called *AccountType*, which defines values called *checking* and *saving* would enable the bot to parse the user request and respond appropriately. See Entities.
4. **Integrate Custom Components**—At this point, your bot can recognize input, but it can't respond to it. To put your bot's intelligence to work, you need to add components and then later, create a dialog flow. Components enable your bot to do its job. There are two types of components: the ones provided by Bots that perform functions ranging from holding the resolved intent to outputting text, and the ones that you provide. The components belonging to this latter category are known was custom components. Unlike the built-in components that you can use in any bot that you build with the Bots platform, the custom components perform tasks that are specific to a single bot, like checking a user's age, or returning account information. Custom components don't reside within the Bots platform, so for your bot to use them, you need to access them through a REST service. See The Custom Component Service.
5. **Create the Dialog Flow**— Next, you need to give the bot the wherewithal to express its intelligence to its users by creating the dialog flow. The dialog flow describes how you bot reacts as different intents are resolved. It defines what your bot says to its users, how it prompts them for input, and how it returns data. Think of the dialog flow as a flow chart that's been transposed to a simple markdown language. In Bots, this markdown language is a version of YAML called OBotML. See The Dialog Flow Definition.
6. **Test the Bot**—Once you've started your dialog flow, you can chat with your bot to test it out..
7. **Configure Channels**—Users subscribe to your bot through messaging platforms such as Facebook Messenger. You don't have to rewrite your bot for each messaging platform, but you do need to configure a channel for each one. Bots enables you to integrate with Facebook Messenger quickly through its Facebook Channel. You don't need to craft any REST calls to run your bot on Facebook. Instead, you complete a UI using artifacts that are generated by both Facebook and Bots. See Running Your Bot on Facebook Messenger. To integrate your bot with other services, Bots provides the Webhook channel. You build your own webhooks for these non-Facebook integrations. To help you out, you can use the

sample chat server that we provide with the Developer Resources or create your own.

3
Quick Reference

Managing Your Bots

Using the landing page (accessed by clicking **Development > Bots** in the left navbar), you can manage the lifecycle of your bots by creating, revising, or deleting them. It's also where you can import a bot and access the Instant App builder and configure a translation service. See Autotranslation and Instant Apps.



Managing Your Bots

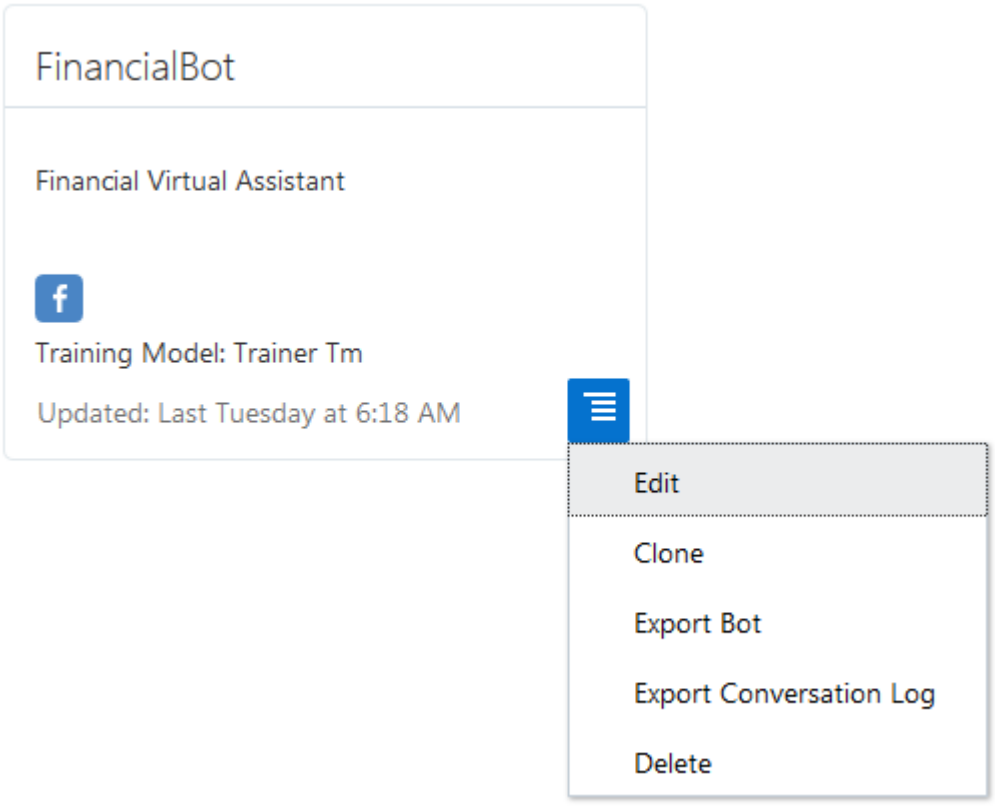
Clicking the tile menu gives you access to the following options:

- **Edit**—Update your bot.
- **Clone**—You can clone a copy of your bot to try out new features, or you can use this option to create an entirely new version of your bot.
- **Export Bot**—Share your bot as a ZIP file. This ZIP packages the different components of your bot in different formats. The bot itself is a `.json` file. The dialog flow is a `.yaml` file, and the other components (intents, entities, channels, settings, and resource bundles) are all `.json` files. You can use the **Export Bot** option, for example, when you need someone to troubleshoot your app. You can send the ZIP to your designated expert to get a second opinion. When you get it back, you add it to your bot library by clicking **Import Bot**. If you or your expert added comments to your dialog flow definition (#), they'll be preserved.

  > **Note:**
  >
  > If you import a bot, you'll need to re-enter the user credentials for component services and channels.

- **Export Conversation Log**—You can also export the conversation logs to test out new and revised versions of your bot.

- **Delete**—Trashes the bot.



4
The Sample Bots

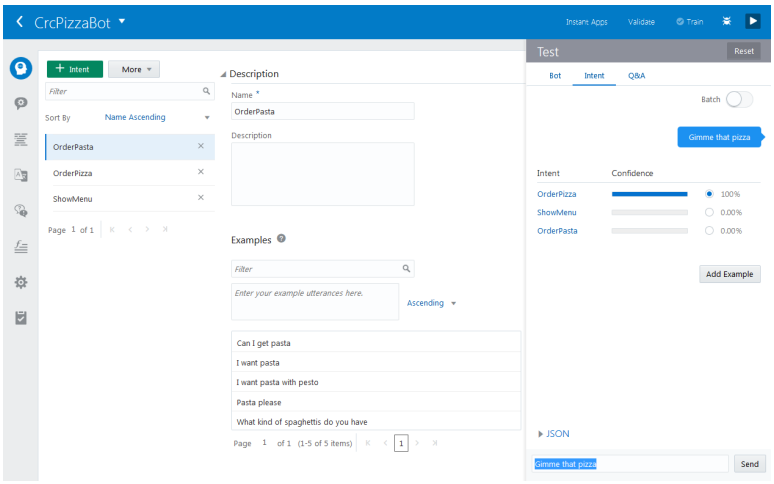
To get you familiar with the Bot Builder and some of the techniques used to create dialog flows, intents, and entities, we've provided you with some sample bots. You can use them as references as you build your own. The container for the Developer Resources, which includes these sample bots, is generated by selecting **Install Sample** when you create a Bots-only stack. Once the samples are installed, you can access them from the landing page (accessed by clicking **Development > Bots** in the left navbar).

Bot	Description
PizzaBot	Shows you the basics of bot building through a simple dialog flow.
PizzaBotWithMemory	A step up from the PizzaBot in that it demonstrates how to store user information from prior visits.
CrcPizzaBot	Another version of the PizzaBot, but this one shows you how you can incorporate scrolling menus and quick action buttons without having to write a backend service. It's all in the configuration.
FinancialBot	Another retail bot, but unlike the either of the PizzaBots, this bot shows you how to create a sophisticated dialog that maintains the user state across different transitions. This bot collects user input in two ways: through natural, free-flowing conversation and, when it needs to collect structured data, through forms and radio buttons. To do this, the bot calls a wizard-like app called an instant app. The bot passes values to the instant app, which opens in a webview. The instant app, which is populated with these values, guides the user through a series of pages. When the user completes the instant app, it seamlessly returns the user to the bot along with the values that it collected from the user.

5
Intents

Intents allow your bot to understand what the user wants it to do. An intent categorizes typical user requests by the tasks and actions that your bot performs. The PizzaBot's OrderPizza intent, for example, labels a direct request, *I want to order a Pizza*, along with another that implies a request, *I feel like eating a pizza*.

Intents are comprised of permutations of typical user requests and statements, which are also referred to as *utterances*. As described in Create an Intent, you can create the intent by naming a compilation of utterances for a particular action. Because your bot's cognition is derived from these intents, each intent should be created from a data set that's robust (one to two dozen utterances) and varied, so that your bot can interpret ambiguous user input. A rich set of utterances enables a bot to understand what the user wants when it receives messages like "Forget this order!" or "Cancel delivery!"—messages that mean the same thing, but are expressed differently. To find out how sample user input allows your bot to learn, see Intent Training and Testing.

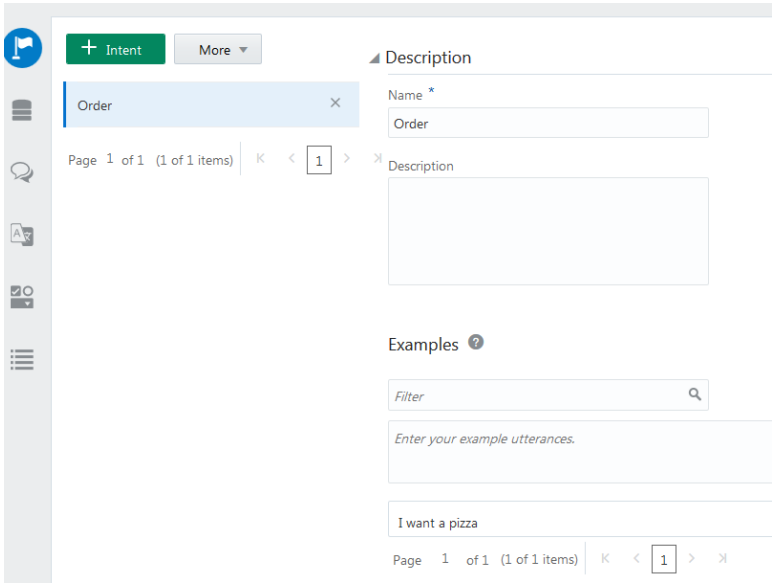


Create an Intent

To create an intent:

1. Click **Intents** (in the left navbar.
2. Click **Add Intent**.
3. Name the intent.
4. As a optional step, add description of the intent. Your description should focus on what makes the intent unique and the task or actions it performs.
5. Start building the training corpus by adding utterances that illustrate the meaning behind the intent. To ensure optimal intent resolution, use terms, wording, and phrasing specific to the individual intent. Ideally, you should base your training

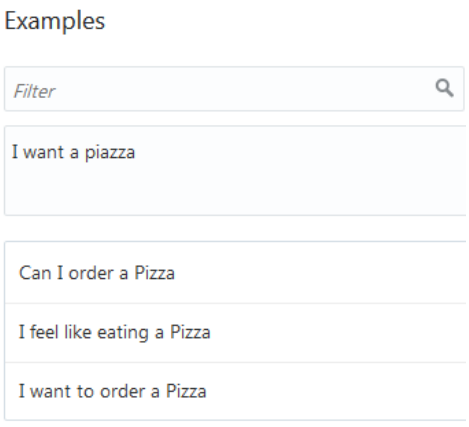
data on real-world phrases, but if you don't any, aim for one-to-two dozen utterances for each intent. That said, you can get your bot up and running with fewer (three-to-five) when you train it with Trainer Ht. You can save your utterances by clicking Enter or by clicking outside of the input field.



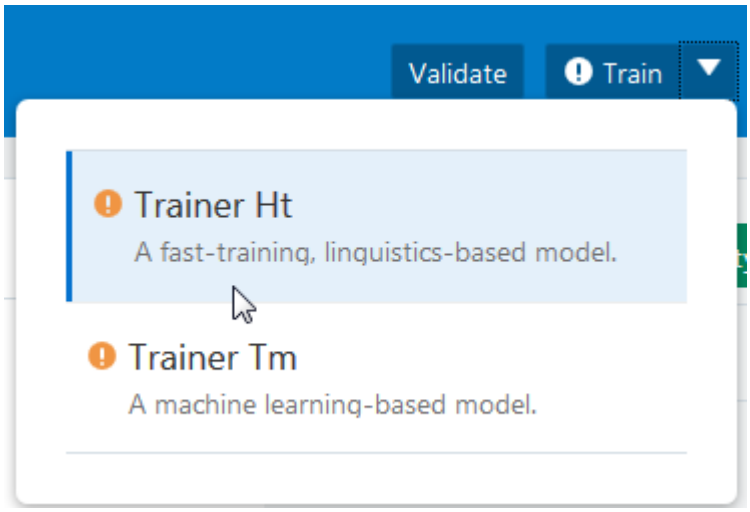
To manage the training set, select a row to access the **Edit** ( ) and **Delete** ( ) functions.

Alternatively, you can add an entire set of intents and their respective utterances by importing them from a CSV file.

You can make your bot more resilient by adding utterances that contain commonly misspelled and misused words. See Guidelines for Building Your Training Corpus.



6. Add an entity if the intent needs one to resolve the user input. To find out how, see Add Entities to Intents.
7. To teach your bot how to comprehend user input using the set of utterances that you've provided so far, click **Train**, choose a model and then click **Submit**.



As described in Which Training Model Should I Use?, we provide two models that learn from your corpus: Trainer Ht and Trainer Tm. Each uses a different algorithm to reconcile the user input against your intents. Trainer Ht uses pattern matching while Trainer Tm detects variations in user input. You'd typically follow training process when you're creating intents:

a. Create the initial training corpus.

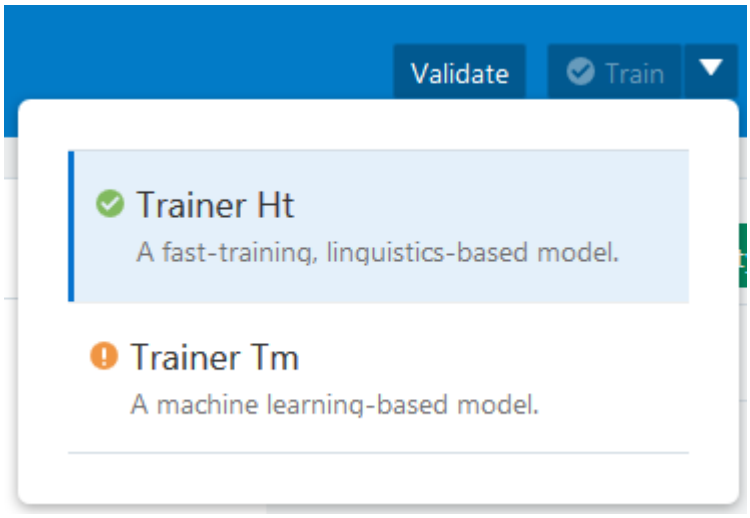
b. Train with Trainer Ht. You should start with Trainer Ht because it doesn't require a large set of utterances. As long as there are enough utterances to disambiguate the intents, your bot will be able to resolve user input.

c. Refine your corpus, retrain with Trainer Ht. Repeat as necessary—training is an iterative process.

d. Train with Trainer Tm. Use this trainer when you've accumulated a robust set of intents.

The **Train** button (

) activates whenever you add an intent or when you update an intent by adding, changing, or deleting its utterances. To bring the training up to date, choose a training model and then click **Train**. The model displays an exclamation point whenever it needs training. When its training is current, it displays a check mark.

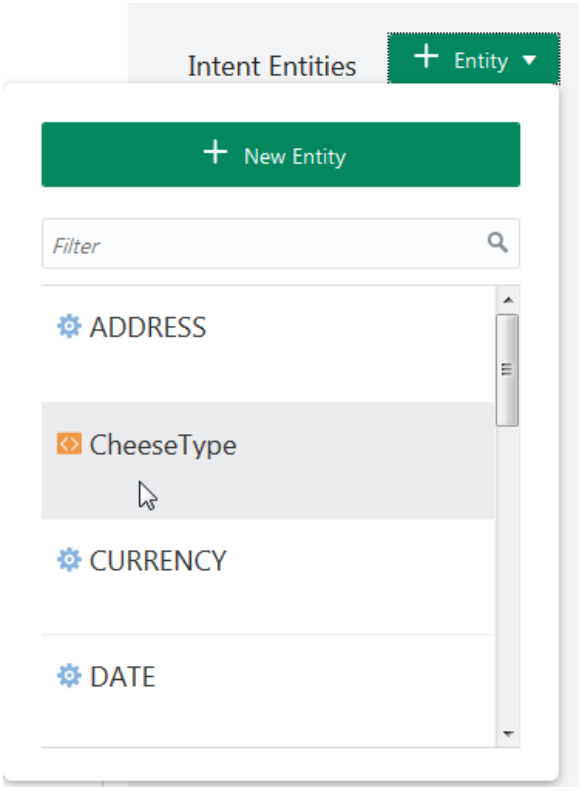


> **Note:**
>
> For Trainer Ht, your bot needs at least two intents which each have three or more utterances. Trainer Tm also requires more than one intent.

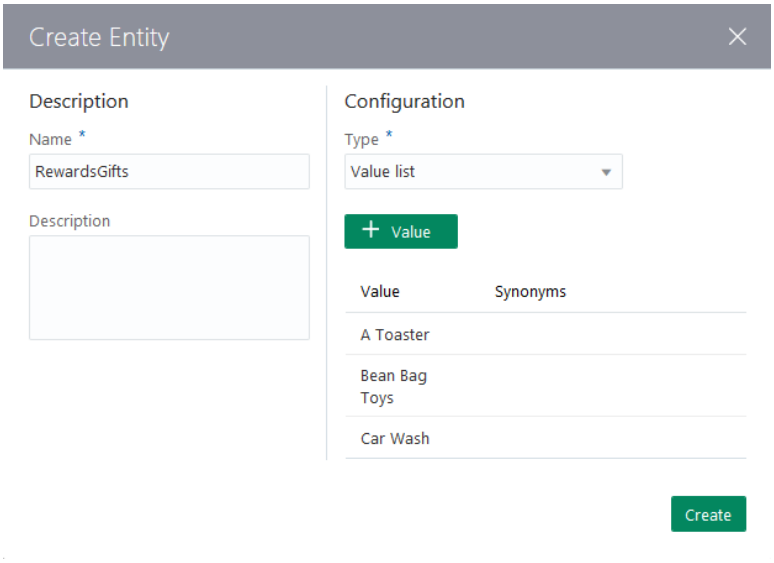
8. In the test window, click **Intents** and then enter some of the phrases from your test set. See Testing Intents.

Add Entities to Intents

Some intents require entities—both built-in and custom— to complete an action within the dialog flow or make a REST call to a backend API. The system uses only these entities, which are known, as intent entities, to fulfill the intent that's associated with them. In the absence of intent entities, the system attempts to complete the intent by iterating through all of the bot's entities. You can associate an entity to an intent when you click **Add New Entity** and then select from the custom ( ) or built-in ( ) entities.



Alternatively, you can click **New Entity** to add an intent-specific entity. See Custom Entity Types.



> **Tip:**
>
> Only intent entities are included in the JSON payloads that are sent to, and returned by, the Component Service. The ones that aren't associated with an intent won't be included, even if they contribute to the intent resolution by recognizing user input. If your custom component accesses entities through entity matches, then be sure to add the entity to your intent. See How Do Custom Components Work?

Import Intents from a CSV File

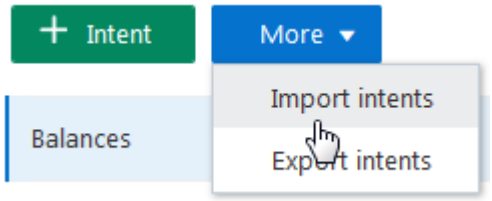
You can add your intents manually, or import them from a CSV file. You can create this file by exporting the intents and entities from another bot, or by creating it from scratch in a spreadsheet program or a text file.

The CSV file has two columns: `query` and `topIntent`:

```
query,topIntent
I want to order a pizza,OrderPizza
I want a pizza,OrderPizza
I want a pizaa,OrderPizza
I want a pizzaz,OrderPizza
I'm hungry,OrderPizza
Make me a pizza,OrderPizza
I feel like eating a pizza,OrderPizza
Gimme a pie,OrderPizza
Give me a pizza,OrderPizza
pizza I want,OrderPizza
I do not want to order a pizza,CancelPizza
I do not want this,CancelPizza
I don't want to order this pizza,CancelPizza
Cancel this order,CancelPizza
Can I cancel this order?,CancelPizza
Cancel my pizza,CancelPizza
Cancel my pizaa,CancelPizza
Cancel my pizzaz,CancelPizza
I'm not hungry anymore,CancelPizza
```

To import a CSV file:

1. Click **Intents** ( ) in the left navbar.
2. Click **More**, and then choose **Import intents**.



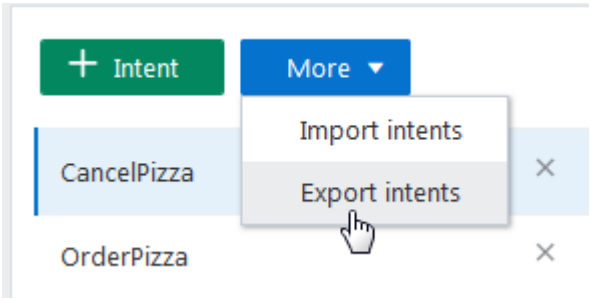
3. Select the `.csv` file and then click **Open**.
4. Train your bot.

Export Intents to a CSV File

You can reuse your training corpus by exporting it to CSV. You can then import this file to another bot.

To export your intents and their utterances:

1.
2. Click **Intents** (in the left navbar.
3. Click **More**, and then choose **Export intents**.



4. Save the file.

> **Tip:**
>
> Remember to train your bot after you import the CSV file.

Intent Training and Testing

Training a model with your training corpus allows your bot to discern what users say (or in some cases, are trying to say).

You can improve the acuity of the cognition through rounds of intent testing and intent training. In Bots, you control the training through the intent definitions alone; the bot can't learn on its own from the user chat.

Test Sets

We recommend that you set aside 20% percent of your corpus for testing your bot and train your bot with the remaining 80%. Keep these two sets separate so that the test set remains "unknown" to your bot.

Apply the 80/20 split to the each intent's data set. Randomize your utterances before making this split to allow the training models to weigh the terms and patterns in the utterances equally.

The Intent Tester

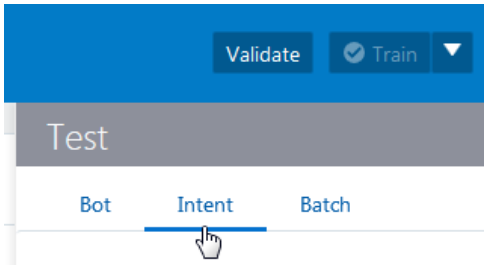
The Intent tester is your window into your bot's cognition. By entering phrases that are not part of the training corpus (the utterances that you've maintained in your testing

set), you can find out how well you've crafted your intents and entities through the ranking and the returned JSON. This ranking, which is the bot's estimate for the best candidate to resolve the user input, demonstrates its acuity at the current time.

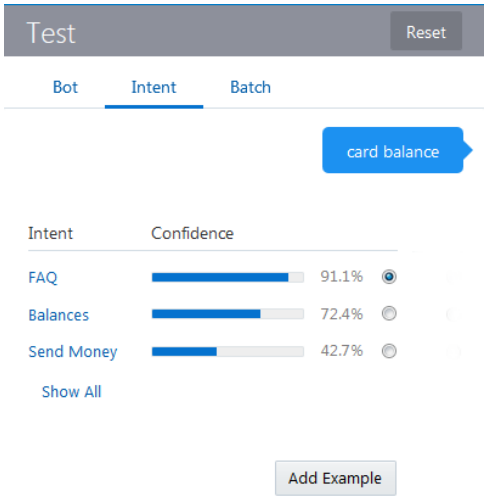
Testing Intents

To find out how well you've labeled your intents and entities:

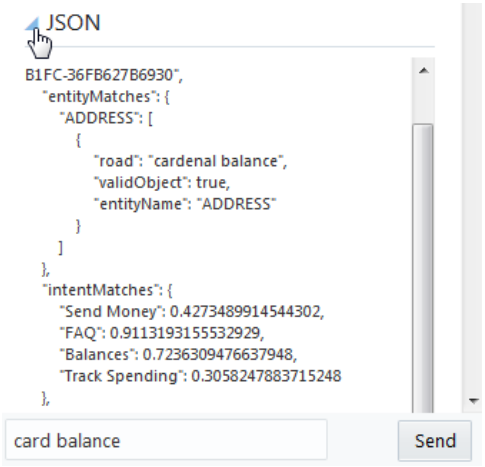
1. Click **Test** ( ) to open the tester window.
2. Open the Intent tab.



3. Enter a string of text that is not part of the corpora.
4. Click **Send** and then take a look at the ranking.

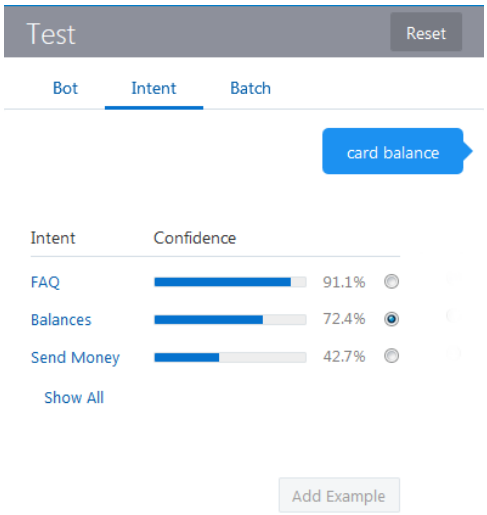


5. Expand the **JSON** window to see how your bot ranked the intents.



If your bot's top-ranking candidate isn't what you expect, you might need to retrain the intents after doing one or both of the following:

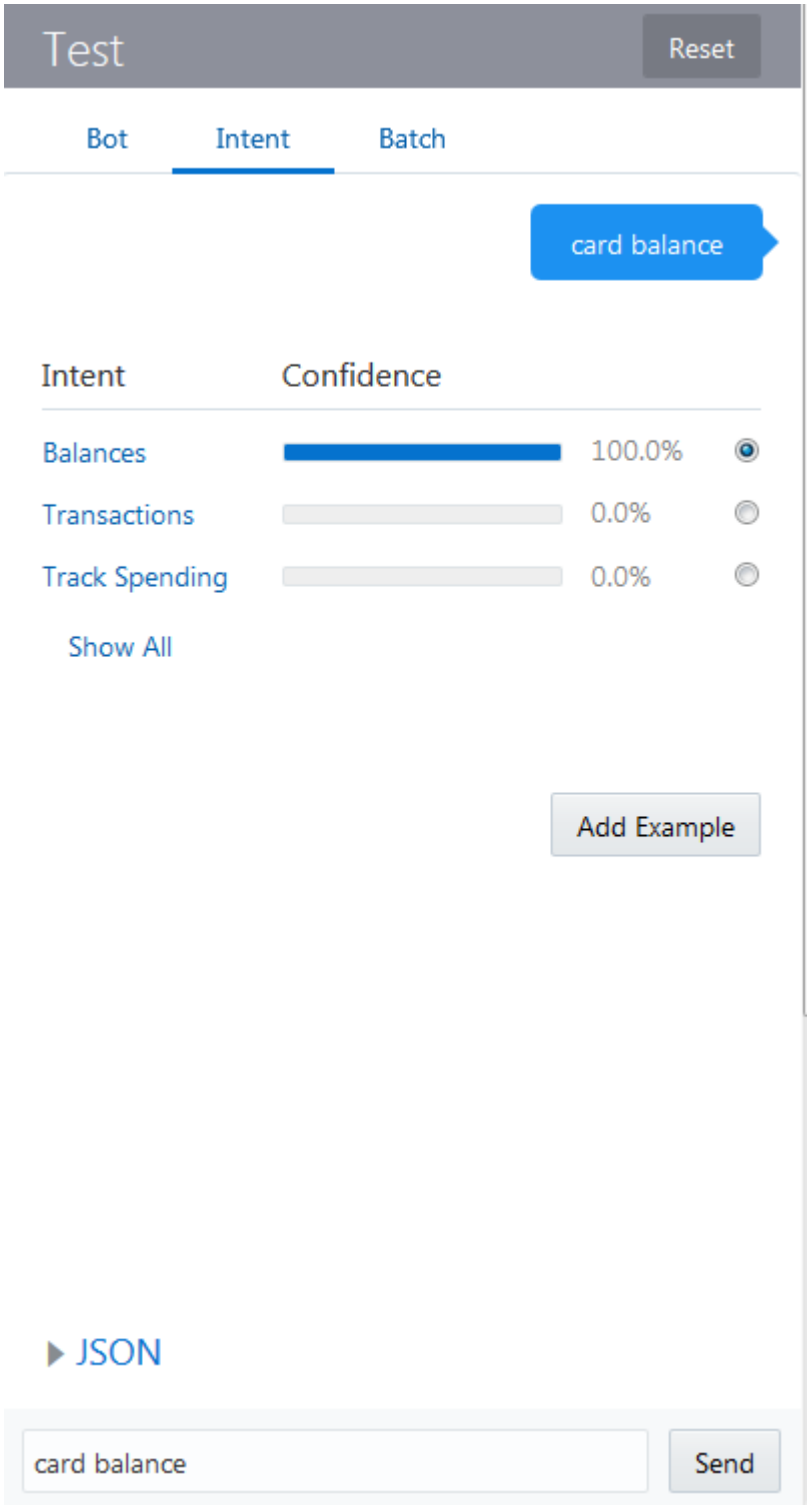
- Update the better candidate's corpus with the input text that you just entered—Select the appropriate intent and then click **Add Example**.



> **Caution:**
>
> Adding a test phrase can change how utterances that are similar to it get classified after you retrain the bot, so consider the impact before updating the training set with a test phrase. In addition, adding a test phrase invalidates the test, because it's now incorporated into the training set and therefore ensures that the test will be successful.

- Correct the system by editing the corpus using the **Edit** ( ) and **Delete** ( ) functions.

You need to retrain an intent whenever you add, change, or delete an utterance. A dirty **Train** icon (Train) indicates when your training becomes outdated. When the retraining completes, click **Reset** (Reset) and then send the test phrase again.



The Intent Testing History

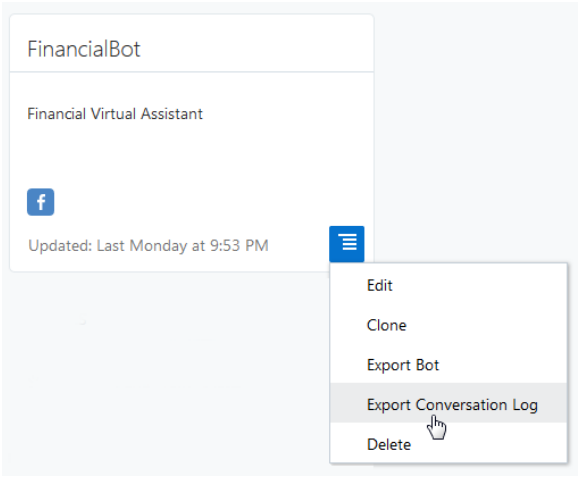
You can export the training data into CSV file so that you can find out how the intents were trained.

By examining these logs in a text editor or spreadsheet program like MicroSoft Excel, you can see each user request and bot reply. You can sort through these logs to see where the bot matched the user request with the right intent and where it didn't.

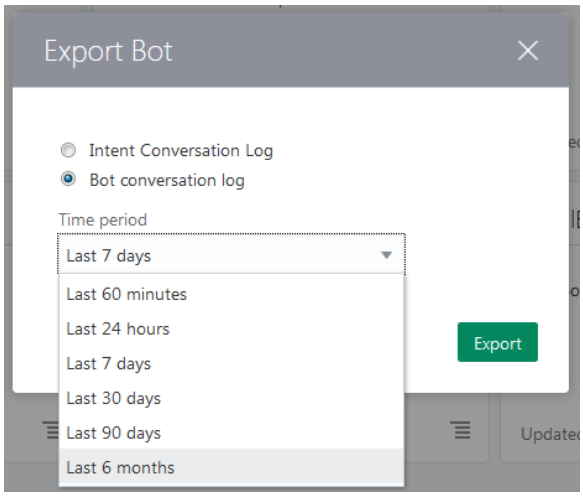
Export Intent Data

To export data:

1. In the bots catalog, open the menu in the tile and then click **Export Conversation Log**.



2. In the Export Bot dialog, choose the log type (conversation or intent) and a logging period.



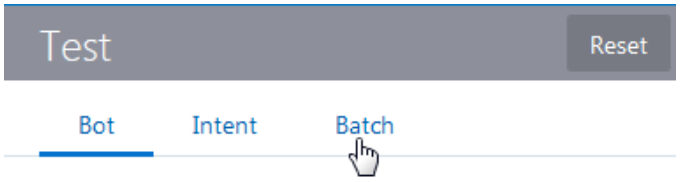
3. Open the CSV files in a spreadsheet program to review it. You see if your model matches intents consistently by filtering the rows by keyword.

Batch Testing Intents

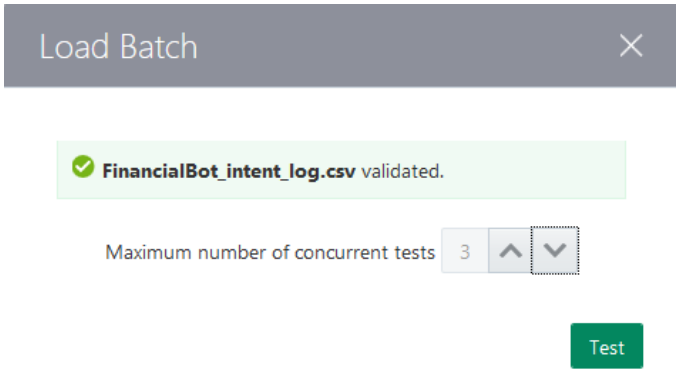
You can use the intent testing data that you've exported on new iterations of your bot to gauge the accuracy of its intent detection.

To use that test data:

1. Open the tester ( ) and then click **Batch**.

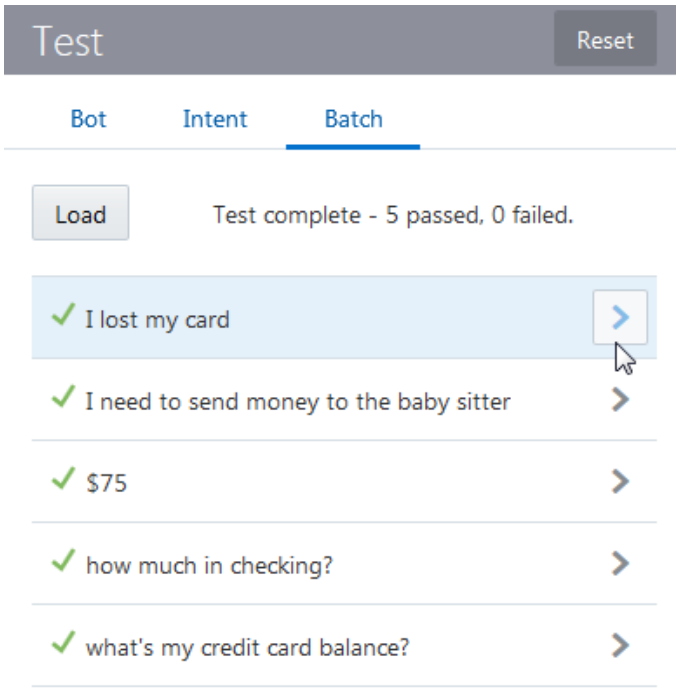


2. Click **Load** and then upload the intents log.



3. Choose the number of tests running in parallel. Increasing the number of concurrent tests may speed up testing, but may also burden the system.
4. Click **Test**.

   The results display in the test window.



5. Drill down (>) to see how the test results compare to the batch data.

Which Training Model Should I Use?

We provide a duo of models that you can train to mold your bot's cognition. You can use one or both of these models, each of which uses a different approach to machine learning.

Trainer Ht

Trainer Ht is the default training model. It needs only a small training corpus, so use it as you develop the entities, intents, and the training corpus. When the training corpus has matured to the point where tests reveal highly accurate intent resolution, you're ready to add a deeper dimension to your bot's cognition by training Trainer Tm.

You can get a general understanding of how Trainer Ht resolves intents just from the training corpus itself. It forms matching rules from the sample sentences by tagging parts of speech and entities (both custom and built-in) and by detecting words that have the same meaning within the context of the intent. If an intent called *SendMoney* has both *Send $500 to Mom* and *Pay Cleo $500*, for example, Trainer Ht interprets *pay* as the equivalent to *send* . After training, Trainer Ht's tagging reduces these sentences to templates (*Send Currency to person*, *Pay person Currency*) that it applies to the user input.

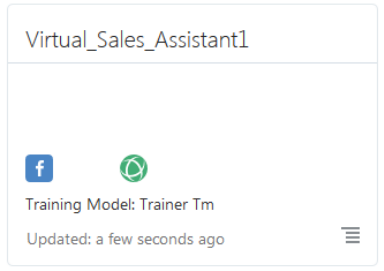
Because Trainer Ht draws on the sentences that you provide, you can predict its behavior: it will be highly accurate when tested with sentences similar to the ones that make up the training corpus (the user input that follows the rules, so to speak), but may fare less well when confronted with esoteric user input.

> **Tip:**
>
> Because of its quick training, use Trainer Ht to help you define and refine your training corpus. While you can add sentences to an intent whenever the resolution is faulty (or in the worst case, add your entire testing corpus), be sure to aim for a concise training corpus by following the guidelines in Guidelines for Building Your Training Corpus.

Trainer Tm

Because Trainer Tm doesn't focus as heavily on matching rules as Trainer Ht, it can help your bot interpret user input that falls outside of your training corpus. Trainer Tm differs from Trainer Ht in other ways as well: its intent resolution can be less predictable across training sessions.

> **Note:**
>
> Trainer Ht is the default model, but you can change this by clicking **Settings > General** and then by choosing another model from the list. The default model displays in the tile in the bot catalog.



Guidelines for Building Your Training Corpus

When you define an intent, you first give it a name that illustrates some user action and then follow up by compiling a set of real-life user statements, or utterances. Collectively, your intents, and the utterances that belong to them, make up a training corpus. The term *corpus* is just a quick way of saying "all of the intents and sample phrases that I came up with to make this bot smart". The corpus is the key to your bot's intelligence. By training a model with your corpus, you essentially turn that model into a reference tool for resolving user input to a single intent. Because your training corpus ultimately plays the key role in deciding which route the bot-human conversation will take, you need to choose your words carefully when building it.

Generally speaking, a large and varied set of sample phrases increases a model's ability to resolve intents accurately. But building a robust training corpus doesn't just begin with well-crafted sample phrases; it actually begins with intents that are clearly delineated. Not only should they clearly reflect your use case, but their relationship to their sample sentences should be equally clear. If you're not sure where a sample sentence belongs, then your intents aren't distinct from one another.

You probably have sample utterances in mind when you create your intents, but you can expand upon them by using these guidelines:

- Create 12 to 24 sample phrases per intent (if possible). Keep in mind that the more examples you add, the more resilient your bot becomes.

  > **Important:**
  >
  > Trainer Tm can't learn from an intent that has only one utterance.

- Avoid sentence fragments and single words. Instead, use complete sentences (which can be up to 255 characters). If you must use single key word examples, choose them carefully.
- Vary the vocabulary and sentence structure in your sample phrases by one or two permutations using:
  - slang words (moolah, lucre, dough)
  - common expressions (*Am I broke?* for an intent called *AccountBalance*)
  - alternate words (*Send cash to savings*, *Send funds to savings*, *Send money to savings*, *Transfer cash to savings*.)
  - different categories of objects (*I want to order a pizza*, *I want to order some food*.
  - alternate spellings (check, cheque)
  - common misspellings ("buisness" for "business")
  - unusual word order (*To checking, $20 send*)
  - Create parallel sample phrases for opposing intents. For intents like *CancelPizza* and *OrderPizza*, define contrasting sentences like *I want to order a pizza* and *I do not want to order a pizza*.
  - When certain words or phrases signify a specific intent, you can increase the probability for a correct match by bulking up the training data not only with the words and phrases themselves, but with synonyms and variations as well. For example, a training corpus for an *OrderPizza* intent might include a high concentration of "I want to" phrases, like *I want to order a Pizza*, *I want to place an order*, and *I want to order some food*. Use similar verbiage sparingly for other intents, because it might skew the training if used too freely (say, a *CancelPizza* intent with sample phrases like *I want to cancel this pizza*, *I want to stop this order*, and *I want to order something else*). When the high occurrence of unique words or phrases within an intent's training set is unintended, however, you should revise the initial set of sentences or use the same verbiage for other intents.

    Use different concepts to express the same intent, like *I am hungry* and *Make me a pizza*.
- Watch the letter casing: use uppercase when your entities extract proper nouns, like Susan and Texas, but use lowercase everywhere else.
- Grow the corpus by adding any mismatched sentence to the correct intent.

> **Tip:**
>
> Keep a test corpus as CSV file to batch test intent resolution by clicking **More** and then **Export Intents**. Because adding a new intent example can cause regressions, you might end up adding several test phrases to stabilize the intent resolution behavior.

Reference Intents in the Dialog Flow

Within your dialog flow, your intents can define the `actions` property, as shown in the PizzaBot's `intent` state. See System.Intent.

```
intent:
  component: "System.Intent"
  properties:
    variable: "iResult"
    confidenceThreshold: 0.4
  transitions:
    actions:
      OrderPizza: "resolvesize"
      CancelPizza: "cancelorder"
      unresolvedIntent: "unresolved"
```

6
Entities

While intents map words and phrases to a specific action, entities add context to the intent itself. They help to describe the intent more fully and enable your bot to complete a user request. The *OrderPizza* intent, for example, describes a user request, but only in general terms. To fill in the specifics, this intent is augmented by the *PizzaSize* entity, which identifies values like *large*, *medium*, and *small* from the user input. There are two types of entities, both of which you can declare as variables in the dialog flow: built-in entities that we provide for you and custom entities, which you can add on your own.

Built-In Entities

We provide entities that identify objective information from the user input, like time, date, and addresses.

These built-in entities are divided into two groups: simple entities that extract primitive values like strings and integers, and complex entities that detect values from the user input using groups of properties.

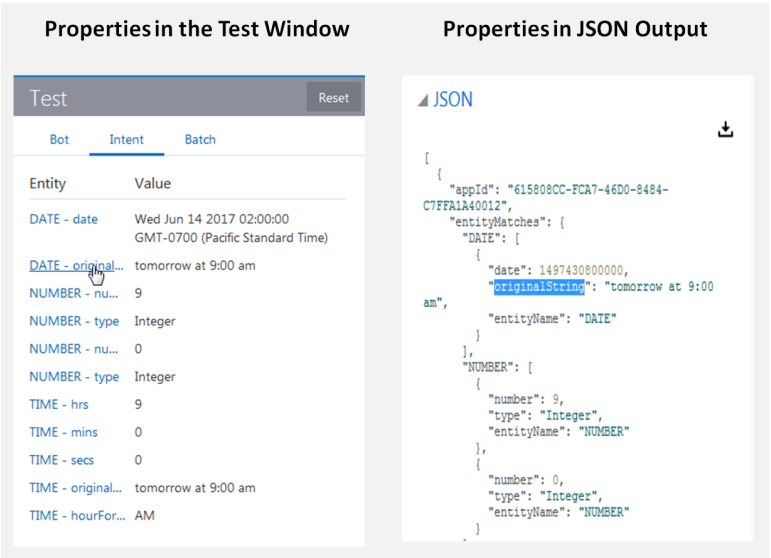
Whenever you define a variable as an entity in your dialog flow, be sure to match the entity name and letter case exactly. In other words, you'll get a validation error if you enter `confirm: "YESNO"` instead of `confirm: "YES_NO"`.

Simple Entities

Entity Name	Content Identified	Examples
NUMBER	Matches ordinal and cardinal numbers	• 1st • first • 1 • one
EMAIL	An email address—The NLU system can recognize email addresses that have combinations of letters (a-z), numbers (0–9), underscores (_) and plus (+) and minus (-) signs before the at symbol (@). The address can't have a dot (.) immediately after the @ symbol, but the domain name (which is comprised of letters and numbers only), must include the dot.	ragnar.smith@example.com
ADDRESS	An address or location-related entities	200 Oracle Parkway, Redwood Shores, CA
YES_NO	Detects a "yes" or "no".	Yes, Yeah, no

Complex Entities

Unlike simple entities, complex entities extract content using properties, each of which recognizes a specific value. When you train and test your intents, you can see these properties in the Tester. The JSON output that's returned by the NLU Engine shows these entities along with the value that they've identified from the user input.



When you define a variable in your dialog flow that uses a complex entity, you will need to include these properties in the value expression. See Accessing Variable Values with Apache FreeMarker FTL.

Entity Name	Content Extracted	Examples	Properties (Referenced in Value Expressions)	Example NLU Engine Response
DATE	An absolute or relative date	• November 9, 2016 • Today	• originalString • date	{ "date" : 895685400000, "value" : 895685400000, "entityName" : "DATE" }
TIME	A specific time	2:30 pm	• hrs • mins • secs • “hourFormat”:”PM”	{ "hrs":8, "mins":0, "secs":0, "hourFormat":"PM", "entityName":"TIME" }
DURATION	The amount of time between the two endpoint of a time interval	• 4 years • two weeks	• startDate • endDate	{ "startDate": 1482019200000, "endDate": 1482623999999, "entityName":"DURATION" }
SET	Recurring time periods.	• Every Tuesday • Every two weeks	• minute—The range is {0–59} • hour—The range is {0–23} • dayOfTheMonth—The range is {1–31} • monthOfTheYear—The range is {1–12} • dayOfTheWeek—{0–6}, with 0 being Sunday • year	{ "minute" : [30], "hour" : [19], "dayOfTheMonth" : [15], "monthOfTheYear" : ["MARCH"], "entityName" : "SET" }

Entity Name	Content Extracted	Examples	Properties (Referenced in Value Expressions)	Example NLU Engine Response
CURRENCY	Representations of money	• $67 • 75 dollars	• amount • currency • totalCurrency	{ "amount":50, "currency":"dollar", "total_currency":"50.0 dollar", "entityName":"CURRENCY" }
PHONE NUMBER	A phone number—The NLU Engine recognizes phone numbers that have seven or more digits (it can't recognize any phone number with fewer digits). All country codes need to be prefixed with a plus sign (+), except for the United States of America (where the plus sign is optional). The various parts of the phone number (the area code, prefix, and line number), can be separated by dots (.), dashes (-), or spaces. If there are multiple phone numbers entered in the user input, then the NLU Engine can recognize them when they're separated by commas. It can't recognize different phone numbers if they're separated by dots, dashes or spaces.	• (650)-555–5555 • 16505555555 • +61.3.5555.5555	• phoneNumber • completeNumber	{ "phone_number":"(650)-555-5555", "complete_number":"(650)-555-5555", "entityName":"PHONE_NUMBER" }

Entity Name	Content Extracted	Examples	Properties (Referenced in Value Expressions)	Example NLU Engine Response
URL	A URL—This entity can extract IPv4 addresses, Web URLs, deep links (`http://example.com/path/page`), file paths, and `mailto` URIs. If the user input specifies login credentials, then it must also include the protocol. Otherwise, the protocol isn't required.	`http://example.com`	• `protocol` • `domain` • `fullPath`	`{"protocol":"http","domain":"example.com",}`

Custom Entities

Because the built-in entities extract generic information, they can be used in a wide variety of bots. Custom entities, on the other hand, have a narrower application. Like the FinancialBot's *AccountType* entity that enables various banking transactions by checking the user input for keywords like *checking*, *savings*, and *credit cards*, they're tailored to the particular actions that your bot performs.

Custom Entity Types

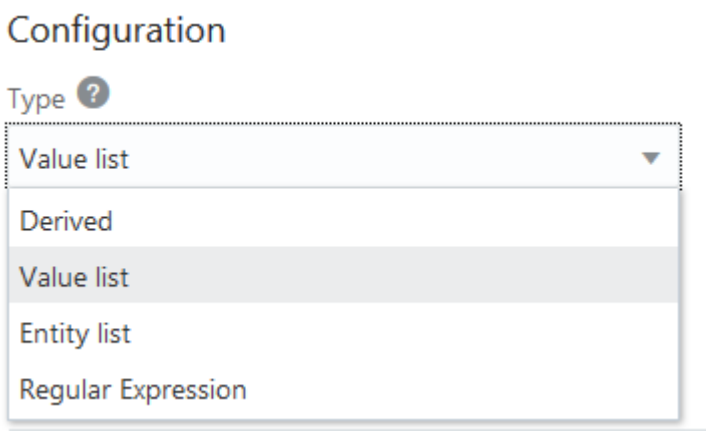
Entity Type	Description
Derived	A derived entity is the child of a built-in entity or another entity that you define. You base this relationship on prepositional phrases (the "to" and "from" in utterances like *I want to go from Boston to Dallas* or *Transfer money from checking to savings*).
Value list	An entity based on a list of predetermined values, like menu items or the FinancialBot's *checking*, *credit*, and *card* options that are output by the `System.List` component. You can optimize the entity's ability to extract user input by defining synonyms. These can include abbreviations, slang terms, and common misspellings. Synonym values are not case-sensitive: *USA* and *usa*, for example, are considered the same value.

Entity Type	Description
Entity list	A super set of entities. Using a travel bot as an example, you could fold the entities that you've already defined that extract values like airport codes, cities, and airport names into a single entity called *Destination*. By doing so, you would enable your bot to respond to user input that uses airport codes, airport names, and cities interchangeably. So when a user enters "I want to go to from JFK to San Francisco," the *Destination* entity detects the departure point using the airport code entities and the destination using the cities entity.
Regular Expression	Resolves an entity using a regular expression (regex). Using regular expressions lets your bot identify pre-defined patterns in user inputs, like ticket numbers.

Create Entities

To create an entity:

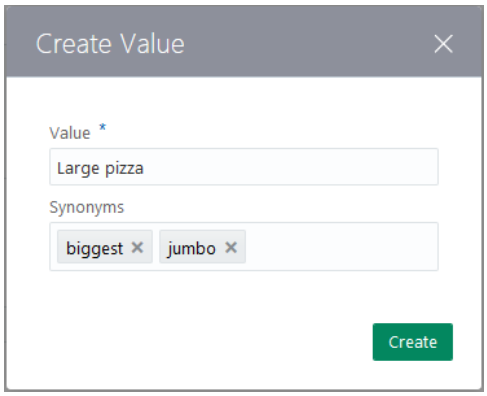
1. Click **Entities** ( ) in the side navbar.
2. Click **Add Entity** and then enter the name.
3. In the Configuration section, choose entity type from the list.



- **Value List**—Define a set of values for an entity that's defined by predetermined terms.

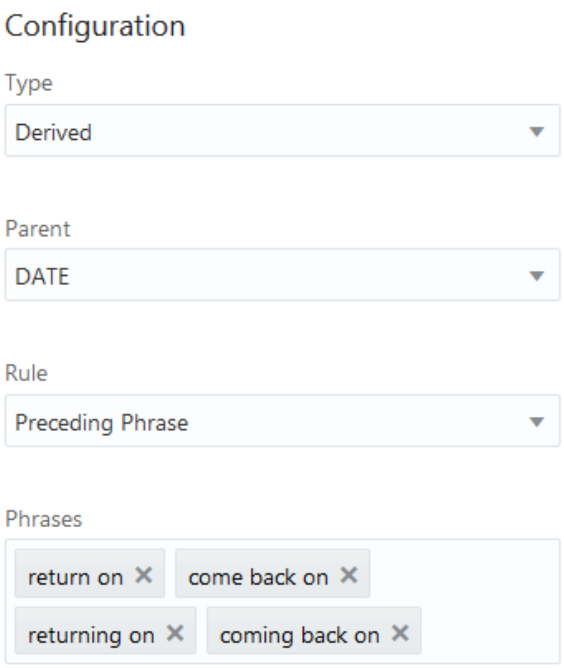
> **Note:**
>
> Adding punctuation or special characters to the same term creates different values: the entity can't match USA from the user input with either *USA?* or *USA!*, or cafe with *café*. To match a value regardless of case or punctuation, create a regular expression entity. You can define it using syntax like `(\w+)`

.



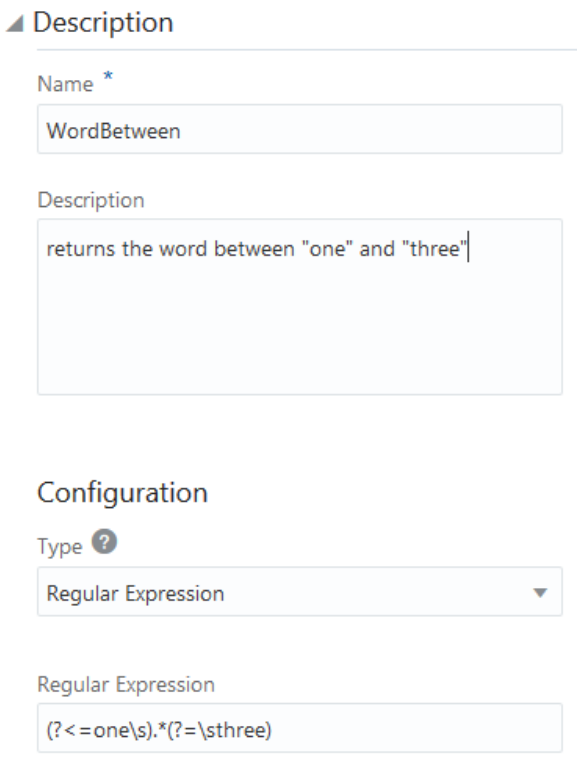
> **Tip:**
>
> You don’t need to add your value list entities one at a time. Like intents, you can define groups of entities in a CSV file first, then import them. You can create this file from scratch, or reuse the entities that you’ve exported from another bot.

- **Entity List**—A super set of entities.
- **Derived Entities**—A derived entity is the child of another entity (either built-in or custom) that’s modified by a preposition that you add using the **Preceding Phrase** or **Following Phrase** rules.



> **Important:**
>
> Derived entities can't be parent entities. And because the NLU Engine detects derived entities only after it detects all of the other types of entities, you can't add derived entities as members of an entities list. Only custom or the built-in entities can belong to a list.

- **Regular Expression**—Enter the regular expression (regex) pattern. Unlike the other entity types, regex-based entities don't use NLP because the matching is strictly pattern-based.



For example this expression, `(?<=one\s).*(?=\sthree)` returns the word that's between "one" and "three" in the user input.

4. Next steps:
   a. Add the entity to an intent. This informs the bot of the values that it needs to extract from the user input during the language processing. See Add Entities to Intents.
   b. If needed, add the entity to an Entity List entity or a Derived entity.
   c. In the dialog flow, declare a context variable for the entity, one that references that value that holds the language processing result, `nlpresult` (for example, `iResult: "nlpresult"`). See Dialog Flow Syntax.
   d. Reference this context variable in the System.Intent component.

      The System.MatchEntity also extracts entity values.

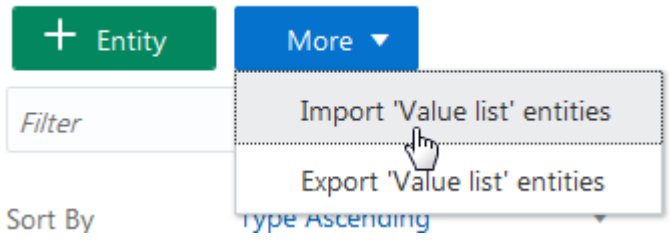
e. After the `System.Intent` component, which processes the input string, access the variable values using Apache FreeMarker expressions. For example, you can use the `entityMatches` keyword to iterate over entity values (`${iResult.value.entityMatches['name of entity']}`). See Built-In FreeMarker Array Operations.

Import Value List Entities from a CSV File

Rather than creating your entities one at a time in the Bot Builder, you can create entire sets of them when you import a CSV file containing the entity definitions. The CSV file is divided into three columns: `entity`, `value`, and `synonyms`. For example:

```
entity,value,synonyms
CheeseType,Mozzarella,Mozarela:Mozzarela
CheeseType,Provolone,
CheeseType,Gouda,
CheeseType,Cheddar,
PizzaSize,Large
PizzaSize,Medium
PizzaSize,Small
```

1. Click **Entities** ( ) in the side navbar.
2. Click **More**, choose **Import Value list entities**, and then select the `.csv` file from your local system.

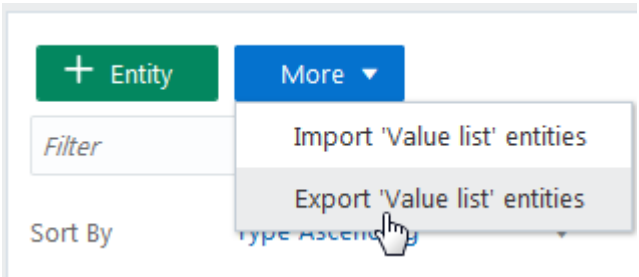


3. Add the entity or entities to an intent (or to an entity list and then to an intent).

Export Value List Entities to a CSV File

You can export your entity definitions in a CSV file for reuse in another bot. To do this:

1. Click **Entities** ( ) in the side navbar.
2. Click **More**, choose **Export Value list entities** and then save the file.



The exported `.csv` file is named for your bot.

7

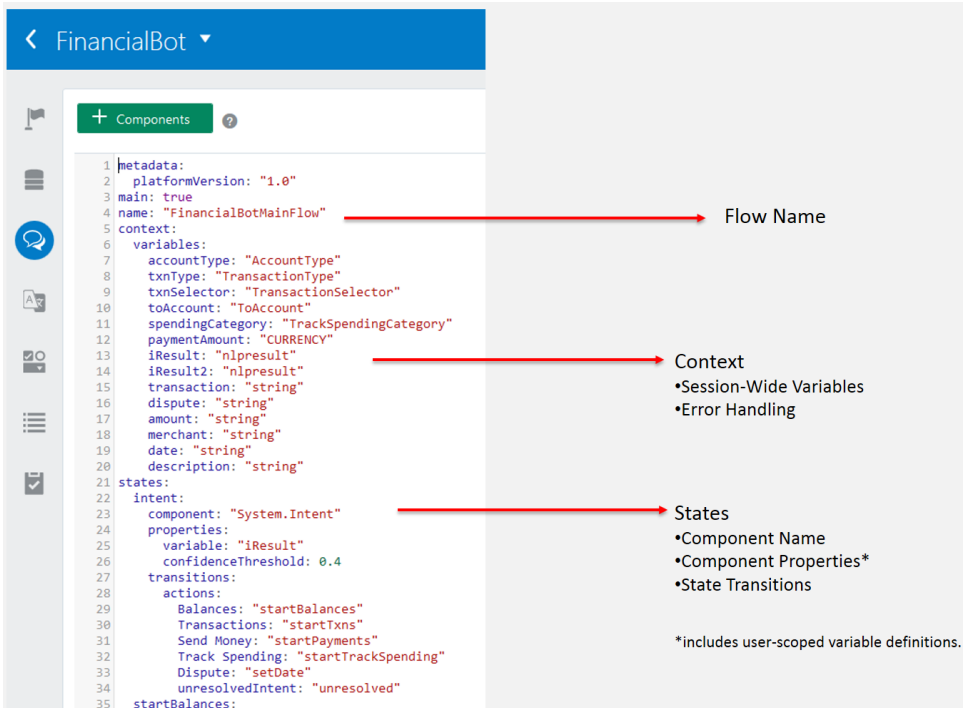
The Dialog Flow Definition

The dialog flow definition is the model for the conversation itself, one that lets you choreograph the interaction between the bot and its users.

Using the Bot Builder, you define the framework of the user-bot exchange in OBotML, Bots' own implementation of YAML. This is a simple markup language, one that lets you describe a dialog both in terms of what your bot says and what it does.

The Dialog Flow Structure

Your OBotML definition is divided into three main parts: `context`, `defaultTransitions`, and `states`. You define the variables that are available across the session within the `context` node. The definition of the flow itself is described in the `states` section.



The dialog flow is laid out as follows:

```
main: true
name: "HelloKids"
context:
  variables:
    variable1: "entity1"
    variable2: "error"
...
States
    state1:
```

```
      component: "a custom or built-in component"
      properties:
        property1: "component-specific property value"
        property2: "component-specific property value"
      transitions:
        actions:
          action1: "value1"
          action2: "value2"
    state2:
      component: "a custom or built-in component"
      properties:
        property1: "component-specific property value"
        property2: "component-specific property value"
      transitions:
        actions:
          action1: "value1"
          action2: "value2"
...
```

Context

The variables that you define within the `context` node can be primitive types like `int`, `string`, `boolean`, `double`, or `float`. They can also describe error handling, or, as in the following snippet from the PizzaBot dialog flow definition, they name entities like `PizzaSize` and `PizzaCrust`. Along with built-in entities and the custom entities, you can also declare a variable for the `nlpresult` entity, which holds the intent that's resolved from the user input. These variables are scoped to the entire flow. How Do I Write Dialog Flows in OBotML? tells you how to assemble the different parts of the dialog flow. You can also scope user variable values to enable your bot to recognize the user and persist user preferences after the first conversation. See User-Scoped Variables.

```
metadata:
platformVersion: "1.1"
main: true
name: "PizzaBot"
context:
  variables:
    size: "PizzaSize"
    type: "PizzaType"
    crust: "PizzaCrust"
    iResult: "nlpresult"
```

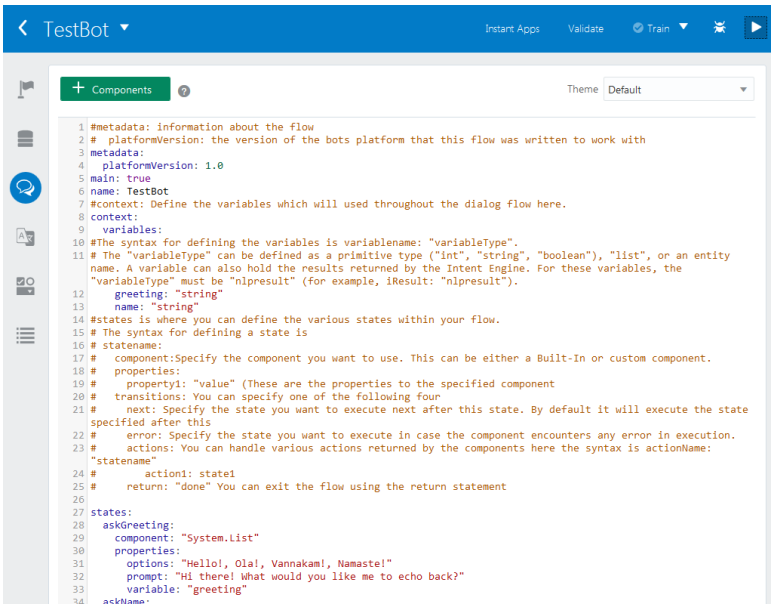
States

You define each bit of dialog and its related operations as a sequence of transitory states, which manage the logic within the dialog flow. To cue the action, each `state` node within your OBotML definition names a component along with component–specific properties and transitions that trigger the next state. The PizzaBot includes a sequence of `state` nodes that verify a customer's age. These states include components that take the user-supplied integer value, check it, and then output a text string as appropriate. To start off the process, the `askage` state's component requests the user input then moves on to the `checkAge` state, whose `AgeChecker` component validates the user input. Here, the dialog is at a juncture: its `transitions` key defines the `block` or `allow` states. If the `allow` state is triggered, then the user can continue on. The subsequent state definitions will track the user input to preserve the user's context until she completes her order. If the user input causes the `AgeChecker` component to trigger the `block` action, however, then conversation ends for the under-age user because the dialog transitions to the `underage` state.

```
metadata:
platformVersion: "1.1"
main: true
name: "PizzaBot"
context:
  variables:
    size: "PizzaSize"
    type: "PizzaType"
    crust: "PizzaCrust"
    cheese: "CheeseType"
    iResult: "nlpresult"
...

  askage:
    component: "System.Output"
    properties:
      text: "How old are you?"
    transitions: {}
  checkage:
    component: "AgeChecker"
    properties:
      minAge: 18
    transitions:
      actions:
        allow: "crust"
        block: "underage"
  crust:
    component: "System.List"
    properties:
      options: "Thick,Thin,Stuffed,Pan"
      prompt: "What crust do you want for your Pizza?"
      variable: "crust"
    transitions: {}
...
    underage:
    component: "System.Output"
    properties:
      text: "You are too young to order a pizza"
    transitions:
      return: "underage"
```

How Do I Write Dialog Flows in OBotML?

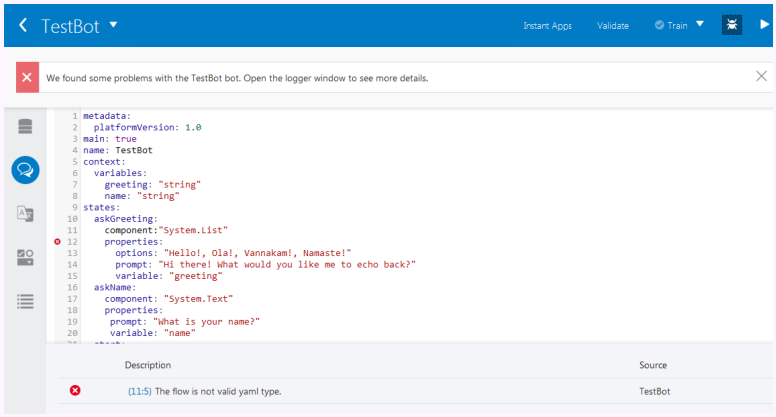
OBotML uses a simple syntax for setting variables and defining states. Because it's a variant of YAML, keep the YAML spacing conventions in mind when you define the dialog flow. You don't need to start from scratch. Instead, you can use the Echo bot's dialog flow definition as a basic template.



Along with the metadata definition at the top of the dialog, the Echo bot already has the `context` and `states` nodes, so you can just delete the existing boilerplate and add your own content. To help you build state definitions that are syntactically sound, use the component templates in the **Add Components** menu. See Dialog Flow Syntax for tips on setting variables and defining states.

Important:

Click **Validate** and check the logger widow ( ) as you write your dialog flow.



Dialog Flow Syntax

How Do I?	Use this	Example OBotML Markup

Set variables that persist the context across the entire dialog flow?	Within the `context` node, use the following syntax: `variablename: "variableType"`	`main: true`
		`name: "FinancialBotMainFlow"`
		`context:`
		`  variables:`
		`  accountType: "AccountType"`
		`  txnType: "TransactionType"`
		`  txnSelector: "TransactionSelector"`
		`  toAccount: "ToAccount"`
		`  spendingCategory: "TrackSpendingCategory"`
		`  paymentAmount: "string"`
		
		You can define variables as entities (like `AccountType` and `ToAccount` and as primitives (`paymentAmount: “string”`).

Define an error handler for your bot?	Define the `defaultTransitions` node.	`context:`
		`  variables:`
		`  iresult: "nlpresult"`
		`defaultTransitions:`
		`  next: "ImplicitTransitionDetected"`
		`  error: "MyErrorState"`
		`  actions:`
		`  unexpectedAction: "HandleUnexpectedAction"`
		
		See Conifguring the Dialog Flow for Unexpected Actions.

Define a variable that holds the value for the resolved intent?	Within the `context` node, define a variable that names the `nlpresult` entity. As its name implies ("nlp" stands for Natural Language Processing), this entity extracts the intent resolved by the Intent Engine. Nearly all of the reference bots declare `nlpresult` variables.	`main: true`
		`name: "FinancialBotMainFlow"`
		`context:`
		`  variables:`
		`  iResult: "nlpresult"`

How Do I?	Use this	Example OBotML Markup

Control the dialog flow based on the user input?	Typically (though not always), you'd define an `nlpresult variable` property for the `System.Intent` component that returns the result from the Intent Engine. See System.Intent.	In the following snippet from the FinancialBot dialog flow, the `System.Intent` component instructs the Dialog Engine to proceed based on the value returned by its `nlpresult` variable (`iResult`). As described in The Dialog Flow Structure, you can declare an `nlpresult` variable in the flow's `context` node to hold the resolved intent (`iResult:"nlpresult"`). The potential outcome, defined by the states named in the `actions` node, is also predicated on the second property defined for this component, `confidenceThreshold`. You can set this optional property against the probabilistic score given by the Intent Engine. This definition for the `System.Intent` component tells the Dialog Engine to move on to the next state that matches a resolved intent whose accuracy rate at parsing user data is at least 40% or higher (`confidenceThreshold: 0.4`). See The confidenceThreshold Property.
		
		`intent:`
		`component: "System.Intent"`
		`properties:`
		`variable: "iResult"`
		`confidenceThreshold: 0.4`
		`transitions:`
		`actions:`
		`Balances: "startBalances"`
		`Transactions: "startTxns"`
		`Send Money: "startPayments"`
		`Track Spending: "startTrackSpending"`

Equip my bot to handle unresolved intents?	Define a state for the `System.Intent`'s `unresolvedIntent` action. `unresolvedIntent` is an intent that we provide for you to track the messages that couldn't be resolved within the minimum confidence threshold. See Running Failure Reports to find out how to filter a quality report using this intent.	`unresolvedIntent: "unresolved"`
		`...`
		`unresolved:`
		`component: "System.Output"`
		`properties:`
		`text: "Sorry I don't understand that question!"`
		`transitions:`
		`return: "unresolved"`

How Do I?	Use this	Example OBotML Markup

Enable components to access variable values?	Use the `.value` property in your expressions (`${crust.value}`). To substitute a default value, use `${variable.value!\"default value\"}` . For example, `thick` is the default value in `${crust.value!\"thick\"}`.	`context:`
		`  variables:`
		`  size: "PizzaSize"`
		`  confirm: "YES_NO"`
		`  ...`
		`  confirmState:`
		`  component: "System.List"`
		`  properties:`
		`  options: "Yes,No"`
		`  prompt: "You ordered a ${size.value} pizza. Is this correct?"`
		`  variable: "confirm"`
		`...`
		
		Use the Apache FreeMarker default operator (`${variable.value!\"default value\"}`) if it's likely that a null value will be returned for a variable. You can use this operator wherever you define variable replacement in your flow, like the `value` definitions for variables used by system and custom components, or the variables that name states in a `transitions` definition. See Defining Value Expressions for the System.Output Component.

Save user values for return visits?	Within a state definition, add a variable definition with a `user.` prefix. See Defining User-Scoped Variables.	`checklastorder:`
		`  component: "System.ConditionExists"`
		`  properties:`
		`  variable: "user.lastpizza"`
		
		To find out more about user variables, see the dialog flow for the PizzaBotWithMemory reference bot.

Slot values?	Use the `System.SetVariable` , `System.List` , and `System.Text` components. When the `System.SetVariable` component can't access a value, use components `System.List` and `System.Text` to prompt user input. See System.SetVariable.	`askBalancesAccountType:`
		`  component: "System.List"`
		`  properties:`
		`  prompt: "For which account do you want your balance?"`
		`  options: "${ACCOUNT_TYPES.value}"`
		`  variable: "accountType"`
		`  nlpResultVariable: "iResult"`
		`  transitions: {}`

Exit a dialog flow and end the user session.	Use a `return` transition.	`printBalance:`
		`  component: "BalanceRetrieval"`
		`  properties:`
		`  accountType: "${accountType.value}"`
		`  transitions:`
		`  return: "printBalance"`

Flow Navigation

You can set the Dialog Engine on a specific path within the dialog flow by setting the transitions property for a state. Transitions describe how the dialog forks when variable values are either set or not set. They allow you to plot the typical route through the conversation (the “happy” flow) and set alternate routes that accommodate missing values or unpredictable user behavior.

The transition definition depends on your flow sequence and on the component.

To do this...	...Use this transition
Set a default sequence in the dialog flow.	To enable the Dialog Engine to move to the next state in the dialog, use an empty transition (`transitions: {}`) or omit a `transitions` definition altogether.
Specify the next state to be executed.	Setting a `next` transition (`next: “statename”`), tells the Dialog Engine to jump to the state named by the `next` key.

Terminate the conversation.	Defining a `return` transition terminates the user session at the state defined for the `return` key:
	`done:`
	`  component: "System.Output"`
	`  properties:`
	`  text: "Your ${size.value} ${type.value} Pizza is on its way."`
	`  transitions:`
	`  return: "done"`

Trigger conditional actions.	Define the `actions` keys to trigger the navigation to a specific state or an action belonging to a custom component that’s executed by a backend service. If you don’t define any action keys, then the Dialog Engine relies on the default transition or a `next` transition (if one exists). See Transitions to find out about the specific actions that you can define for the user interface components.
Handle component errors.	Set an `error` transition in case an error occurs when the component executes. The Dialog Engine will jump to the state that you define for the error key. If you don’t set an error transition, then the bot outputs the *Oops! I’m encountering a spot of trouble* message and terminates the session.

Conifguring the Dialog Flow for Unexpected Actions

When designing your dialog flow, you typically start modeling the “happy” flow, the path that the user is most likely to follow.

Scenario	Solution

Instead of tapping buttons, the user responds inappropriately in this situation by entering text.	To enable your bot to handle this gracefully, route to a state where the `System.Intent` component can resolve the text input, like `textReceived: Intent` in the following snippet from the CrcPizzaBot:
	
	`ShowMenu:`
	`component: System.CommonResponse`
	`properties:`
	`metadata: ...`
	`processUserMessage: true`
	`transitions:`
	`actions:`
	`pizza: OrderPizza`
	`pasta: OrderPasta`
	`textReceived: Intent`

Scenario	Solution

Users scroll back up to an earlier message and tap its options, even though they're expected to tap the buttons in the current response.	Adding an `unexpectedAction` transition to all of the states that process a user message handles situations where a user taps the button belonging to an older message, because this action tells the Dialog Engine to transition to a single state that handles all of the unexpected actions, such as the `HandleUnexpectedAction` state in the OBotML snippet above. You can use different approaches to create this state:
	• You can use the `System.Output` or `System.CommonResponse` component that outputs a message like "Sorry, this option is no longer available" along with a `return: "done"` transition to invalidate the session so that the user can start over. For example:
	`ActionNoLongerAvailable:`
	`  component: "System.Output"`
	`  properties:`
	`  text: "Sorry, this action is no longer available"`
	`  transitions:`
	`  return: "done"`
	• Using a `System.Switch` component, you can enable your bot to honor some of the request actions by transitioning to another state.
	Note:
	Depending on the factors involved in honoring the request, you may need to create a custom component to implement the routing.

Accessing Variable Values with Apache FreeMarker FTL

You can use Apache FreeMarker Template Language (FTL) to access variable values. The basic syntax for these value expressions is `${...}`. You can incorporate FTL into the property definitions for various components, such as System.SetVariable and System.Output .

> **Note:**
>
> As illustrated by the `text` and `rendered` metadata properties of the System.CommonResponse, you can also define the expressions using the if directive (`<#if>...</#if>`).

To do this...	...Do this

Read values from context variables.	Add the `value` property using dot notation:
	`${variablename.value}`
	For example:
	`${MyEmail.value}`

Read values from context variables defined by complex entities.	Use dot notation to add an additional property:
	`${variablename.value.property}`
	For example:
	`${MyMoney.value.totalCurrency}`
	If you use an expression like `${MyMoney}` in a `System.Output` component, you will see all the properties of the referenced currency JSON object.

Create a comma-delimited list of entity values that display as buttons that are specified by the `options` property.	Use this syntax:
	`${variablename.type.enumValues}`
	For example, for a list value entity like *AccountType* (whose *savings*, *checking*, and *credit card* values are constant and predefined for the user), you'd store these values in the `accountType` variable using `${accountType.type.enumValues}`:
	`accounts:`
	`component: "System.List"`
	`properties:`
	`options: "${accountType.type.enumValues}"`
	`prompt: "Which account?"`
	`variable: "accountType"`
	`transitions: {}`
	When the user taps one the buttons, the bot stores the corresponding value in the `accountType` variable.

To do this...	...Do this

Use built-ins for strings, arrays (sequences), numbers, and dates. See Apache FreeMarker Reference.	Follow the `value` property with a question mark (?) and the operation name:
	`${variable.value?ftl_function}`
	• string operations:
	`toLowercase:`
	`component: "System.SetVariable"`
	`properties:`
	`variable: "userstring"`
	`value: "${userstring.value?lower_case}"`
	`transitions: {}`
	• array operations:
	`setArrayCount:`
	`component: "System.SetVariable"`
	`properties:`
	`variable: "count"`
	`value: "${person.value?size?number}"`
	• number operations:
	`${negativeValue.value?round}`
	• time and date operations:
	`printDateFound:`
	`component: "System.Output"`
	`properties:`
	`text: "Date found is: ${theDate.value.date?long?number_to_date?string.short}"`

Concatenate FTL expressions.	String the operations together using a question mark (?): `${variable.value?ftl_function1?ftl_function2}`

User-Scoped Variables

When the conversation ends, the variable values that were set from the user input are destroyed. With these values gone, your bot users must resort to retracing their steps every time they return to your bot. You can spare your users this effort by defining user-scope variables in the dialog flow. Your bot can use these variables, which store the user input from previous sessions, to quickly step users through the conversation.

Unlike the session-wide variables that you declare in the context node at the start of the flow, you do not need to declare user-scoped. Any reference to a variable name that is prefixed with `user.` is treated as a user-scoped variable. As shown in the following dialog flow excerpt from the PizzaBotWithMemory dialog flow, these variables are identified by the `user.` prefix (such as `user.lastsize` in the `checklastorder` state). The `user.` variable persists the user ID. That ID is channel-specific, so while you can return to a conversation, or skip through an order using your prior entries on bots that run on the same channel, you can't do this across different channels like Facebook Messenger and Amazon Alexa.

```
metadata:
  platformVersion: "1.0"
main: true
name: "PizzaBot"
parameters:
  age: 18
context:
  variables:
    size: "PizzaSize"
    type: "PizzaType"
    crust: "PizzaCrust"
    iResult: "nlpresult"
    sameAsLast: "YesNo"
states:
  intent:
    component: "System.Intent"
    properties:
      variable: "iResult"
      confidenceThreshold: 0.4
    transitions:
      actions:
        OrderPizza: "checklastorder"
        CancelPizza: "cancelorder"
        unresolvedIntent: "unresolved"
  checklastorder:
    component: "System.ConditionExists"
    properties:
      variable: "user.lastsize"
    transitions:
      actions:
        exists: "lastorderprompt"
        notexists: "resolvesize"
  lastorderprompt:
    component: "System.List"
    properties:
      options: "Yes,No"
      prompt: "Same pizza as last time?"
      variable: "sameAsLast"
    transitions: {}
  rememberchoice:
    component: "System.ConditionEquals"
    properties:
      variable: "sameAsLast"
      value: "No"
    transitions:
      actions:
        equal: "resolvesize"
        notequal: "load"
...


  load:
    component: "System.CopyVariables"
    properties:
      from: "user.lastsize,user.lasttype,user.lastcrust"
      to: "size,type,crust"
    transitions: {}
```

Defining User-Scoped Variables

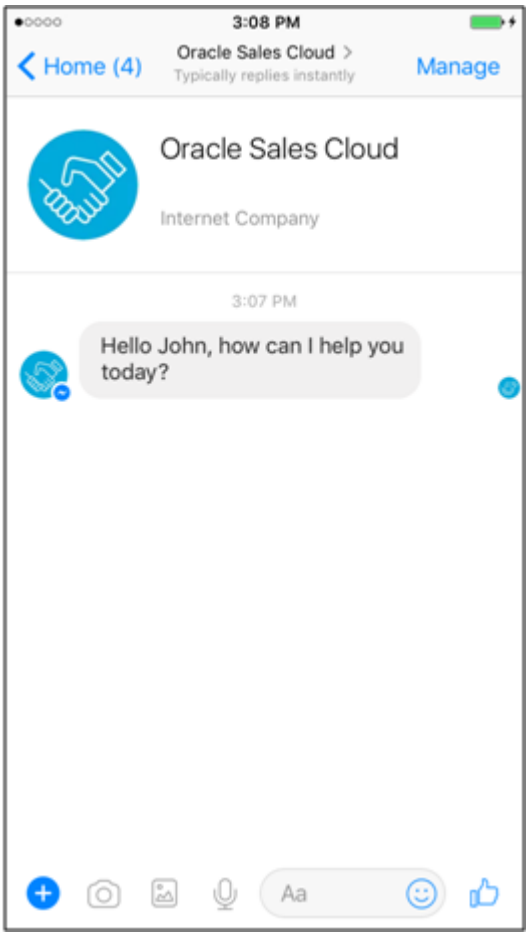
As with other variable definitions in your flow, you enable the components to access the value through value expressions like `"${user.age.value}"`. Using these expressions with the following built-in components, can among other things, set a value to the stored user value. See Built-In Components: Properties, Transitions, and Usage.

Component	Uses
System.SetVariable	Sets the stored user value.
System.ResetVariables	Resets a stored user value.
System.CopyVariables	Copies in the stored user value.
System.Output	Outputs the stored user value as text.
System.ConditionExists	Checks if the user-scoped variable is already in context.
System.ConditionEquals	Checks for the user-scoped variable.
System.Switch	Uses the stored value to switch from one state to another.

Getting the User Context

The `profile` property enables your bot to recognize a user's name, local, and local time. For example:

```
Greeting:
  component: System.Output
  transitions:
    next: Intent
  properties:
    text:  "Hello ${profile.firstName}, how can I help you today?"
```



Use these pre-defined variables to output context-specific for the bot user.

To do this...	Do this
Get the first name of the bot user.	`${profile.firstName}`
Get the last name of the bot user.	`${profile.lastName}`
Get the bot user's locale.	`${profile.locale}`
Get the user's time zone (offset in seconds).	`${profile.timezoneOffset}`

Test the Dialog Flow

Once you have a valid dialog flow, you can test your bot as a whole. Be sure to validate the syntax before you test the bot.

To test the dialog flow:

1. Click **Test** ( ).
2. Click **Bot**.
3. Enter an utterance and then click **Send**. Click to test an attachment response rendered by the `System. CommonResponse` component.

8
Localization

Even though NLP support is in English, you can still add multi-language support for your bot. Using resource bundles and autotranslation services, your bot can automatically translate the users messages that it receives and its own prompts and replies to and from English.

Resource Bundles

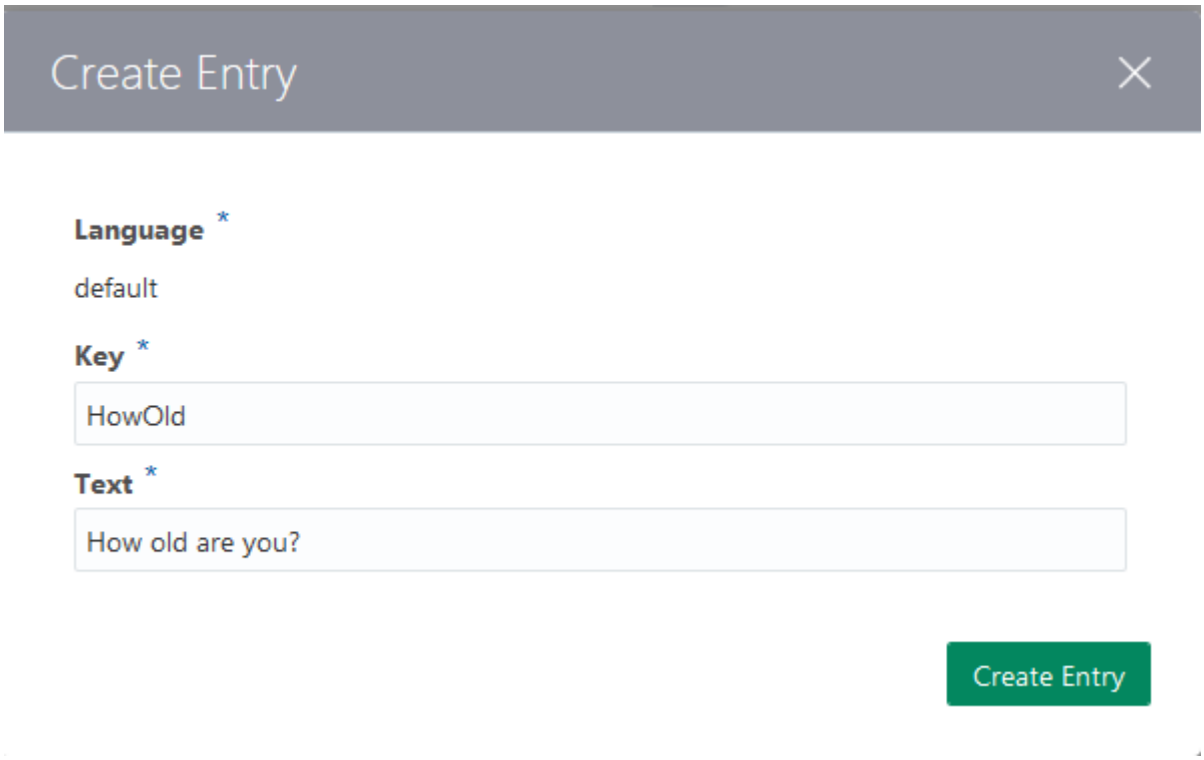
Resource bundles allow you to localize your bot based on the language set for messaging channel currently in use. They not only allow your bot to output messages in the user's language, but in the user's dialect as well. When you don't want to rely on the text provided by the translation service, and instead want to control the wording for your bot's responses in one or several languages, you can opt for resource bundles.

Create Resource Bundle

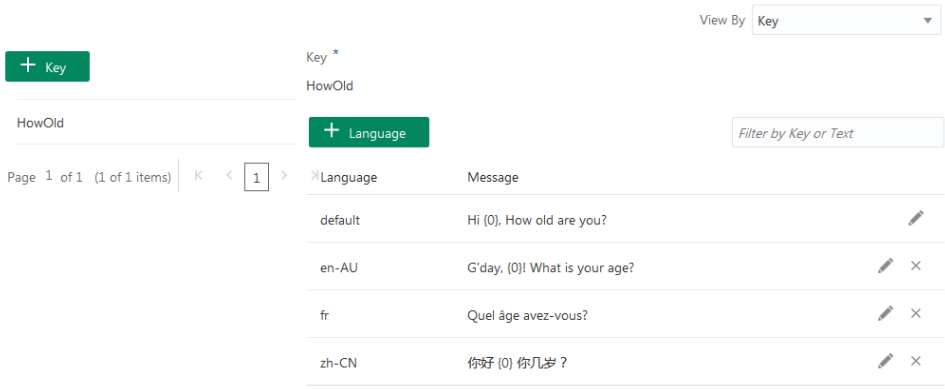
You define a single bundle for each bot that's made up of various keys that identify the output text that needs to be translated.

To create a resource bundle:

1. Click **Resource Bundle** in the left navbar ().
2. Click **Add Bundle**.
3. Enter the key and the and its text . For example, to localize the user prompt, *How old are you?*, you'd enter *HowOld* in the **Key** field and then *How old are you?* in the **Text** field .

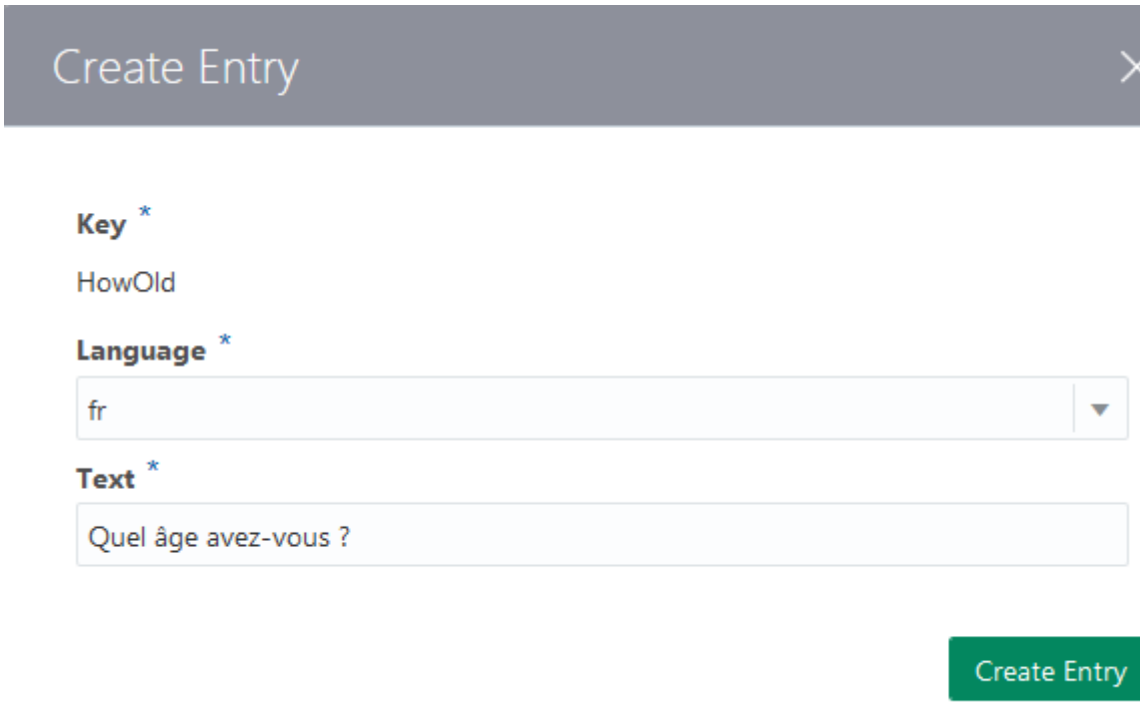


4. Click **Create Entry**.
5. By default, the language for your first key is English. To add a foreign language version of the string, click. **Add Language**.



6. Complete the Create Entry dialog:
   - **Language**—Add an IETF BCP 47 language tag like `fr` for French, `de` for German, or `en-US` for U.S. English.
   - **Text**—The output string. For example, for a French translation (`fr`) of the *HowOld* key, you'd add a string like *quel âge avez-vous ?*

> **Note:**
>
> If the bot can't match the language set for the browser with a language tag defined in the bundle, it defaults to a less-specific tag (if one is available). For example, it uses `fr` (a subtag) if the bundle has no entry for `fr-CA`. If none of the entries match the browser's language, the bot uses the default entry, English. For more information on this fallback to the most generic entry, see Resource Bundle Entry Resolution

- 

7. If you want to translate other strings, click **Add Key** to create another entry in the resource bundle.
8. Reference the resource bundle in the in the dialog flow.

> **Tip:**
>
> You can define the entity prompts as a resource bundle. See Create Entities.

Reference Resource Bundles in the Dialog Flow

To set the output for a built-in component, you need to add a resource bundle context variable and then reference both it and the message key. In the following OBotML snippet for a pizza bot, the resource bundle is declared as the variable, `rb`, in the

context section. Further down, value expressions define the text property for the `System.Output` components reference the `rb` variable and the keys, `WhatType` and `OnTheWay`. The first outputs a simple string and the other uses dynamic values.

```
context:
  variables:
    rb: "resourcebundle"
...

pizzaType:
  component: "System.Output"
  properties:
    text: "${rb('WhatType'}" # rb refers to the variable, WhatType is the key to the
message in the resource bundle.
  transitions: {}
...

done:
  component: "System.Output"
  properties:
    text: "${rb('OnTheWay',size.value,type.value)}" # size.value and type.value are
the arguments for the 'OnTheWay' message code.
  transitions:
    return: "done"
```

For simple messages, you can also reference resource bundles using dot notation (`${rb.WhatType}`).

> **Tip:**
>
> To test your resource bundles using the Tester, set your browser to another language.

Enabling Complete Translation When Using Resource Bundles

If you're returning the user's language from the browser, then simply setting the resource bundle variable and then referencing both it and the message key in an output component is all you need to do. Keep in mind that using this approach requires users to first enter something in English (like "Hello, Pizzabot!"). To start the session in the user's language, you need to enable the translation service for the bot and configure the dialog flow accordingly. The following snippet shows this hybrid approach, which enables your bot to detect the user's language. After the input is translated and the intent is resolved to English, the resource bundles handle the rest. In the following snippet, the `rb` context variable is set, but in this PizzaBot, it's accompanied by another variable called `translated`. Because both the `System.DetectLanguage` and `System.TranslateInput` components are positioned before the `System.Intent` component, they enable the initial user input to be translated into English before it can be used by the `System.Intent` component and resolved to one of the intents.

```
metadata:
  platformVersion: "1.0"
main: true
name: "PizzaBot"
parameters:
  age: 18
context:
```

```
  variables:
    size: "PizzaSize"
    type: "PizzaType"
    crust: "PizzaCrust"
    iResult: "nlpresult"
    rb: "resourcebundle"
    translated: "string" # holds the user's text that's translated into English.
states:
   # add DetectLanguage and TranslateInput components
  detect:
    component: "System.DetectLanguage"
    properties: {}
    transitions: {}

  # translate the user text and store it in the translated variable
  translate:
    component: "System.TranslateInput"
    properties:
      variable: "translated"
    transitions: {}

  intent:
    component: "System.Intent"
    properties:
      variable: "iResult"
      sourceVariable: "translated" # this variable now would holds the translated
text
      confidenceThreshold: 0.4
    transitions:
      actions:
        OrderPizza: "resolvesize"
        CancelPizza: "cancelorder"
        unresolvedIntent: "unresolved"
  ...
```

Resource Bundle Entry Resolution

To find out the users language, you can add a `${profile.locale}` to the dialog flow definition. Bots will look up the right message based on the user's locale. For example, if the `${profile.locale}` returns `en-AU-sydney` as the value for the `languageTag` variable that's set in the context section, Bots returns the bundle entry by first searching for an exact match. If it can't return a match, it broadens its search. In this case, Bots does the following to localize the output as Australian English:

1. Searches the bundle using a language-country-variant criteria. In this case, it searches for `en-AU-sydney`.
2. If it can't find that, it searches the bundle by language and country (`en-AU`).
3. Failing that, it broadens its search for language (`en`).
4. If it can't locate any entries, then it returns the default language, which is English (`en`).

Autotranslation

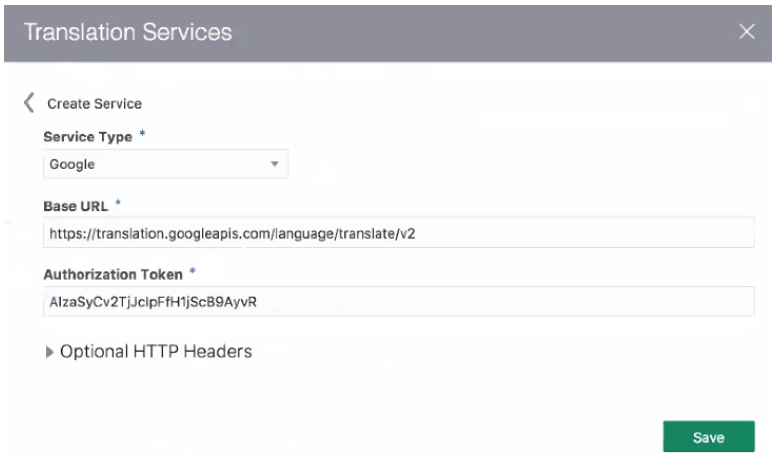
Autotranslation uses services like Microsoft Translator and the Google Translation API to enable the built-in components like `System.Text` and `System.Output` to output their prompts in the user's language.

When a user enters a non-English request or response, the translation service allows the bot to convert this input to English. Once it's translated, the NLP engine can resolve it to an intent and match the entities. Using both a translation service and an OBotML definition that includes the System.DetectLanguage and System.TranslateInput components, you can enable your bot to automatically detect the user's language and translate your bot's messages.

Enable Autotranslation

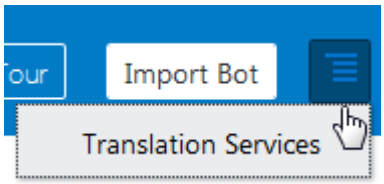
To enable your bot to use autotranslate:

1. First, configure a translation service for your instance of Intelligent Bots. To do this, enter the URL and Authorization token for the Microsoft Translator service or the Google Translation API in the Translation Services dialog.

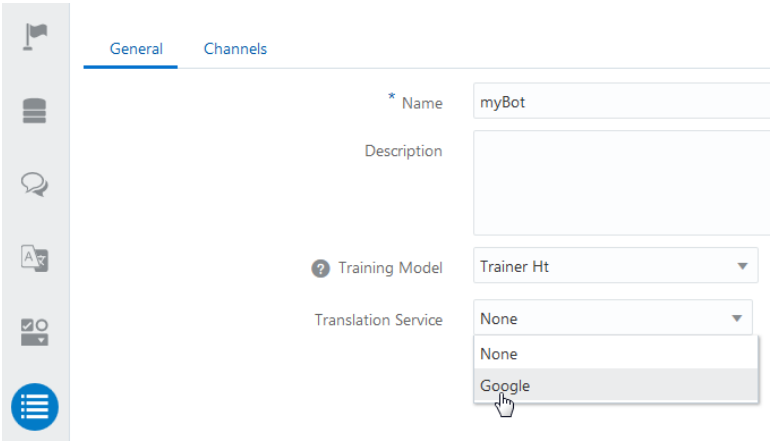


Refer to the documentation for Microsoft Translator and Google Translation API to find out how to get the URL and access token.

You can open this dialog from the menu on the landing page or from the Settings page.



2. Next, click **Settings** in the left navbar and then choose a translation service for your bot.



3. Finally, configure the dialog flow:

   a. Add `autoTranslate: "boolean"` to the `context` node. This variable is common to all bots (or at least the ones that use translation services). As such, you can't change its name or define it as string, another type of primitive, or an entity. You can override the autotranslated output generated for the System.Output, System.Text, and System.List components when you set their `translate` property to `false`.

   > **Note:**
   >
   > Do not set the `autoTranslate` variable to `true` if you're translating text using a resource bundle.

   b. Generally, you'd first add a state with a `System.Intent` component at the beginning of the `states` node. But since the NLP engine can only recognize English, you need to begin the states node with a set of language-specific components and variables to translate the user input so that it can be resolved by `System.Intent` component. The first of these states uses the `System.SetVariable` component. As shown in the following snippet, its `variable` property is defined by the `autoTranslate` context variable. To enable translation, it's set to `true`.

   ```
   setAutoTranslate:
     component: "System.SetVariable"
     properties:
       variable: "autoTranslate"
       value: true
     transitions: {}
   ```

   c. Next, add the System.DetectLanguage component:

   ```
   detect:
     component: "System.DetectLanguage"
     properties: {}
     transitions: {}
   ```

   d. Add the System.TranslateInput component:

   ```
   translate:
     component: "System.TranslateInput"
     properties:
   ```

```
        variable: "translated"
      transitions: {}
```

e. Finally, add the System.Intent component. Set `sourceVariable` to hold the translated input.

```
  intent:
    component: "System.Intent"
    properties:
      variable: "iResult"
      sourceVariable: "translated" # this variable holds the English
translation of the user input.
      confidenceThreshold: 0.4
```

> **Important:**
>
> While the **Add Components** menu adds template state nodes for these translation components, it doesn't insert the `autoTranslate: "boolean"` variable into the `context` node. You'll need to add it yourself.

The following segment shows the PizzaBot equipped for autotranslation. Note that along with the `autoTranslate` variable, this definition also includes a variable that stores the translated output called `translated` (`translated: "string"`) in the context node. The variable property for the `System.TranslateInput` component names this component, as does the `sourceVariable` property for the `System.Intent` component. For example, the `System.TranslateInput` component would store its English translation ("I want pizza") in this variable when a user enters "je voudrais commander une pizza." Because `sourceVariable` names `translated`, it holds "I want pizza," which the `System.Intent` component can resolve to one of the intents.

```
metadata:
  platformVersion: "1.0"
main: true
name: "AutoTranslatePizzaBot"
parameters:
  age: 18
context:
  variables:
    size: "PizzaSize"
    type: "PizzaType"
    crust: "PizzaCrust"
    iResult: "nlpresult"
    autoTranslate: "boolean"
    translated: "string"
states:
   setAutoTranslate:
    component: "System.SetVariable"
    properties:
      variable: "autoTranslate"
      value: true
    transitions: {}
  detect:
    component: "System.DetectLanguage"
    properties: {}
    transitions: {}
  translate:
    component: "System.TranslateInput"
    properties:
```

```
    variable: "translated"
  transitions: {}
intent:
  component: "System.Intent"
  properties:
    variable: "iResult"
    sourceVariable: "translated"
    confidenceThreshold: 0.4
```

9
Components

Components give your bot the functionality that lets it interact with users and carry out their requests.

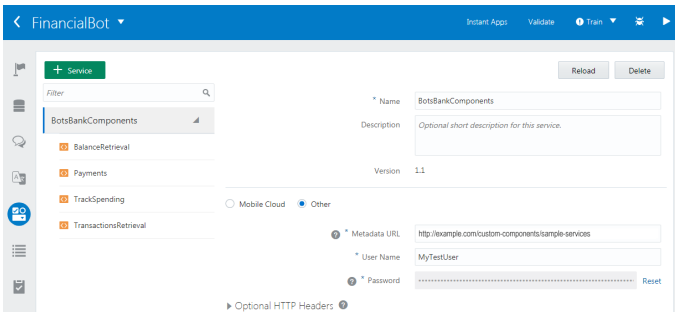
Each state within your flow calls a component to perform actions that can range from basic interactions like taking user input and outputting response text to some service-specific action like fulfilling an order or booking a flight. We provide a set of built-in components that support basic actions like setting variables, allowing OAuth, and enabling user input. If your bot calls for a specific action that's outside of these functions, you'll need to use a custom component. These components let your bot call REST APIs that implement business logic and channel-specific rendering.

The Custom Component Service

Configuring a custom component service makes custom components available to your bot's dialog flow.

A custom component service defines name of the custom component service implementation that provides the bot with its custom components. This configuration also includes that URL that invokes this service and the basic authentication settings that allow the bot to access the service itself. Each bot can have one or more of these custom component services configured for it. By configuring a custom component service, you allow Bots to query the service for its metadata and display this information in the custom components registry.

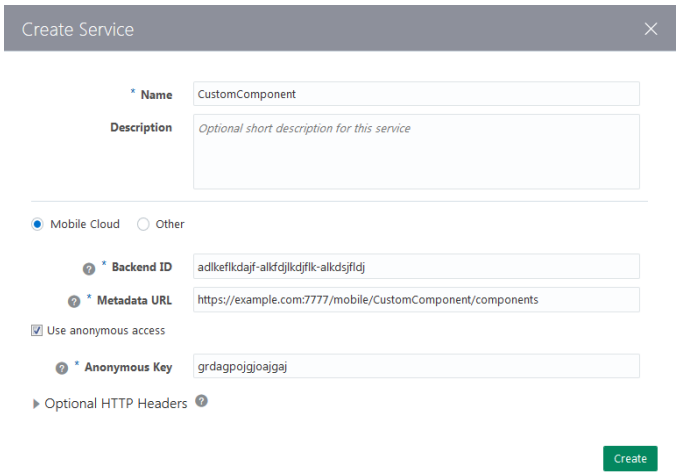
> **Note:**
>
> Every custom component that you've declared in your OBotML definition needs a corresponding custom component service configuration. In other words, your bot can't work without this configuration, which allows Bots to call the custom code service implementation that defines the components.
>
> Refer to the components page ( ) when you define the dialog to ensure the component names and properties are correct.



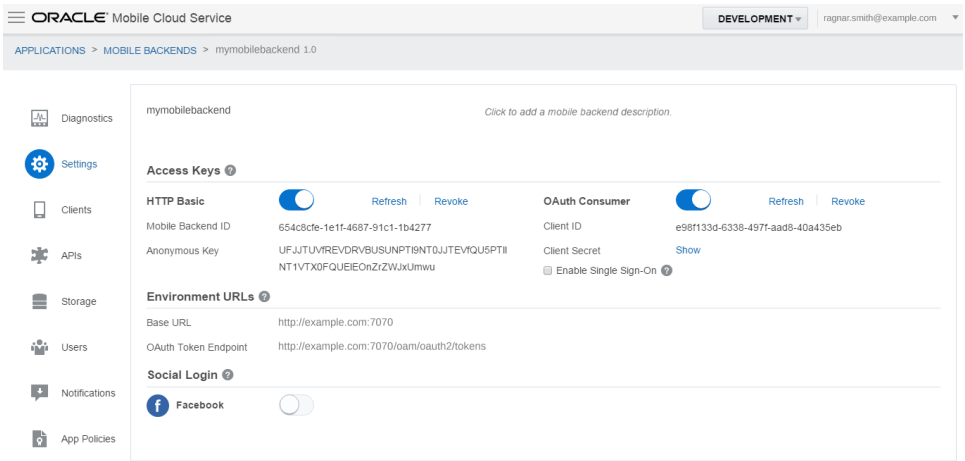
Create a Service

The Custom Component Service authenticates the bot with the service using basic auth. You can implement this on your authentication mechanism, or if you use the node.js SDK, your bot can authenticate through a mobile backend. To create this service:

1. In the left navbar, click **Components** ( ).
2. Click **Add Service** to open the Create Service dialog.



3. Add the name for the custom component service and an optional description.
4. Choose an authentication option:
   - **Mobile Cloud**—For authentication handled by a backend in OMCe. This is the default setting. If your service is handled by a backend, then you need to reference the Settings page for the backend that hosts the API that implements the Custom Component Service. Backend Authentication and Connection Info in *Developing Applications with Oracle Mobile Cloud Enterprise* Describes this page.



Option	Description
Backend ID	The unique identifier assigned to a mobile backend. This ID is passed in the REST header of every call made from the bot.
MetadataURL	The URL that points to the endpoint of the components service. This URL points to the root, which means it should always be appended with `/components`.
Use Anonymous Access	Select this option if the component service allows anonymous login. If you choose this option, enter the Anonymous Key, a unique string that allows your app to access anonymous APIs without sending an encoded username and password combination. The Anonymous Access Key is passed to instead. **Tip**: Click **Show** to reveal the Anonymous Key in the Settings page. If the component service requires a login (meaning no anonymous access), enter the username and password.

- **Other**—For non-backend authentication. For this option, define the following options:

Option	Description
Metadata URL	The URL that points to the endpoint of the components service. This URL points to the root, which means it should always be appended with `/components`.
Username	The username for the service.
Password	The service's password.

5. If the service requires specific parameters, click **Add HTTP Header** and then define the key-value pairs for the headers.
6. Click **Create**.

   The Components page is populated with the component name and properties, which you can then reference in the dialog flow definition. Remember: unlike the built-in components, custom components do not begin with `System`. For example:

```
checkage:
   component: "AgeChecker"
   properties:
     minAge: 18
   transitions:
     actions:
       allow: "crust"
       block: "underage"
```

How Do Custom Components Work?

Your bot uses custom components when it needs to return data, execute some kind of business logic, or render channel-specific UI components like the carousel in Facebook Messenger.

Like the built-in components, the custom components are re-usable units of work that you define within each state node of your dialog flow. But unlike the built-in components, custom components perform actions that are specific to your bot. They execute functions that the system components can't. While the FinancialBot uses system components for generic tasks like setting variables and outputting text, it uses custom components for the operations that are unique to banking transactions, such as returning account balances (`BalanceRetrieval` in the following state node, `printBalance`).

```
printBalance:
    component: "BalanceRetrieval"
    properties:
      accountType: "${accountType.value}"
    transitions:
      return: "printBalance"
```

Custom components don't reside within Bots. Their functionality is provided through backend services that are accessed through calls made to, and returned from, a REST service called the Component Service. As the Dialog Engine enters a state in the dialog flow, it assesses the component. When it encounters one of the built-in components (noted by `System.`), it executes one of the generic tasks described in Built-In Components: Properties, Transitions, and Usage. When the Dialog Engine discovers a custom component, however, it calls the Component Service, which hosts one or more custom components.

The Component Service is like a shim. It first finds and then invokes the custom component on behalf of the Dialog Engine. When a custom component is invoked, it can pass input parameters to a backend service and return the result. The Dialog Engine then resumes, moving on to the next sate in the dialog flow (or to the state dictated by the action described in the returned JSON payload).

The Component Service assists the bot through two methods: GET and POST. The GET method returns the metadata for all of the components hosted by the Component Service. This is a design time call, one that returns the names of the components along with their properties and actions that you include in your dialog flow definition. At runtime, the POST method invokes the component named in the state definition.

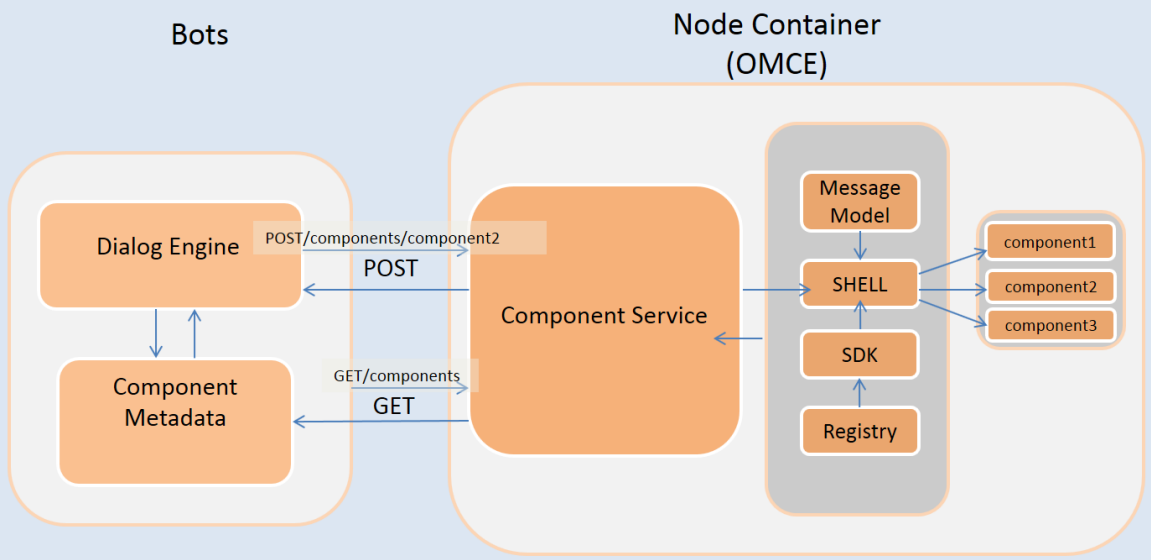
The JSON payload of the call made by the Dialog Engine includes input parameters, variable values, user-level context, and the user's message text. When the component gets this input from the Component Service, it mutates the variable values, and then returns the call. The Dialog Engine parses the returned payload and proceeds.

The Component Service

The Component Service doesn't reside within Bots, but is instead hosted in a separate Node container. Because the Component Service is a REST service, you can implement using any language.

As pictured here, the Node container can be part of OMCE, but it can be part of any other REST infrastructure. If you opt for OMCE as the container for your custom components, you can integrate them with remote services using various connectors. Because they are implemented as custom code APIs, they can access the OMCE platform APIs (such as the Analytics API) through the OMCE SDK. There's another advantage to implementing the Component Service in OMCE: you can get it up and running with minimal coding using the Bots SDK because it provides you with a starter application that gives you everything you need. To find out about the artifacts included in the SDK, see How Do I Implement the Component Service in OMCe?.

> **Note:**
>
> You can still integrate them with remote services if you use another Node container, but keep in mind that direct REST calls can give rise to additional concerns and tasks. With no backend to manage the connection, for example, you'll need to update the code whenever the connection changes.



The Shell

The Shell routes the GET and POST requests. It produces a list of components in response to the GET call made by Bots when you register a Component Service. The Shell also invokes the component using the component name that's appended to the POST call (`POST uri/components/{ComponentName}`). To respond to these requests, the Shell component references a file in the Registry component that maps the component names to their corresponding JavaScript implementation files.

The Registry

The Registry component maps each component to its implementation.

Within the `Registry.js` file, a JSON object definition surfaces the components to the Shell. Each component is described by a name-value pair in which the *name* is the name of the component (like `'Balance Retrieval'` in the following import statement) and the *value* is a return function with a reference to the JavaScript module location relative to the `Registry.js` file (`./`). In this snippet, the three components, BalanceRetrieval, TrackSpending, and Payments are custom components, each of which map to a separate JavaScript module. The `require` function includes these separate modules in the `Registry.js` file.

```
'use strict';

module.exports = {

'BalanceRetrieval':
```

```
require('./banking/balance_retrieval'),

'TrackSpending':
 require('./banking/track_spending'),

'Payments':
require('./banking/payments')}
```

> **Tip:**
>
> Declare strict mode (`'use strict'`) at the beginning of the `Registry.js` file to safeguard against the inadvertent creation of global variables from erroneous user input. The strict mode improves error checking by throwing exceptions for errors that would otherwise occur silently, like values set on a read-only property.

Because the `Shell.js` component assumes that it shares the same file location as the `Registry.js` , the `Shell.js` file uses the following import statement:

```
var registry = require('./registry');
```

Remember that you don't need to edit the `Shell.js` file. You just need to make sure that it's in the same directory with the `Registry.js` module (and if you're using , the `SDK.js` module as well).

Component Modules

Each component is written as JavaScript module. If you're writing one of these modules, then you need to include two functions that mirror the GET and POST calls in the Component Service REST contract: `metadata` and `invoke`. You also need to conclude the module with the callback function,`done`.

The `metadata` function provides the component descriptions that you use when you define your dialog flow. It includes a name (which must be unique), and the names and types of the input parameters that it expects. It also includes the actions supported by the component. For example:

```
metadata: () => ({
    "name": "helloWorld",
    "properties": {
      "properties": {
        "name": {
          "type": "string",
          "required": false"
      }
    },
    "supportedActions": ["nameFound", "nameNotFound"]
  }),
```

The `invoke` function executes the REST call. It includes two arguments: `conversation`, which is a reference to the SDK and `done`, a callback invoked by the component when it has finished processing. The `done` function tells the Shell to create the component's response payload and send it back to the bot.

> **Important:**
>
> Always include the `done()` callback at the end of each component. The component can't send its response without it and as a result, the bot will time out.

```
module.exports = {
    metadata: () => (
    {
        "name": "BalanceRetrieval",                    },
        "properties": {
                "accountType": { "type": "string", "required": true }
            },
            "supportedActions": []
        }
    },
...

invoke: (conversation, done) => {
      var accountType = conversation.properties().accountType;
...

      var accounts = AccountService.account(accountType);

...
      done();

         }
};
```

Along with the component name and properties that get returned during design time by the `invoke` function, this code sample shows how the `invoke` function uses one of the SDK's helper methods (`conversation.properties`) to retrieve the value of the `accountType` from the payload of the POST request. With the value retrieved, the custom code can use it to call connectors or other APIs running in OMCe.

> **Note:**
>
> The `invoke` function enables access to the OMCE platform APIs using the `conversation.OracleMobile` object. To find out how to instrument the custom component code to call the Analytics API (`conversation.oracleMobile.analytics.postEvent`), see Setting up the PizzaBot Custom Component.

The SDK

If you implement the Component Service with OMCe, you can also leverage the SDK, whose helper methods enable the components to access the context of a bot's request messages, which can be comprised of elements that describe the variable values, the language processing results, the extracted entities, and any input parameters that have been defined for the component. The SDK also enables the components to return a response to the bot.

The Shell passes the SDK to the custom components with each call to the `invoke` function. To access the SDK's methods, the `invoke` function uses an argument called `conversation`, which is automatically passed with each request along with the essential `done ()` callback that signals the Shell when the component has completed its work.

```
invoke: (conversation, done) => {

    var listdata =
        "item1, item2, item3";

    conversation.variable("listDataVar", listdata);
    conversation.transition();
    conversation.keepturn(true);

    done();
```

The Message Model

The Message Model is a utility class that creates and validates the message structure. An instance of this class is instantiated with the payload that represents the message so that the message can be parsed and validated.

> **Note:**
>
> Version 1.1 of the Bots SDK lets you leverage the Conversation Message Model (the CMM), a framework that defines various platform-agnostic templates for the messages sent between the bot and its users. Not only does the CMM allow your bot to output messages as loops of cards that have actions configured for both the images and buttons that display within each of them, it also gives your bot other capabilities as well, such displaying context- specific messages and allowing users to share locations or upload audio, video, file, or image attachments. The Bots SDK documentation describes how you integrate the CMM into the code for your custom components, the methods for different types of message formats, and how you can upgrade your custom component service to use the CMM.

How Do I Implement the Component Service in OMCe?

While you can use the Shell and Registry components in any REST framework that produces a JSON object from the incoming request, you can only use the SDK's helper methods if you implement the Component Service in OMCe. To use the SDK and get ready-made versions of the Shell and Registry, you need the Bots SDK.

Accessing the Bots SDK

You can get the Bots SDK (`omce-bots-sdk-<version_number>.zip`) from the Oracle Technology Network's Oracle Mobile Cloud Enterprise download page. You can also access this page by clicking **Downloads** in the left navbar.

After you unzip the file, open the `api_implementation` folder. It contains the following artifacts that you modify to build your service. It includes JavaScript files for the Shell, Registry and the SDK (`shell.js`, `registry.js`, and `sdk.js`). It also includes the following:

- `mcebots.js`—Contains the generic component logic. You copy and paste this into your own component service.
- `package.json`—Contains the `node.js` module dependencies required for the project's `package.json` file.
- `mcsbots.raml`—A template for creating the OMCe custom API.

Creating the Component Service in OMCe

You can find out more in Custom APIs in *Developing Applications with Oracle Mobile Cloud Enterprise*, but the process in terms of the custom components is as follows:

1. Define the GET and POST endpoints—You can define these endpoints on your own, or use the starter RAML template (`mcebots.raml`).
   a. In OMCe, click **New API**.
   b. Enter the API name, a description, and a short description.
   c. Drag `mcebots.raml` into the dialog and then click **Create**.
2. If you want to enable anonymous access, click **Security** in the left navbar and then switch off **Login Required**.
3. Click **Save**.
4. Download the JavaScript scaffold:
   a. Click **Implementation** in the left navabr.
   b. Choose **Download JavaScript Scaffold**.
   c. Unzip the scaffold file. This file contains the following:
      - The component service file—This file, which is named after your API, contains the REST endpoints defined for OMCe custom code APIs.
      - `package.json`—The project configuration file. It includes a list of module dependencies.
5. Implement the Custom Component:
   a. Within the scaffold file, add a directory with the SDK, Registry, MessageModel and Shell modules.

      > **Note:**
      >
      > The Shell, Registry, MessageModel and SDK components must reside within the same directory as the Component Service.

   b. Implement the scaffold's JavaScript to add the custom component logic. To do this, you're going to replace most of the contents of the component service file with those of the `mcebots.js` file from the Bots SDK:
      i. Open the component service file in the JavaScript editor of your choice.
      ii. Note the `service.get` function URI. It looks something like `/mobile/custom/MyFirstComponentService/components`.
      iii. Delete all of the contents of the file except for the comments at the top of the file.

iv. Open the `mcebots.js` file and then copy its contents to the component service file.

v. Replace the value of `const apiURL = '/mobile/custom/bots/components';` with the value of the `service.get` function. For example, `const apiURL = '//mobile/custom/MyFirstComponentService/components';`.

vi. Point to the `shell.js` file. Because the component service file and the directory containing the Bots SDK artifacts (which includes the `shell.js` file) are not located in the same folder, you need to modify the Shell variable's `./shell` parameter to reference the location of the `shell.js` file. For example, if `shell.js` resides in a directory called `js`, you would change the default parameter from this:

```
var shell = require('./shell')();
```

to this:

```
var shell = require('./js/shell')();
```

vii. Save the file.

c. Edit the `package.json` file in the scaffold file with the Bot SDK dependencies in the `package.json` file from the Bots SDK:

i. Open the Bots SDK's `package.json` file in the text editor of your choice and then copy and paste its `dependencies` definition to a clipboard:

```
  "dependencies": {
    "joi": "^9.2.0"
  },
```

ii. In the scaffold's `package.json` file, paste the definition on its own line, one directly after the `"main":` attribute.

6. Create the custom component module by creating a JavaScript file. This file includes the `metadata` and `invoke` functions described in Component Modules. The scaffold for the file looks like this:

```
"use strict";

module.exports = {
        metadata: () => (
        {
            "name": "sample.hello",
            "properties": {
                "name": { "type": "string", "required": true }
             },
             "supportedActions": []
        }
    ),
    invoke: (conversation, done) => {
        const name = conversation.properties().name ?
conversation.properties().name : '';
        conversation.reply({ text: 'Hello ' + name });
        conversation.transition();
        done();
    }
};
```

Use the functions exposed by the SDK to allow interactions with the bot's request payload. See The SDK Helper Methods.

> **Important:**
>
> All custom component files must reside within the same directory. Also, make sure that all of your component files all have the `.js` extension.

7. Edit the `registry.js` file with the name and location the component file.
8. Install the node modules.
9. Package the scaffold and upload the node project to OMCe.
10. Associating APIs with a Backend in *Developing Applications with Oracle Mobile Cloud Enterprise* and then test your API.
11. Register the component service with Bots so that it can be discovered by the Dialog Engine. To do this first click **Components** () in the left navbar and then **Add Service**. Complete the dialog by adding a name, selecting **Mobile Cloud**, and then by providing the following:

    - **Backend ID**—This value is generated when you create a mobile backend. It's listed on the Settings page.
    - **Metadata URL**—The is custom API URL, which is displayed in the Overview panel of the API Designer when you click the GET method in the

      > **Important:**
      >
      > Be sure to append this URL with `/components` so that it can return the component information in the Bot Builder's Components page.

    - The user name and password. If you selected **Use anonymous Access**, you need to provide the Anonymous Key. This value is generated when you create a backend. It's displayed on the Settings page for the backend that manages your API.

    You're now ready to add the custom components to your OBotML definition.

10
Channels

To introduce your bot to the users of these services, you need to configure a channel.

We provide a channel for Facebook Messenger and a generic channel called Webhook that you can use for other messaging services. Your bots are limited to messaging services; using one of our SDKs, you can integrate them in web pages, or the Android and iOS messaging platforms.

> **Tip:**
>
> Check out the Developer Resources to find out about configuring other channels and setting up different types of chat clients and the sample chat server.

Your bot can run on any messaging service that supports webhooks, calls that allows real-time messaging without polling. You don't need to implement a webhook to get your bot running on Facebook Messenger: all you need to configure the Facebook channel are the keys that are generated by both Facebook and Bots. Setting up the Webhook channel for other messaging services require you to perform a few more tasks in addition to the channel configuration, like setting up an HTTP server with a webhook for sending and receiving your bot's messages.

Running Your Bot on Facebook Messenger

You'll need the following to configure the channel for Facebook Messenger:

- A Facebook Page
- A Facebook App
- A Page Access Token
- An App Secret ID
- The webhook URL
- A Verify Token

> **Note:**
>
> You also need a Facebook Developer account.

To run your bot on Facebook Messenger, you need to set up a Facebook page and a Facebook App. You can find out more about this from the Facebook Messaging Platform documentation, but in a nutshell, the Facebook page hosts your bot. Users chat with your bot through this page when they use chat window in a desktop browser. When they use a mobile device, users interact with your bot directly through Facebook

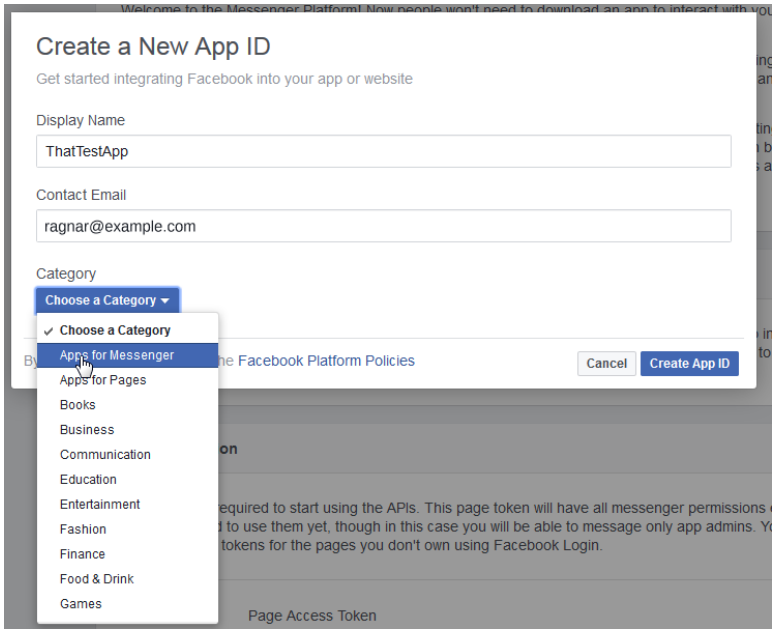
Messenger itself. In this scenario, the Facebook App allows your bot to get the messages that are handled by Facebook Messenger.

To create a Facebook Messenger channel, you need artifacts that are generated by both Bots and by Facebook Messenger. From Bots, you'll need the webhook URL that connects your bot to Facebook messenger and the Verify Token that enables Facebook Messenger to identify the bot. From Facebook Messenger, you'll need the Page Access Token and the App Secret ID. Because you need transfer these artifacts between Bots and Facebook Messenger, you'll need to switch between these two platforms as you configure the channel.

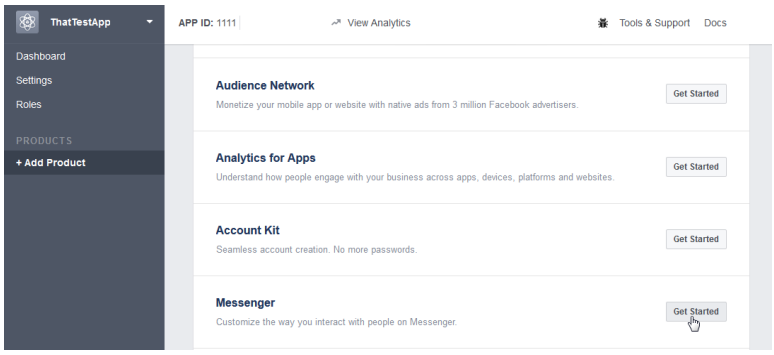
Step 1: Set Up Facebook Messenger

Start off by generating the App Secret and the Page Access token in Facebook Messenger.

1. Log into your Facebook developer's account.
2. Create a Facebook page that hosts your bot. The description, images, and cover page you add to the page will identify your bot to its users.
3. Next, create the Facebook app that you'll link to this page. Because this is a Messenger app, choose **Apps for Messenger** and then click **Create App ID**.

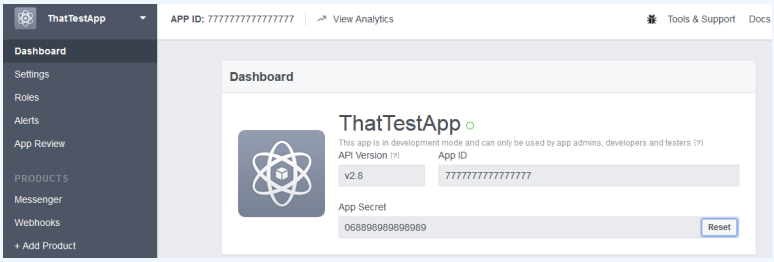


If you didn't choose the **Apps for Messenger** option in this dialog (for example, if you're creating a test app), then click **Add Product** in the left navbar, choose **Messenger** from the Product Setup page, and then click **Get Started**.

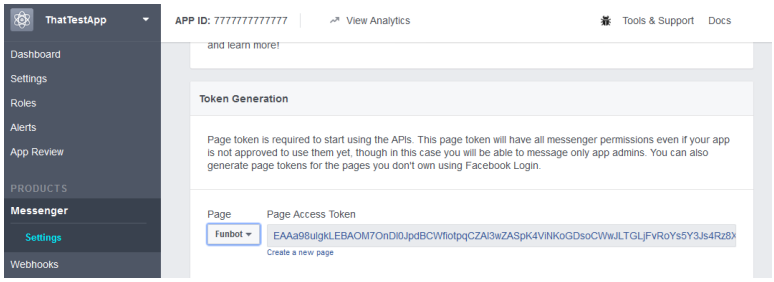


Note:

You'll need the App Secret to complete your Facebook channel configuration in the Bot Builder.



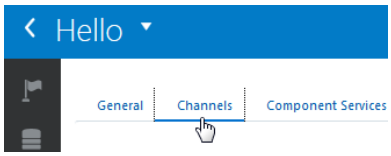
4. In the Dashboard for your app, generate the Page Access Token by selecting your Facebook page. You'll use this token, which gives your Facebook App access to Facebook's Messaging API, to complete your channel definition.



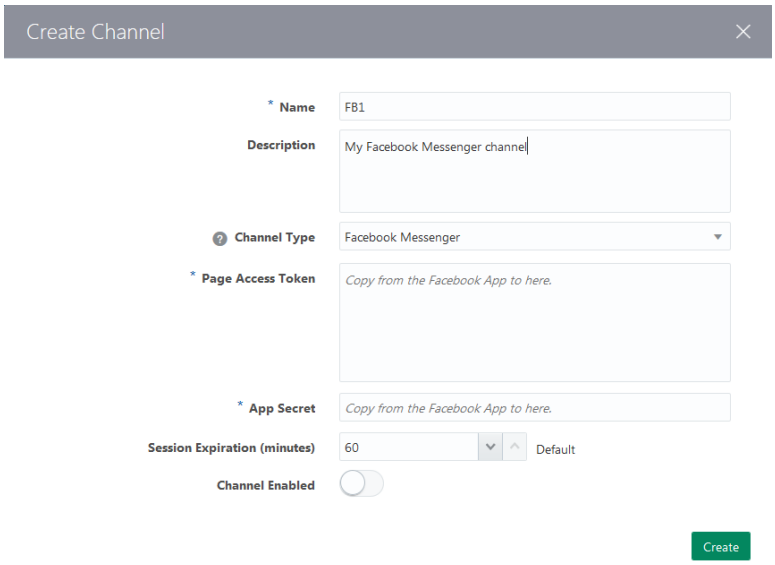
Step 2: Add the Facebook Keys

Complete the Create Channel dialog by providing the Page Access Token and App Secret keys from Facebook.

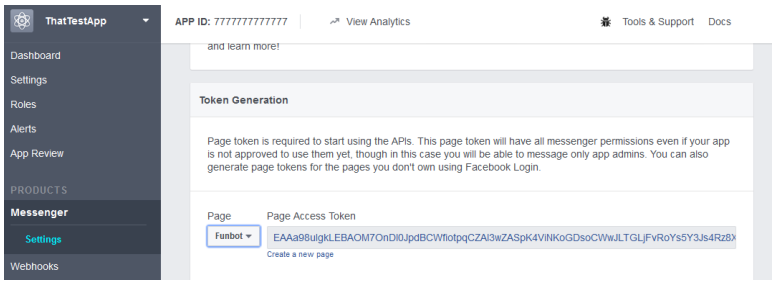
1. In Bots, click **Settings** ( ) in the left navbar and then choose the Channels tab.



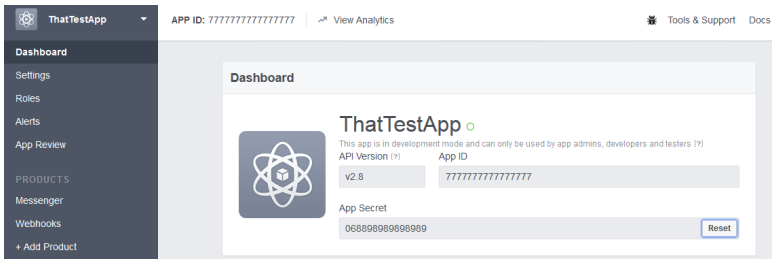
2. Next, click **Add Channel** to open the Create Channel dialog.
3. Give your channel a name.
4. Choose **Facebook Messenger** as the channel type.



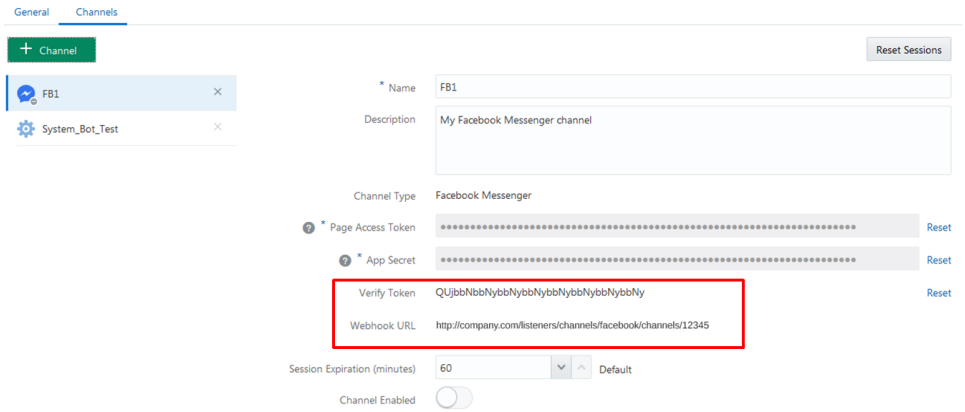
5. Copy and the Page Access Token from Facebook and paste it into the Page Access Token field in the Create Channel Dialog. You can find this key in the Facebook Messenger Platform settings page.



6. Copy the App Secret —You can find this in the Facebook Messenger Platform dashboard for your Facebook App.



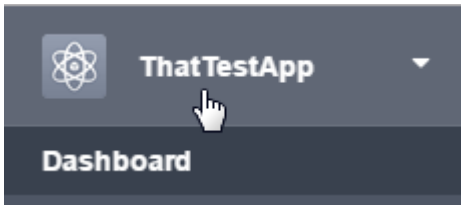
7. Click **Create**.
8. In the Channels page, note the Verify Token and WebHook URL. You'll need these to configure the Facebook webhook.



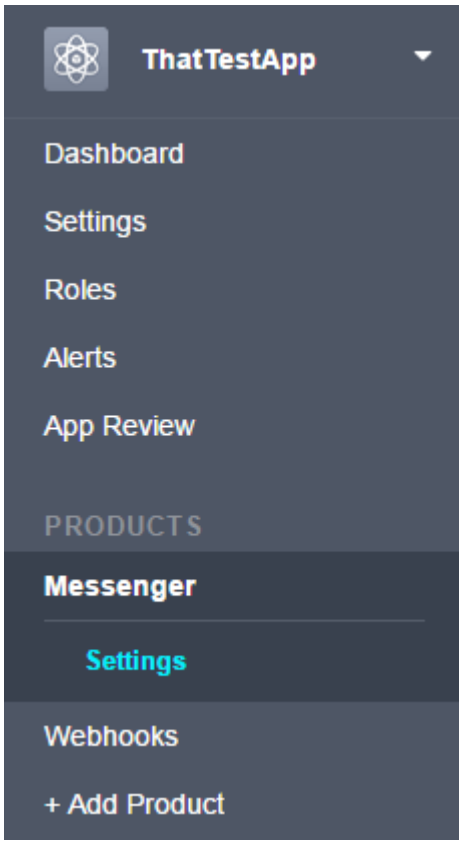
Step 3: Configure the Facebook Messenger Webhook

Define the Callback URL by adding the Webhook URL generated by Bots to Facebook Messenger. Refer to the Channels page in the Bot Builder for these two values.

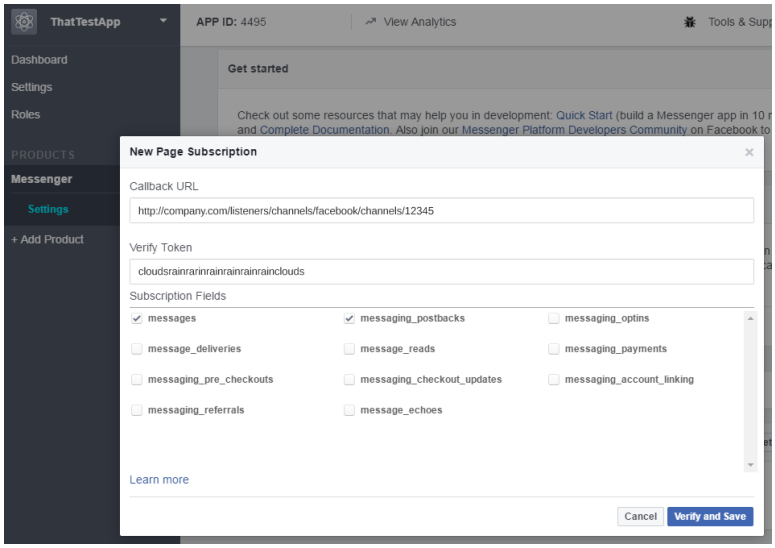
1. In Facebook Messenger, be sure that you've selected the project that you initially created for the webhook.



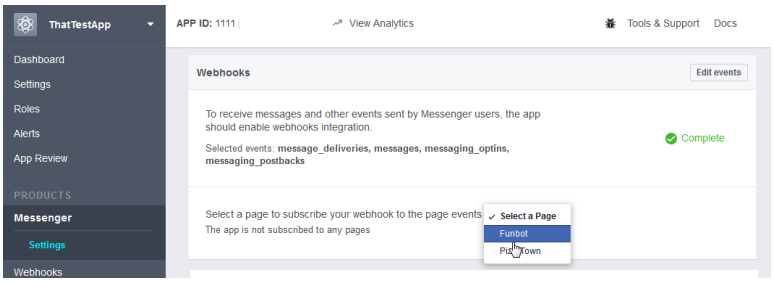
2. Click Messenger and then choose **Settings** .



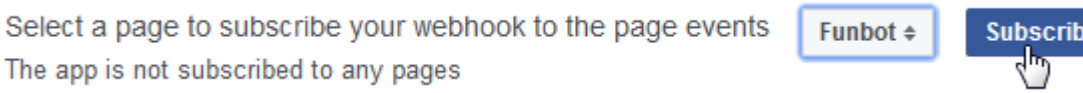
3. Click **Setup Webhooks** to open the New Page Subscription dialog.
4. Copy the Webhook URL from the Bots Channels page and paste it in the CallBack URL field in the New Page Subscription dialog.
5. Copy the Verify Token generated by Bots and paste it into the Verify Token field.
6. Subscribe to only the **messages** and **messaging_postbacks** callback events. The `messages` event is triggered whenever someone sends a message to your Facebook page.



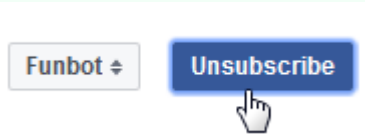
7. Subscribe to the page:
   a. Choose your bot's Facebook page.



   b. Click **Subscribe**.

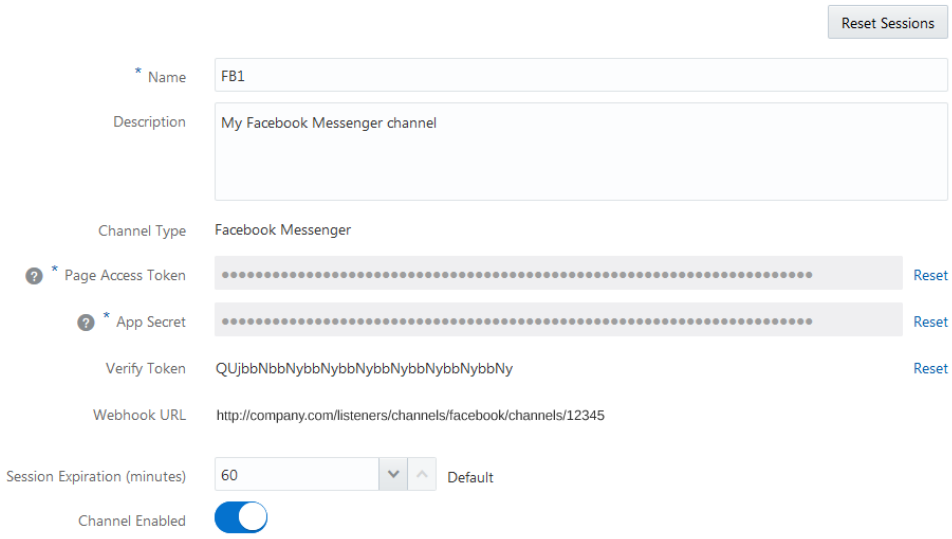


.

> **Tip:**
>
> You might need to bounce your webhook by first clicking **Unsubscribe** then **Subscribe**.
>
> 
>

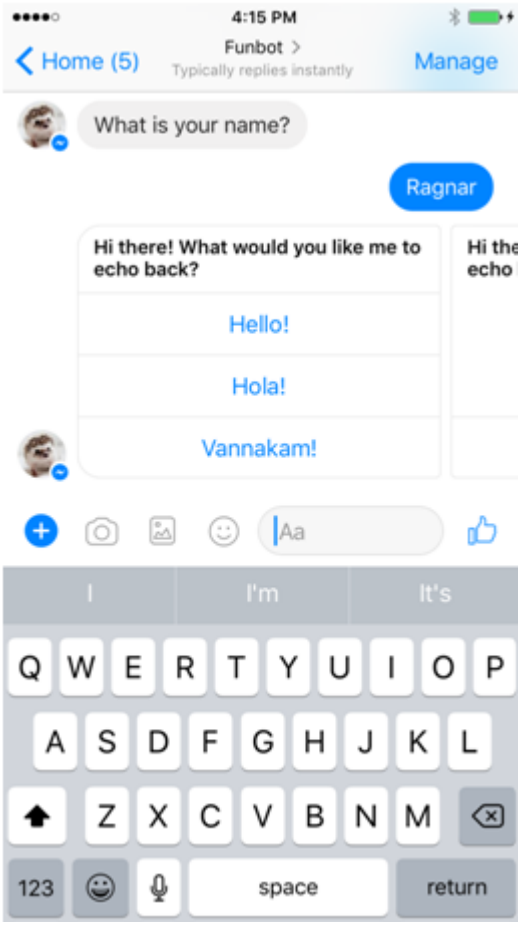
Step 4: Enable the Facebook Channel

With the configuration complete, you're ready to activate the Facebook channel by switching on **Channel Enabled** in Bots. You can now test out your bot.



Step 5: Testing Your Bot on Facebook Messenger

With the Facebook Channel and messaging configuration complete, you can test your simultaneously using your Facebook page, Facebook Messenger (`https://www.messenger.com/`) and the Facebook Messenger app on your phone (). Once you locate your bot in the search, you're ready to start chatting with it. You can see the changes that you make to the dialog flow in real time.

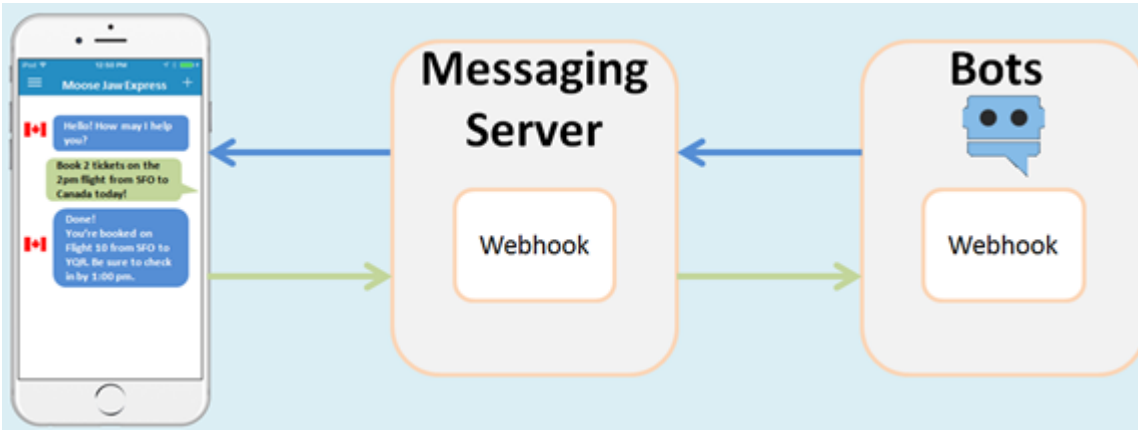


Running Your Bot on Other Messaging Services

To allow your bot to talk to users who aren't subscribed to Facebook Messenger, you need to configure the Webhook channel.

To create a Webhook channel, you need the following:

- A publicly accessible HTTP messaging server that relays messages between the user device and your bot using a webhook.



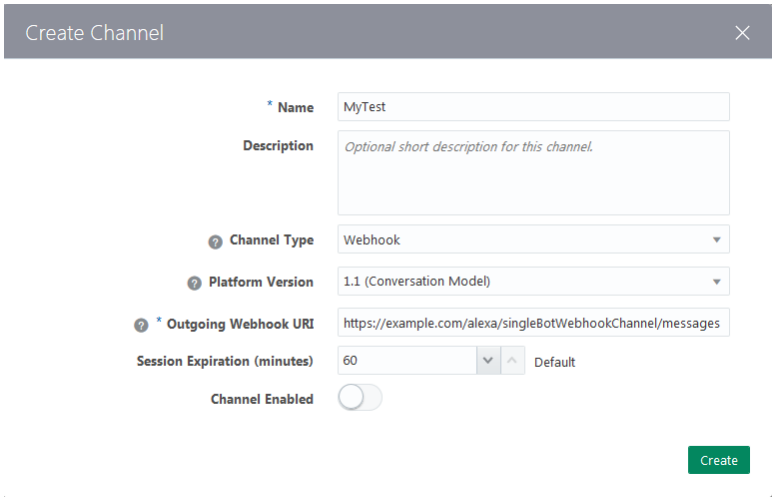
You implement this webhook with:

1. A POST call that enables the server to receive messages from your bot.
2. A POST call that enables the server to send messages to your bot.

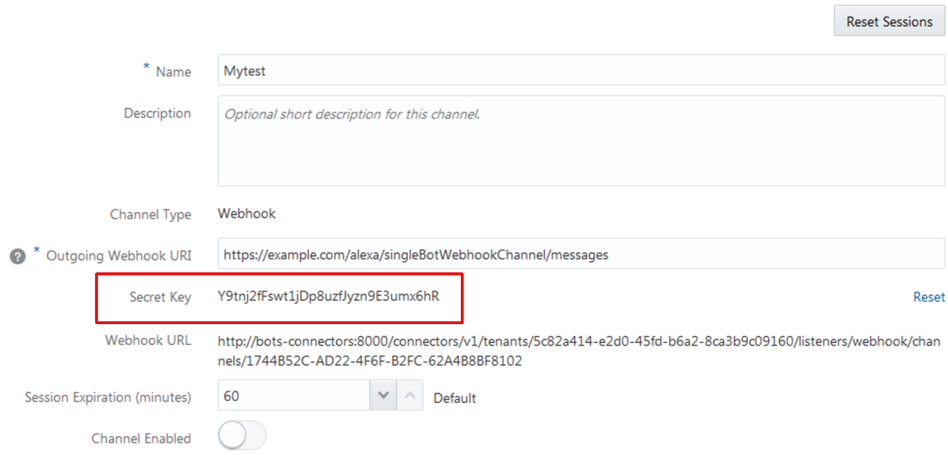
- Because your bot needs to know where to send its message, you need the URI of the webhook call that receives your bot's messages.
- Likewise, the message server needs to know how to find your bot, so you need the Webhook URL that's generated for your bot after you complete the Create Channel dialog.

To assemble these pieces into a webhook:

1. Set up the server.
2. To receive messages from your bot, publish the POST call on the server.
3. In the Create Channel dialog, enter a name and then:
   - Choose Webhook as the channel type.
   - Set Platform Version to **1.1 (Conversation Model)**.
   - Register the server as the recipient of your bot's messages by entering the URI to this POST call in the Outgoing Webhook URI field.
   - If needed, enter the session expiry and switch on **Channel Enabled**.



4. After you click **Create**, Bots generates the webhook URL for your bot and its Secret Key for encrypting messages. Keep the webhook URL handy, because it's the pointer that your messaging server needs to send messages back to your bot.



5. On your server, publish the second POST API, one that sends messages to your bot using the webhook URL.
6. Switch the **Channel Enabled** option on.

Outbound Messages

You need to publish the calls in the JSON format that Bots expects, along with the authorization header.

The call for your bot's outbound messages includes:

1. An `X-Hub-Signature` header containing the SHA256 value of the payload, calculated using the Secret Key as the key.

> **Note:**
>
> Bots uses the `X-Hub-Signature` header to allow the recipient to authenticate your bot as the sender and validate the integrity of the payload.

2. A JSON payload containing the `userID`, a unique identifier that's specified by the inbound message, the `type`, which can be `text`,`attachment`, and `card`. As shown in the following examples, both the `text` and `card` response types can have associated actions. Any of the response types can also include global actions.

Response Type	Example Payload

text	<pre>{
	 "userId":"22343248763458761287
	 "messagePayload": {
	  "type": "text",
	  "text": "Hello, how are you?"
	  }
	}</pre>
	The following snippet show a text response with actions:
	<pre>{
	 "userId":"22343248763458761287
	 "messagePayload": {
	  "type": "text",
	  "text": "What do you want to do?",
	  "actions": [
	  {
	  "type": "postback",
	  "label": "Order Pizza",
	  "postback": {
	  "state": "askAction",
	  "action": "orderPizza"
	  }
	  },
	  {
	  "type": "postback",
	  "label": "Cancel A Previous Order",
	  "postback": {
	  "state": "askAction",
	  "action": "cancelOrder"
	  }
	  ]
	 }
	}</pre>

Response Type	Example Payload
card	...

```
{
 "type": "card",
  "layout": "horiztonal",
  "cards": [
    {
      "title": "Hawaiian Pizza",
      "description": "Ham and
pineapple on thin crust",
      "actions": [
        {
          "type": "postback",
          "label": "Order Small",
          "postback": {
            "state": "GetOrder",
            "variables": {
              "pizzaType": "hawaiian",
              "pizzaCrust": "thin",
              "pizzaSize": "small"
            }
          }
        },
        {
          "type": "postback",
          "label": "Order Large",
          "postback": {
            "state": "GetOrder",
            "variables": {
              "pizzaType": "hawaiian",
              "pizzaCrust": "thin",
              "pizzaSize": "large"
            }
          }
        }
      ]
    },
    {
      "title": "Cheese Pizza",
      "description": "Cheese pizza
(i.e. pizza with NO toppings) on thick
crust",
      "actions": [
        {
          "type": "postback",
          "label": "Order Small",
          "postback": {
            "state": "GetOrder",
            "variables": {
              "pizzaType": "cheese",
              "pizzaCrust": "thick",
              "pizzaSize": "small"
            }
          }
        },
        {
          "type": "postback",
          "label": "Order Large",
```

Response Type	Example Payload

	<pre>"postback": {
	"state": "GetOrder",
	"variables": {
	"pizzaType": "cheese",
	"pizzaCrust": "thick",
	"pizzaSize": "large"
	}
	}
	}
	]
	}
	],
	"globalActions": [
	{
	"type": "call",
	"label": "Call for Help",
	"phoneNumber": "123456789"
	}
	]
	}</pre>

attachment	The attachment response type can an image, audio file, or a video:
	<pre>...
	{
	"type": "attachment",
	"attachment": {
	"type": "video",
	"url": "https://www.youtube.com/watch?v=CMNry4PE93Y"
	}
	}</pre>

Inbound Messages

The call for sending messages to your bot must have:

1. An `X-Hub-Signature` header containing the SHA256 value of the payload. The call includes functions that create this hash using Secret Key as the key.

```
const body = Buffer.from(JSON.stringify(messageToBot), 'utf8');
    const headers = {};
    headers['Content-Type'] = 'application/json; charset=utf-8';
    headers['X-Hub-Signature'] = buildSignatureHeader(body, channelSecretKey);

...

function buildSignatureHeader(buf, channelSecretKey) {
    return 'sha256=' + buildSignature(buf, channelSecretKey);
}

function buildSignature(buf, channelSecretKey) {
    const hmac = crypto.createHmac('sha256', Buffer.from(channelSecretKey,
'utf8'));
    hmac.update(buf);
```

```
    return hmac.digest('hex');
}
```

2. A JSON obect with `userId`, `userProfile`, and `messagePayload` properties:

```
{
 "userid: "33c0bcBc8e-378c-4496-bc2a-b2b9647de2317"
 "userProfile": {
    "firstName": "Bob",
    "lastName": "Franklin",
    "age": 45
   },
 "messagePayload: {....}
}
```

Property	Description	Type	Required?
`userId`	A unique identifier for the user. This ID is specific to the caller.	String	Yes
`userProfile`	Properties that represent the user, like `firstName` and `LastName`.	JSON object	No

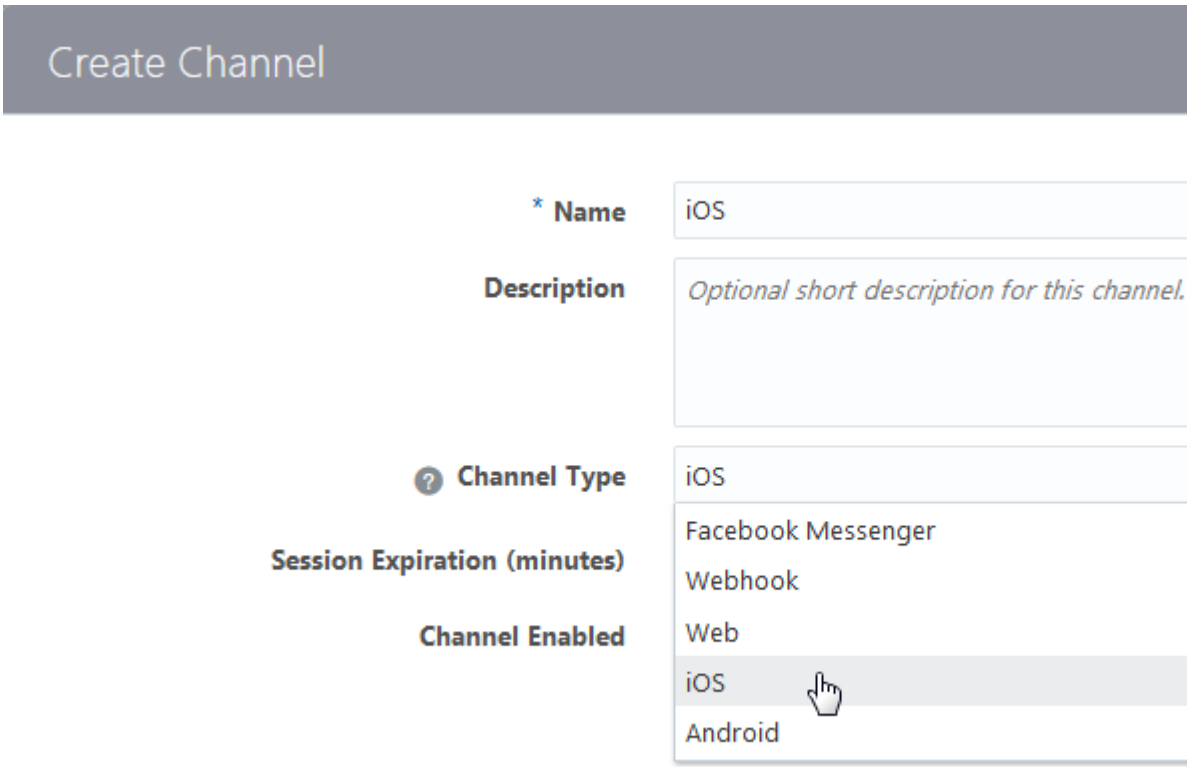
Property	Description	Type	Required?
messagePayload	The messagePayload can be text, postback, attachment, and location: • text { "type": "text", "text": hello, world!" } • postback { "type": "postback", "postback": { "state": "orderPizza", "action": "deliverPizza", "variables": { "pizzaSize": "Large", "pizzaCrust": "Thin", "pizzaType": "Hawaiian" } } } • attachment { "type": "attachment", "attachment": { "type": "image", "url": "https://image.freepik.com/free-icon/attachment-tool-ios-7-interface-symbol_318-35539.jpg" } }	JSON object	Yes

Property	Description	Type	Required?

	• location
	
	{
	"type":
	"location",
	
	"location": {
	
	"longitude":
	-122.265987
	
	"latitude":
	37.529818
	}
	}		

Running Your Bot Within Client Messaging Apps and Web Pages

We provide SDKs that enable you to integrate your bot with iOS apps, Android apps, and web pages. For any of these integrations, you need to generate the App Id by creating a Web, iOS, or Android channel.



After you create the App Id, you copy and paste it into the client app code or, if you're integrating your bot into a web page, the `<script>` tag.

Bots Client SDKs

Bots Client SDK for Android

- Adding the Bots Client SDK for Android to Your App
- Localization
- Permissions

Adding the Bots Client SDK for Android to Your App

The Bots Client SDK for Android library is distributed in both AAR and JAR formats. If you are using Android Studio, follow the instructions for installation of the AAR package.

> **Note:**
>
> Compile your app using API Level 26 (Android Oreo) or higher. Level 19 (Android 4.4, Kitkat) is the lowest version that can support the Bots client SDK for Android. If your app needs to support even earlier versions, keep in mind that we haven't tested these and therefore can't guarantee their compatibility.

Adding the SDK and AAR Files

1. Download the Bots Client SDK for Android 18.2.3.0 module from the Oracle Technology Network's Oracle Mobile Cloud Enterprise download page.
2. Import the core and UI files (`bots-client-sdk-android-core-v18.2.3.aar` and `bots-client-sdk-android-ui-v18.2.3.aar` into your Android Studio project by going to **File > New > New Module > Import .JAR/.AAR Package**.
3. Add the following lines to the project's `build.gradle` file:

```
compile project(':bots-client-sdk-android-core-1.2.1')
compile project(':bots-client-sdk-android-ui-1.2.1')

compile 'com.google.firebase:firebase-messaging:11.0.4'
compile 'com.google.firebase:firebase-core:11.0.4'
compile 'com.google.code.gson:gson:2.4'
compile 'com.squareup.okhttp3:okhttp:3.4.1'
compile 'com.android.support:support-annotations:26.0.2'
compile 'com.android.support:appcompat-v7:26.0.2'
compile 'com.android.support:recyclerview-v7:26.0.2'
compile 'com.nostra13.universalimageloader:universal-image-loader:1.9.5'
compile 'com.davemorrissey.labs:subsampling-scale-image-view:3.5.0'
compile 'com.google.android.gms:play-services-location:11.0.4'
```

Initialize the Bots Android SDK in Your App

Before your code can invoke the SDK's functionality, you'll have to initialize the library using your app's ID.

To get this unique ID, first click **Add Channel** to open the Add Channel dialog. Complete the dialog by adding a channel name and then choosing **Android** as the channel type. After you click **OK**, Bots generates the App ID.

After you obtain this ID, use the following to initialize the SDK.

```
Bots.init(this, new Settings("YOUR_APP_ID"), newBotsCallback() {
     @Override
     public void run(Response response) {
          // Your code after init is complete
    }
});
```

> **Note:**
>
> Make sure to replace `YOUR_APP_ID` with your app ID.

To ensure that the SDK is always initialized properly, copy the following snippet and save it to your application package.

```
package your.package;

import android.app.Application;
import oracle.cloud.mobile.core.Bots;

public class YourApplication extends Application {
    @Override
    public void onCreate() {
        super.onCreate();
        Bots.init(this, new Settings("YOUR_APP_ID"), new BotsCallback() {
            @Override
            public void run(Response response) {
                // Your code after init is complete
            }
        });
    }
```

> **Note:**
>
> You need to declare this in the `Application` class because it's the class that's required by `Bots.init(this, new Settings("YOUR_APP_ID")`. If you declare this class elsewhere (say, `AppCompatActivity`), then add the following snippet, which uses `getApplication()` as the first argument:
>
> ```
> @Override
> protected void onCreate(Bundle savedInstanceState) {
>     super.onCreate(savedInstanceState);
>
>     Bots.init(getApplication(), new Settings("YOUR_APP_ID"), new
> BotsCallback() {
>         @Override
>         public void run(Response response) {
>             // Your code after init is complete
>         }
>     });
> }
> ```

You also need to declare your newly created class in the `<application>` tag in your `AndroidManifest` file.

```
<application
    android:name="your.package.YourApplication">
    ...
</application>
```

> **Note:**
>
> Remember to replace `your.package`, `YourApplication`, `YOUR_APP_ID` with the appropriate names and the App Id for the Android channel.

Displaying the Bots Android SDK User Interface

Once you've initialized Bots Android SDK, you're ready to try it out.

Find a suitable place in your app's interface to invoke the SDK and use the code below to display the Android Messenger user interface. You invoke up the Bots Android SDK whenever your user needs access to help or needs to contact you.

```
ConversationActivity.show(getApplicationContext());
```

Calling Other Functions

You can call various functions when the SDK has been initialized successfully. As shown in the following snippet, you can update user properties before your app calls the `ConversationActivity` class:

```
if (Bots.getInitializationStatus() == InitializationStatus.Success) {
   Log.d(TAG,"Already Initialized with App ID "+ mAppID);
   User.getCurrentUser().setFirstName("John");
   User.getCurrentUser().setLastName("Smith");
   User.getCurrentUser().setEmail("john.smith@example.com");
   User.getCurrentUser().setSignedUpAt(new Date());
```

```
    final Map<String, Object> customProperties = new HashMap<>();
    customProperties.put("premiumUser", true);
    customProperties.put("numberOfPurchases", 20);
    customProperties.put("itemsInCart", 3);
    customProperties.put("couponCode", "PREM_USR");
    User.getCurrentUser().addProperties(customProperties);

    ConversationActivity.show(getApplicationContext(), Intent.FLAG_ACTIVITY_NEW_TASK);
}
```

Replacing the FileProvider

If you do not have a `FileProvider` entry in your `AndroidManifest.xml` file, you can safely ignore this section. These steps will fix the `Manifest merger failed : Attribute provider#android.support.v4.content.FileProvider@authorities` compile error

To replace the FileProvider with your own, please do the following:

1. Add `tools:replace="android:authorities"` to the `<provider>` entry.
2. Add the following path to your android.support.FILE_PROVIDER_PATHS resource file:

   ```
   <external-path name="dcim" path="DCIM"/>
   ```

3. When initializing the SDK, call `settings.setFileProviderAuthorities(authoritiesString);` on the settings object.

   ```
   Settings settings = new Settings(appId);
   settings.setFileProviderAuthorities(authoritiesString);
   Bots.init(this, settings, myInitCallback);
   ```

Localization

Every string you see in Bots can be customized and localized. Bots provides a few languages out of the box, but adding new languages is easy to do. When localizing strings, Bots looks for values in the `strings.xml` in your app first then in the Bots UI bundle, enabling you to customize any strings and add support for other languages.

Adding More Languages

To enable other languages beside the provided ones, first copy the English `strings.xml` file from the Bots UI bundle to the corresponding values folder for that language. Then, translate the values to match that language.

Customization

- Strings Customization
- Styling the Conversation Interface

Strings Customization

Bots lets you customize any strings it displays by overwriting its keys. To do this, simply add `res/values-<your-language-code>/strings.xml` file in your Android project and specify new values for the keys used in Bots. You can find all of the available keys by browsing to the `artifacts:bots-client-sdk-android-ui-x.x.x/res/values/values.xml` file in the External Libraries in Android Studio.

Dates shown in the conversation view are already localized to the user's device.

For example, if you wanted to override strings for English, you would create a file called `res/values-en/strings.xml` and include the following in that file:

```
<resources>
<string name="Bots_activityConversation">Messages</string>
<string name="Bots_startOfConversation">This is the start of your conversation with
the team.</string>
<string name="Bots_welcome">Feel free to leave us a message about anything that\'s
on your mind.</string>
<string name="Bots_messageHint">Type a message…</string>
</resources>
```

> **Note:**
>
> if you want to specify new strings for the default fallback language, you must override them in the `res/values/string.xml` file.

Styling the Conversation Interface

Using a `colors.xml` file in your `res/values` folder, you can change the colors used by Bots:

```
<resources>
<color name="Bots_accent">#9200aa</color>
<color name="Bots_accentDark">#76008a</color>
<color name="Bots_accentLight">#be7cca</color>

<color name="Bots_backgroundInput">#ffffff</color>

<color name="Bots_btnSendHollow">#c0c0c0</color>
<color name="Bots_btnSendHollowBorder">#303030</color>

<color name="Bots_header">#989898</color>

<color name="Bots_messageDate">@color/Bots_header</color>
<color name="Bots_messageShadow">#7f999999</color>

<color name="Bots_remoteMessageAuthor">@color/Bots_header</color>
<color name="Bots_remoteMessageBackground">#ffffff</color>
<color name="Bots_remoteMessageBorder">#d9d9d9</color>
<color name="Bots_remoteMessageText">#000000</color>

<color name="Bots_userMessageBackground">@color/Bots_accent</color>
<color name="Bots_userMessageBorder">@color/Bots_accentDark</color>
<color name="Bots_userMessageFailedBackground">@color/Bots_accentLight</color>
<color name="Bots_userMessageText">#ffffff</color>
</resources>
```

If you need to update the image of the Send button, simply add an image with the following name to your drawables:

```
bots_btn_send_normal.png
```

You can find the original resources by browsing external libraries through Android Studio.

Permissions

The Bots Client SDK for Android library includes the following permissions by default:

```
<uses-permission android:name="android.permission.ACCESS_NETWORK_STATE"/>
<uses-permission android:name="android.permission.INTERNET"/>
<uses-permission android:name="android.permission.VIBRATE"/>
<uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE"/>
<uses-permission android:name="android.permission.ACCESS_FINE_LOCATION"/>
```

- WRITE_EXTERNAL_STORAGE is used to take photos and to store downloaded pictures locally to avoid needless re-downloading.
- ACCESS_FINE_LOCATION is used in order to access the customer's location when requested using location request buttons.

If you do not intend to request the user's location at any point, it is safe to remove the `ACCESS_FINE_LOCATION` using the following override:

```
<uses-permission android:name="android.permission.ACCESS_FINE_LOCATION"
tools:node="remove" />
```

All other permissions are necessary for Bots to function as intended.

Bots Client SDK for iOS

- Adding the Bots Client SDK for iOS to Your App
- Updating the SDK
- Localization of iOS Apps
- Customization

Adding the Bots Client SDK for iOS to Your App

1. Download the Bots Client SDK for iOS 18.2.3.0 module from the Oracle Technology Network's Oracle Mobile Cloud Enterprise download page.
2. Unzip the file. This creates a directory called `Bots.framework` (the framework).
3. Add the framework to your Xcode project by selecting **File > Add Files** to My_Project and then selecting `Bots.framework` in the file picker.
4. In your project settings, add `Bots.framework` to the list of Embedded Binaries in the General tab for your application target.

   You can now import the framework (`#import <Bots/Bots.h>`) and start using it in your code.

Import the Bots Header File

Import the Bots file into the your app delegate's `.m` file and any other places you plan to use it.

- Objective-C:

  ```
  #import <Bots/Bots.h>
  ```

- Swift:

  ```
  import Bots
  ```

Add Required Keys in Your App's info.plist

The Bots Client SDK for iOS may need to ask users permission to use certain features. Depending on the feature, you must provide a description in your app's `Info.plist` to explain why access is required. These descriptions will be displayed the moment it prompts the user for permission.

Images

The Bots Client SDK for iOS allows users to send images. To support this feature, you need to provide a description for the following keys:

- `NSCameraUsageDescription`—Describes the reason your app accesses the camera (for example: camera permission is required to send images to `${PRODUCT_NAME}`). For more information, see the iOS documentation about `NSCameraUsageDescription`.
- `NSPhotoLibraryUsageDescription`—Describes the reason your app accesses the photo library (for example: photo library permission is required to send images to `${PRODUCT_NAME}`). For more information, see the iOS documentation about `NSPhotoLibraryUsageDescription`.

> **Note:**
>
> Beginning with iOS 10, these values are required. If they are not present in your app's `Info.plist`, the option to send an image will not be displayed.

Location

The Bots Client SDK for iOS also allows users to send their current location. To support this feature, you must provide a description for any of the following keys depending on your app's use of location services. The SDK will ask the user for the location depending on the key you provide:

- `NSLocationWhenInUseUsageDescription`—Describes the reason for your app to access the user's location information while your app is in use (for example: location services is required to send your current location to `${PRODUCT_NAME}`). This permission is recommended if your app does not use location services. The SDK will default to it if both keys are included. See the iOS documentation about `NSLocationWhenInUseUsageDescription`.
- `NSLocationAlwaysUsageDescription`—Describes the reason for your app to access the user's location information at all times (for example: location services is required to send your current location to `${PRODUCT_NAME}`). See the iOS documentation about `NSLocationAlwaysUsageDescription`.

> **Note:**
>
> If you don't provide one of these keys, any attempt from the user to send their current location will fail.

Initialize the Bots Client SDK for iOS in Your App

Before your code can invoke the SDK, you'll have to initialize the library using the App Id that's generated for your Bot when you add an iOS channel. To get this ID, click **Add Channel** to open the Create Channel dialog. Add a name for the iOS channel and then choose **iOS** as the channel type. After you click **Create**, Bots generates the App Id. Next, use this ID to replace `YOUR APP ID` in the `applicationDidFinishLaunchingWithOptions:` method:

- Objective-C

```
[Bots initWithSettings:[OMCSettings settingsWithAppId:@"YOUR_APP_ID"]
completionHandler:^(NSError * _Nullable error, NSDictionary * _Nullable
userInfo) {
    // Your code after init is complete
}];
```

- Swift:

```
Bots.initWith(OMCSettings(appId: "YOUR_APP_ID")) { (error: Error?, userInfo:
[AnyHashable : Any]?) in// Your code after init is complete}
```

You can show the Bots UI anywhere in your app after it finishes loading by adding the following line::

- Objective-C:

```
[Bots show];
```

- Swift:

```
Bots.show();
```

Calling Other Functions

You can call other functions after the SDK has been initialized successfully. For example, you can update the user's first name, last name, and email address:

```
// Update first name and last name
   [Bots setUserFirstName:@"John"
           lastName:@"Smith"];

// Update email address
   [OMCUser currentUser].email = @"john.smith@example.com";
```

Updating the SDK

Run the following to update Carthage dependencies:

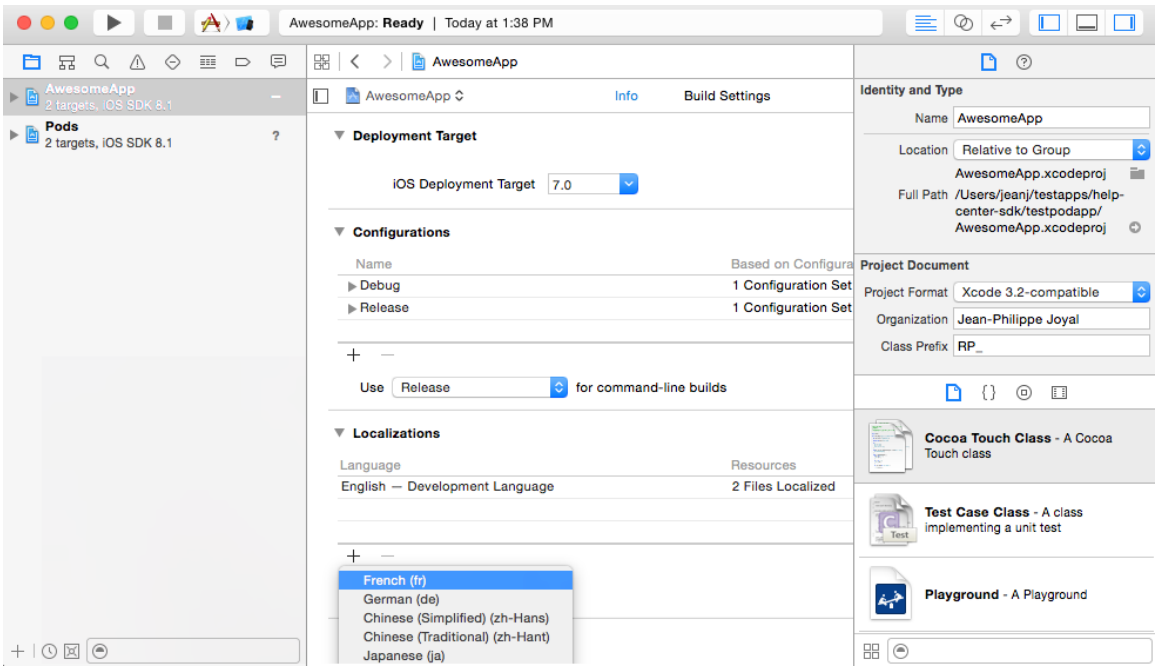
```
$ carthage update
```

Localization of iOS Apps

Every string you see in your bot can be customized and localized. Bots provides a few languages out of the box, but adding new languages is easy to do. When localizing strings, Bots looks for `BotsLocalizable.strings` in your app bundle first then in the Bots bundle, enabling you to customize any strings and add support for other languages.

Enabling Localization in Your iOS App

For Bots to display a language other than English, your app needs to first enable support for that language. You can enable a second language in your Xcode project settings:



Once you have this, Bots will display itself in the device language for the supported language.

These languages are included with the Bots Client SDK for iOS: Arabic, English, Finnish, French, German, Italian, Japanese, Korean, Mandarin Chinese (traditional and simplified), Persian, Portuguese (Brazil and Portugal), Russian, Slovenian, Spanish, and Swedish.

> **Note:**
>
> Localization is subject to caching. If you can't see your changes, cleaning your project, resetting the simulator, deleting your app from your test devices are good measures.

Customization

- Strings Customization
- Styling the Conversation Interface

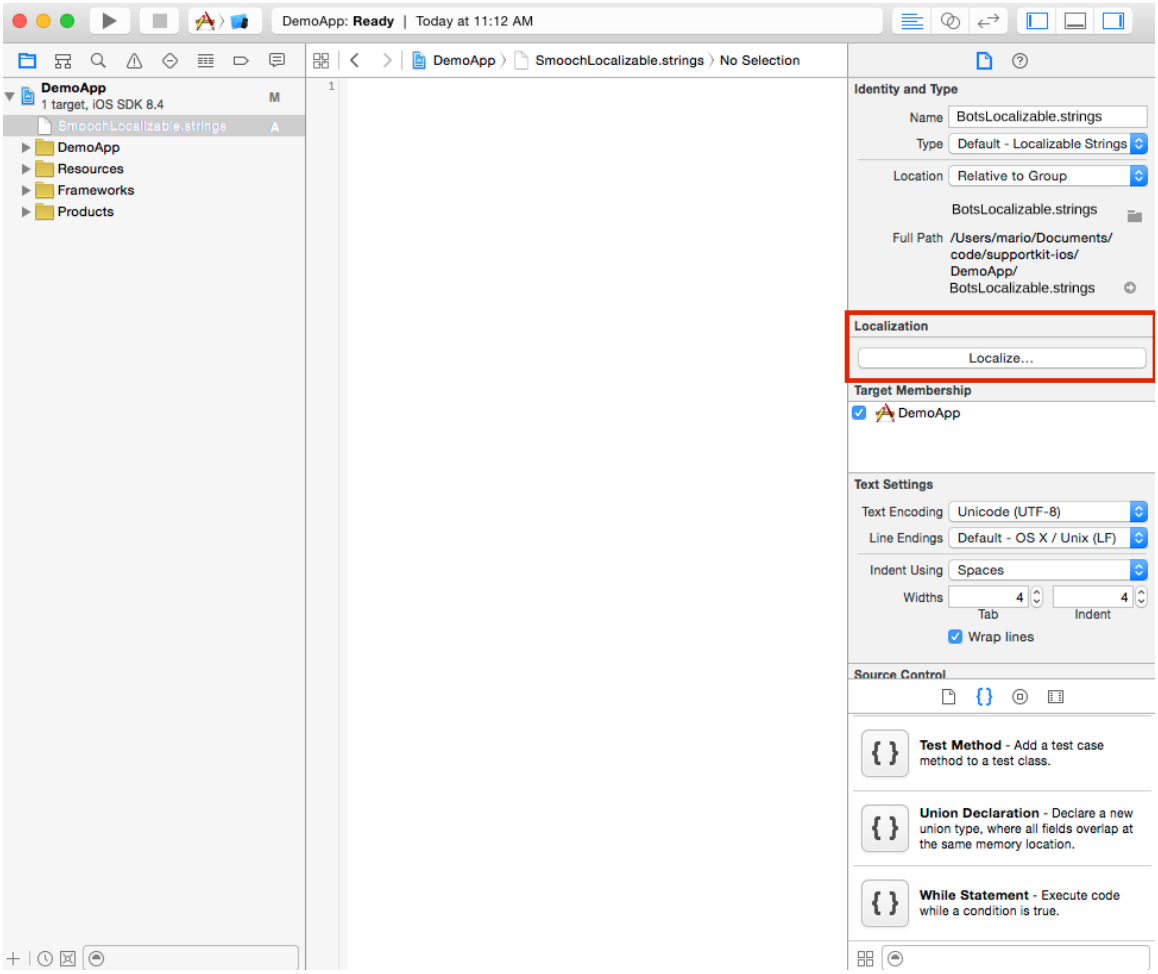
Strings Customization

Bots lets you customize any strings it displays via Apple's localization mechanism. To override one or more strings, add an empty string file named `BotsLocalizable.strings`

in your Xcode project and specify new values for the keys you would like to override. For example, to change the "Messages" header, and the "Done" button create a file with these contents:

```
"Messages" = "My Messages"

"Done" = "I'm Done"
```

To enable string customization across languages, make sure you localize your `BotsLocalizable.strings` file in Xcode.



The BotsLocalizable.strings File

Here is the full set of keys:

```
/* Nav bar button, action sheet cancel button */
"Cancel" = "...";

/* Conversation title */
"Messages" = "...";

/* Conversation header. Uses CFBundleDisplayName */
"This is the start of your conversation with the %@ team. We'll stay in touch to
help you get the most out of your app.\nFeel free to leave us a message about
anything that's on your mind. We'll get back to your questions, suggestions or
```

```
anything else as soon as we can." = "...";

/* Conversation header when there are previous messages */
"Show more..." = "...";

/* Conversation header when fetching previous messages */
"Retrieving history..." = "...";

/* Error message shown in conversation view */
"No Internet connection" = "...";

/* Error message shown in conversation view */
"Could not connect to server" = "...";

/* Error message shown in conversation view */
"An error occurred while processing your action. Please try again." = "...";

/* Error message shown in conversation view */
"Reconnecting..." = "...";

/* Fallback used by the in app notification when no message author name is found */
"%@ Team" = "...";

/* Conversation send button */
"Send" = "...";

/* Conversation text input place holder */
"Type a message..." = "...";

/* Conversation nav bar left button */
"Done" = "...";

/* Failure text for chat messages that fail to upload */
"Message not delivered. Tap to retry." = "...";

/* Status text for chat messages */
"Sending..." = "...";

/* Status text for sent chat messages */
"Delivered" = "...";

/* Status text for chat messages seen by the appMaker */
"Seen" = "...";

/* Timestamp text for recent messages */
"Just now" = "...";

/* Timestamp text for messages in the last hour */
"%.0fm ago" = "...";

/* Timestamp text for messages in the last day */
"%.0fh ago" = "...";

/* Timestamp text for messages in the last week */
"%.0fd ago" = "...";

/* Action sheet button label */
"Take Photo" = "...";

/* Action sheet button label */
"Use Last Photo Taken" = "...";
```

```
/* Action sheet button label */
"Choose from Library" = "...";

/* Photo confirmation alert title */
"Confirm Photo" = "...";

/* Action sheet button label */
"Resend" = "...";

/* Action sheet button label */
"View Image" = "...";

/* Error displayed in message bubble if image failed to download */
"Tap to reload image" = "...";

/* Error displayed as message if location sending fails */
"Could not send location" = "...";

/* Error title when user selects "use latest photo", but no photos exist */
"No Photos Found" = "...";

/* Error description when user selects "use latest photo", but no photos exist */
"Your photo library seems to be empty." = "...";

/* Error title when user attempts to upload a photo but Photos access is denied */
"Can't Access Photos" = "...";

/* Error description when user attempts to upload a photo but Photos access is
denied */
"Make sure to allow photos access for this app in your privacy settings." = "...";

/* Error title when user attempts to take a photo but camera access is denied */
"Can't Access Camera" = "...";

/* Error description when user attempts to take a photo but camera access is denied
*/
"Make sure to allow camera access for this app in your privacy settings." = "...";

/* Generic error title when user attempts to upload an image and it fails for an
unknown reason */
"Can't Retrieve Photo" = "...";

/* Generic error description when user attempts to upload an image and it fails for
an unknown reason */
"Please try again or select a new photo." = "...";

/* Error title when user attempts to send the current location but location access
is denied */
"Can't Access Location" = "...";

/* Error description when user attempts to send the current location but location
access is denied */
"Make sure to allow location access for this app in your privacy settings." = "...";

/* UIAlertView button title to link to Settings app */
"Settings" = "...";

/* UIAlertView button title to dismiss */
"Dismiss" = "...";
```

```
/* Title for payment button */
"Pay Now" = "...";

/* Title for message action when payment completed */
"Payment Completed" = "...";

/*
 Instructions for entering credit card info. Parameters are as follows:
 1. Amount (e.g. 50.45)
 2. Currency (e.g. USD)
 3. App name (Uses CFBundleDisplayName)
*/
"Enter your credit card to send $%@ %@ securely to %@" = "...";

/* Error text when payment fails */
"An error occurred while processing the card. Please try again or use a different
card." = "...";

/* Button label for saved credit card view */
"Change Credit Card" = "...";

/*
 Information label for saved credit card view. Parameters are as follows:
 1. Amount (e.g. 50.45)
 2. Currency (e.g. USD)
 3. App name (Uses CFBundleDisplayName)
 */
"You're about to send $%@ %@ securely to %@" = "...";

/* Title for user notification action */
"Reply" = "...";

/* Date format used for message grouping headers on the conversation screen */
"MMMM d, h:mm a" = "MMMM d, h:mm a";

/* Date format used for message timestamps on the conversation screen */
"hh:mm a" = "hh:mm a";

/* Error message when the content of a webview fails to load */
"Failed to open the page" = "...";
```

Styling the Conversation Interface

The style of the conversation user interface can be controlled through two techniques:

- Using the `UIAppearance` proxy of `UINavigationBar` to style the navigation bar's color and appearance.
- The `OMCSettings` class provides access to the status bar and the color of the message bubbles.

Suppose you wanted the conversation UI to have a black navigation bar and red message bubbles. First, you'd use `UINavigationBar`'s appearance proxy to set up the navigation bar. Then, you'd use `OMCSettings` to finish styling the UI:

- Objective C

```
OMCSettings* settings = [OMCSettings settingsWithAppId:@"YOUR_APP_ID"];
settings.conversationAccentColor = [UIColor redColor];
settings.conversationStatusBarStyle = UIStatusBarStyleLightContent;

[[UINavigationBar appearance] setBarTintColor:[UIColor blackColor]];
```

```
[[UINavigationBar appearance] setTintColor:[UIColor redColor]];
[[UINavigationBar appearance]
setTitleTextAttributes:@{ NSForegroundColorAttributeName : [UIColor redColor] }];
```

- Swift

```
var settings = OMCSettings(appId: "YOUR_APP_ID");
settings.conversationAccentColor = UIColor.red();
settings.conversationStatusBarStyle = UIStatusBarStyle.LightContent
UINavigationBar.appearance().barTintColor = UIColor.black();
UINavigationBar.appearance().tintColor = UIColor.red();
UINavigationBar.appearance().titleTextAttributes =
[ NSForegroundColorAttributeName : UIColor.red()];
```

Bots Client SDK for JavaScript

The Bots Client SDK for JavaScript module is a highly customizable messaging widget that can be added to any web page.

- Configuring the Library
- Deploying the SDK Files
- Adding Bots Client SDK for JavaScript to Your Site
- Customization

Configuring the Library

The Bots library is composed of multiple assets that get fetched at runtime for better performance. For that reason, the public path (the URL where the `static` files are hosted) is hardcoded in multiple places.

To configure the library for your environment, run:

```
./configure
```

The script generates a folder with the configured project in it.

Setup Examples

Local Testing Setup

If the static files are hosted at `http://localhost:8000/static/` and you run the following script from the `/home/your-name/` folder:

```
./configure http://localhost:8000/static/
```

then the files will be available at `/home/your-name/http:__localhost:8000_static_/`.

Production Setup

If the static files are hosted at `https://cdn.acme.org/` and you run the following script from the `/home/your-name/` folder:

```
./configure https://cdn.acme.org/
```

then the files will be available at `/home/your-name/https:__cdn.acme.org_/`.

Deploying the SDK Files

1. Download the Bots Client SDK for JavaScript 18.2.3.0 module from the Oracle Technology Network's Oracle Mobile Cloud Enterprise download page.
2. Put all of the files from the generated folder at the root of the storage within `https://placeholder.public.path/`. For example if your files are hosted at `http://localhost:8000/static/`, copy all the files to the `static` folder on your local server.
3. If your storage is behind a CDN (Content Delivery Network), issue a cache invalidation for `https://placeholder.public.path/loader.json`.
4. Make sure your server allows CORS requests.
5. Test your deployment by initializing the SDK as described in Adding Bots Client SDK for JavaScript to Your Site.

Adding Bots Client SDK for JavaScript to Your Site

You include the Bots Client SDK for JavaScript by editing the `<script>` tag. You need an App Id to do this, so if don't have one already for the Web channel, start off by clicking **Add Channel**. In the Create Channel dialog, add a name for the channel and then choose **Web** as the channel type. When you click **Create**, Bots generates the App Id. You then substitute this value for `<app-id>` in the code.

Updating the Script Tag

Step 1: Include the Bots Client SDK for JavaScript in Your Web Page

Add the following code towards the end of the `head` section on your page and replace `<sdk-folder-url>` with the URL where the SDK is hosted.

```
<script>
  !function(e,t,n,r){
    function s(){
     try{
       var e;
        if((e="string"==typeof this.response?
JSON.parse(this.response):this.response).url){
        var n=t.getElementsByTagName("script")[0],r=t.createElement("script");
        r.async=!0,r.src=e.url,n.parentNode.insertBefore(r,n)
      }
     }
     catch(e){}}var o,p,a,i=[],c=[];e[n]={init:function(){o=arguments;
      var e={then:function(t){
        return c.push({type:"t",next:t}),e
        } ,catch:function(t){return c.push({type:"c",next:t}),e}};
        return e},on:function(){
         i.push(arguments)},render:function(){p=arguments},destroy:function()
{a=arguments}
        } ,e.__onWebMessengerHostReady__=function(t){
         if(delete e.__onWebMessengerHostReady__,e[n]=t,o)for(var
r=t.init.apply(t,o),s=0;s<c.length;s++){
          var u=c[s];
           r="t"===u.type?r.then(u.next):r.catch(u.next)
           } p&&t.render.apply(t,p),a&&t.destroy.apply(t,a);
         for(s=0;s<i.length;s++)t.on.apply(t,i[s])};
         var u=new XMLHttpRequest;u.addEventListener("load",s),u.open("GET",r+"/
loader.json",!0),u.responseType="json",u.send()
```

```
        }
     (window,document,"Bots", "<sdk-folder-url>");
</script>
```

Step 2: Initialize the Bots Client SDK for JavaScript with Your New App ID

Next, initialize the SDK by adding the following snippet near the end of the `body` section of your page. and replace `<app-id>` with your App Id for the Web channel found in your app settings page.

```
<script>
    Bots.init({appId:'<app-id>'});
</script>
```

Customization

- Embedded Mode
- Strings Customization
- Date Localization
- Sound Notification

Embedded Mode

To embed the widget in your existing markup, you need to pass embedded: true when calling `Bots.init`. By doing so, you are disabling the auto-rendering mechanism and you will need to call `Bots.render` manually. This method accepts a DOM element which will be used as the container where the widget will be rendered.

```
Bots.init({
    appId: '<app-id>',
    embedded: true
});


Bots.render(document.getElementById('chat-container'));
```

> **Note:**
>
> The embedded widget will take full width and height of the container. You must give it a height, otherwise, the widget will collapse.

Strings Customization

Bots lets you customize any strings it displays by overwriting its keys. Simply add the `customText` key in your `Bots.init()` call and specify new values for the keys used in Bots. You can find all available keys here. If some text is between `{}`, or if there is an html tag such as `<a>`, it needs to stay in your customized text.

For example:

```
Bots.init({
    appId: <'app-id'>,
    customText: {
        actionPostbackError: 'An error occurred while processing your action. Please
try again.',
```

```
        clickToRetry: 'Message not delivered. Click to retry.',
        conversationTimestampHeaderFormat: 'MMMM D YYYY, h:mm A',
        fetchHistory: 'Load more',
        fetchingHistory: 'Retrieving history...',
        headerText: 'How can we help?',
        inputPlaceholder: 'Type a message...',
        invalidFileError: 'Only images are supported. Choose a file with a supported
extension (jpg, jpeg, png, gif, or bmp).',
        introductionText: 'We\'re here to talk, so ask us anything!',
        locationNotSupported: 'Your browser does not support location services or
it’s been disabled. Please type your location instead.',
        locationSecurityRestriction: 'This website cannot access your location.
Please type your location instead.',
        locationSendingFailed: 'Could not send location',
        locationServicesDenied: 'This website cannot access your location. Allow
access in your settings or type your location instead.',
        messageError: 'An error occurred while sending your message. Please try
again.',
        messageIndicatorTitlePlural: '({count}) New messages',
        messageIndicatorTitleSingular: '({count}) New message',
        messageRelativeTimeDay: '{value}d ago',
        messageRelativeTimeHour: '{value}h ago',
        messageRelativeTimeJustNow: 'just now',
        messageRelativeTimeMinute: '{value}m ago',
        messageTimestampFormat: 'hh:mm A',
        messageSending: 'Sending...',
        messageDelivered: 'Delivered',
        sendButtonText: 'Send',
        settingsHeaderText: 'Settings',
        tapToRetry: 'Message not delivered. Tap to retry.',
        unsupportedMessageType: 'Unsupported message type.',
        unsupportedActionType: 'Unsupported action type.'
    }
});
```

Date Localization

When you translate the user interface by customizing strings, you might also want to show the date and time in the target language as well. To do this, pass `locale` at initialization time. You might also want to override the timestamp format to match your language.

```
Bots.init({
    appId: <'app-id'>,
    locale: 'fr-CA',
    customText: {
        // ...
        conversationTimestampHeaderFormat: 'Do MMMM YYYY, hh:mm',
        // ...
    }
});
```

The locale options is using the language-COUNTRY format. You can find language codes here and country codes here. The country part is optional, and if a country is either not recognized or supported, it will fallback to using the generic language. If the language isn't supported, it will fallback to en-US. A list of supported locales can be found on the date—nfs Github repository.

> **Note:**
>
> The `locale` option only affects date and time localization, not the strings.

Sound Notification

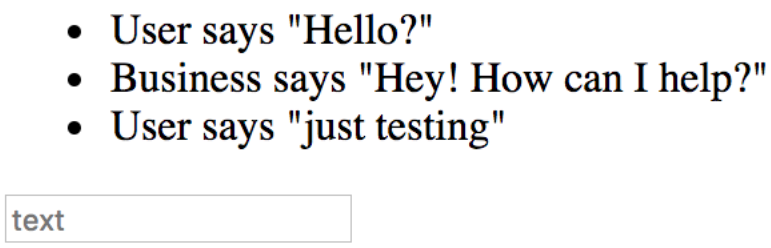
By default, a sound notification will be played when a new message comes in and the window is not in focus.

To disable this feature, you need add the `soundNotificationEnabled` option to the `Bots.init` call, like this:

```
Bots.init({
    appId: <'app-id>',
    soundNotificationEnabled: false // Add this line to your 'Bots.init' call
});
```

Creating a Custom User Interface with the Bots Client SDK for JavaScript

While the SDK's widget provides a rich, prebuilt UI, you can build your own using the SDK's APIs for sending messages and its callback event interface for receiving messages. Using the following snippet as a starting point, you can build a simple user interface that looks something like this.



You'll update the `<body>` and `<script>` elements of this snippet to enable the app to do the following:

- Initialize the Bots Client SDK for JavaScript in Embedded Mode
- Fetch the Initial Data
- Send Messages
- Receive Messages
- Add Postback Actions

You can see the complete code sample here. This app outputs a simple text message. You can find out how to add more complex message types and actions, see Message Types.

> **Note:**
>
> You need to update `SDK_FOLDER_URL` with the URL where the SDK is hosted.

```
<!DOCTYPE html>
 <html lang="en">
 <head>
   <meta charset="UTF-8">
   <title>Document</title>
 </head>
 <body>
   <script>
     !function(e,t,n,r){
         function s(){try{var e;if((e="string"==typeof this.response?
JSON.parse(this.response):this.response).url){var n=t.getElementsByTagName("script")
[0],r=t.createElement("script");r.async=!
0,r.src=e.url,n.parentNode.insertBefore(r,n)}}catch(e){}}var
o,p,a,i=[],c=[];e[n]={init:function(){o=arguments;var e={then:function(t){return
c.push({type:"t",next:t}),e},catch:function(t){return
c.push({type:"c",next:t}),e}};return e},on:function()
{i.push(arguments)},render:function(){p=arguments},destroy:function()
{a=arguments}},e.__onWebMessengerHostReady__=function(t){if(delete
e.__onWebMessengerHostReady__,e[n]=t,o)for(var r=t.init.apply(t,o),s=0;s<c.length;s+
+){var u=c[s];r="t"===u.type?
r.then(u.next):r.catch(u.next)}p&&t.render.apply(t,p),a&&t.destroy.apply(t,a);for(s=0
;s<i.length;s++)t.on.apply(t,i[s])};var u=new
XMLHttpRequest;u.addEventListener("load",s),u.open("GET",r+"/loader.json",!
0),u.responseType="json",u.send()
     }
(window,document,"Bots", "<SDK_FOLDER_URL>");
 </script>
 </body>
 </html>
```

Initialize the Bots Client SDK for JavaScript in Embedded Mode

To initialize the SDK but prevent the default UI from displaying:

1. Create a container that prevents the widget from displaying. In the `<body>` element, define the `<div>` tag that hides the default widget.

   ```
   <div id="no-display" style="display:none;"></div>
   ```

2. In the `<script>` element, initialize the Bots Client SDK for JavaScript in embedded mode and render the "`no-display`" element:

   ```
   Bots.init({ appId: appId, embedded: true });
   Bots.render(document.getElementById('no-display'));
   ```

Fetch the Initial Data

To determine the initial state of the UI, use the SDK's `Bots.getConversation` method. This method provides access to things like the unread message count and the conversation history.

1. Display the conversation by adding the following tag in the `<body>`:

   ```
   <ul id="conversation"></ul>
   ```

2. Within the `<script>` tag, define a function that when called, displays a message in the custom UI.

   ```
   function displayMessage(message) {
     var conversationElement = document.getElementById('conversation');
     var messageElement = document.createElement('li');
     messageElement.innerText = message.name + ' says "' + message.text + '"';
   ```

```
  conversationElement.appendChild(messageElement);
}
```

3. To display the initial conversation state after the initialization of the SDK, replace the `Bots.init` call with the following:

```
Bots.init({ appId: appId, embedded: true }).then(function() {
  // displays initial messages
  var conversation = Bots.getConversation();
  conversation.messages.forEach(displayMessage);
});
```

Send Messages

To enable the widget to send messages:

1. Create a text input element in the `<body>` that allows the widget to accept plain text messages:

```
<input type="text" id="text-input" placeholder="text"
```

2. In the `<script>` tag, add the following element that calls `Bots.sendMessage` function right after the `Bots.init` call. When the text input element is active, this function, which enables users to send plain text or structured messages, gets called whenever a user taps Enter:

```
var inputElement = document.getElementById('text-input');

inputElement.onkeyup = function(e) {
  if (e.key === 'Enter') {
    Bots.sendMessage(inputElement.value)
      .then(function() {
        inputElement.value = '';
      });
  }
}
```

Receive Messages

To update the UI with the new message content, use the SDK's `Bots.on` event interface to bind the `message:received` event (inbound messages) and the `message:sent` event (outbound messages) to the `displayMessage` function. To call this function whenever these message events occur, add the following somewhere after the `Bots.init` call.

```
Bots.on('message:sent', displayMessage);
Bots.on('message:received', displayMessage);
```

Add Postback Actions

You can add actions to the bot's reply message by updating the `displayMessage` function.

```
function displayMessage(message) {
    var conversationElement = document.getElementById('conversation');
    var messageElement = document.createElement('li');
    messageElement.innerText = message.name + ' says "' + message.text + '"';

    if(message.actions && message.actions.length > 0){
        var wrapperElement = document.createElement('div');
        for(var i = 0; i < message.actions.length; i++){
            var action = message.actions[i];
```

```
            var btnElement = createButtonElement(action);
            wrapperElement.appendChild(btnElement);
        }
        messageElement.appendChild(wrapperElement);
    }
    conversationElement.appendChild(messageElement);
}

function createButtonElement(action) {
    var btnElement = document.createElement('button');
    var btnTitle = document.createTextNode(action.text);
    btnElement.appendChild(btnTitle);
    btnElement.onclick = function(e){Bots.triggerPostback(action._id);};
    return btnElement;
}
```

Calling Other Functions

You can call other functions after the SDK has been initialized successfully. For example, you can update a user profile by calling `updateUser`.

```
Bots.updateUser({

    "givenName":"John",
    "surname":"Smith",
    "email": "john.smith@example.com",
    "properties": {
       "BotsCustomVariable1":"Lord",
       "BotsCustomVariable2":"Commander"
     }
     }).catch(function (err) {
        console.error(err);
     });
```

Sample Code for the Custom UI

```
<!DOCTYPE html>
 <html lang="en">
 <head>
   <meta charset="UTF-8">
   <title>Document</title>
 </head>
 <body>
   <div id="no-display" style="display:none;"></div>
   <p>User ID: <span id="user-id"></span></p>
   <ul id="conversation"></ul>
   <input type="text" id="text-input" placeholder="text">

  <script>
        var appId = '<APP_ID>';
     !function(e,t,n,r){
          function s(){try{var e;if((e="string"==typeof this.response?
JSON.parse(this.response):this.response).url){var n=t.getElementsByTagName("script")
[0],r=t.createElement("script");r.async=!
0,r.src=e.url,n.parentNode.insertBefore(r,n)}}catch(e){}}var
o,p,a,i=[],c=[];e[n]={init:function(){o=arguments;var e={then:function(t){return
c.push({type:"t",next:t}),e},catch:function(t){return
c.push({type:"c",next:t}),e}};return e},on:function()
{i.push(arguments)},render:function(){p=arguments},destroy:function()
{a=arguments}},e.__onWebMessengerHostReady__=function(t){if(delete
e.__onWebMessengerHostReady__,e[n]=t,o)for(var r=t.init.apply(t,o),s=0;s<c.length;s+
```

```
+){var u=c[s];r="t"===u.type?
r.then(u.next):r.catch(u.next)}p&&t.render.apply(t,p),a&&t.destroy.apply(t,a);for(s=0
;s<i.length;s++)t.on.apply(t,i[s])};var u=new
XMLHttpRequest;u.addEventListener("load",s),u.open("GET",r+"/loader.json",!
0),u.responseType="json",u.send()
     }
(window,document,"Bots", "<SDK_FOLDER_URL>");

  inputElement.onkeyup = function(e) {
  if (e.key === 'Enter') {
  Bots.sendMessage(inputElement.value)
  .then(function() {
  inputElement.value = '';
  });
  }
  }
  // display new messages
  Bots.on('message:sent', displayMessage);
  Bots.on('message:received', displayMessage);

  // initialize Bots and render the UI in a hidden element
  Bots.init({ appId: appId, embedded: true })
  .then(function () {

  // displays initial messages
  var conversation = Bots.getConversation();
  conversation.messages.forEach(displayMessage);

  Bots.render(document.getElementById('no-display'));

  function displayMessage(message) {
  var conversationElement = document.getElementById('conversation');
  var messageElement = document.createElement('li');
  messageElement.innerText = message.name + ' says "' + message.text + '"';
  conversationElement.appendChild(messageElement);
  }
    </script>
 </body>
 </html>
```

Message Types

The custom UI supports the these messaging types:

- Text Message
- Carousel Message
- Image Message
- File Message
- Location Message

Text Message

A text type message is sent with text and/or actions.

```
{
    /**
     * The text content of the message. Optional only if actions are provided.
     */
    text?: string,
    /**
```

```
     * Message type
     */
    type: 'text',
    /**
     * Message role can be 'appUser' or 'appMaker'
     * Added by SDK when send through the sendMessage method
     */
    role?: 'appMaker',
    /**
     * Array of action buttons.
     */
    actions?: MessageAction[],
    /**
     * Url to the avatar for this message sender
     */
    avatarUrl?: string
}
```

Carousel Message

Carousel messages are a horizontally scrollable set of items, each of which can contain combinations of text, images, and action buttons.

```
{
    /**
     * Message type
     */
    type: 'carousel',
    /**
     * Message role can be 'appUser' or 'appMaker'
     * Added by SDK when send through the sendMessage method
     */
    role?: 'appMaker',
    /**
     * Url to the avatar for this message sender
     */
    avatarUrl?: string,
    /**
    * Array of message items. The array is limited to 10 items.
    */
    items: Item[],
    /**
    * Settings to adjust the carousel layout.
    */
    displaySettings?: {
        imageAspectRatio: 'horizontal' | 'square'
    }
}
```

Image Message

An image type message is a message that is sent with an image, and, optionally, text and/or actions.

```
{
    /**
     * Message type
     */
    type: 'image',
    /**
     * Message role can be 'appUser' or 'appMaker'
     * Added by SDK when send through the sendMessage method
     */
```

```
    role?: 'appMaker',
    /**
     * Url to the avatar for this message sender
     */
    avatarUrl?: string,
    /**
    * The text content of the message. Optional only if actions are provided.
    */
    text?: string,
    /**
    * The media type is defined here, for example image/jpeg. If mediaType is not
specified, the media type will be resolved with the mediaUrl.
    */
    mediaType?: string;
    /**
    * The image URL used for the image message.
    */
    mediaUrl: string;
    /**
    * Array of action buttons.
    */
    actions?: MessageAction[]
}
```

File Message

A file type message is a message that is sent with a file attachment.

```
{
    /**
     * Message type
     */
    type: 'file',
    /**
     * Message role can be 'appUser' or 'appMaker'
     * Added by SDK when send through the sendMessage method
     */
    role?: 'appMaker',
    /**
     * Url to the avatar for this message sender
     */
    avatarUrl?: string,
    /**
    * The text content of the message. Optional only if actions are provided.
    */
    text?: string,
    /**
    * The media type is defined here, for example application/pdf. If mediaType is
not specified, the media type will be resolved with the mediaUrl.
    */
    mediaType?: string;
    /**
    * The URL of the file attachment.
    */
    mediaUrl: string;
}
```

Location Message

A location type message includes the location coordinates (latitude and longitude). Typically, these messages are sent in response to a location request.

```
{
    /**
     * Message type
     */
    type: 'location',
    /**
     * Message role can be 'appUser' or 'appMaker'
     * Added by SDK when send through the sendMessage method
     */
    role?: 'appMaker',
    /**
     * Url to the avatar for this message sender
     */
    avatarUrl?: string,
    /**
    * The coordinates of the location.
    */
    coordinates?: {
        /**
        * A floating point value representing the latitude of the location
        */
        lat: number,
        /**
        * A floating point value representing the longitude of the location
        */
        long: number
    }
}
```

Message Actions

- Postback Action
- Link Action
- Location Request Action
- Reply Action
- Webview Action
- Share Action

Postback Action

A `postback` action posts the action payload when tapped.

```
{
    _id: string,
    /**
     * The button text.
     */
    text: string,
    /**
     * Type of the action
     */
    type: 'postback',
    /**
     * Value indicating whether the action is the default action for a message item.
     */
    default: boolean,
    /**
     * Flat object containing any custom properties associated with the action.
     */
```

```
    metadata?: any,
    /**
    * A string payload to help you identify the action context. You can also use
metadata for more complex needs.
    */
    payload: string
}
```

Link Action

A `link` action opens the provided URI when tapped.

```
{
    /**
     * The button text.
     */
    text: string,
    /**
     * Type of the action
     */
    type: 'link',
    /**
     * Value indicating whether the action is the default action for a message item.
     */
    default: boolean,
    /**
     * Flat object containing any custom properties associated with the action.
     */
    metadata?: any,
    /**
    * The action URI. This is the link that will be used in the clients when
clicking the button.
    */
    uri: string
    /**
    * Extra options to pass directly to the channel API.
    */
    extraChannelOptions?: any
}
```

Location Request Action

A `location` request action prompts users to share their location.

```
{
    /**
     * The button text.
     */
    text: string,
    /**
     * Type of the action
     */
    type: 'locationRequest',
    /**
     * Value indicating whether the action is the default action for a message item.
     */
    default: boolean,
    /**
     * Flat object containing any custom properties associated with the action.
     */
    metadata?: any,
}
```

Reply Action

A `reply` action echoes the user's choice as a message.

> **Tip:**
>
> You can also specify an `iconURL` which renders as an icon for each option.

```
{
    /**
     * The button text.
     */
    text: string,
    /**
     * Type of the action
     */
    type: 'reply',
    /**
     * Value indicating whether the action is the default action for a message item.
     */
    default: boolean,
    /**
     * Flat object containing any custom properties associated with the action.
     */
    metadata?: any,
    /**
    * A string payload to help you identify the action context. Used when posting
the reply. You can also use metadata for more complex needs.
    */
    payload: string,
    /**
    * An icon to render next to the reply option
    */
    iconUrl?: string
}
```

Webview Action

When a user taps or clicks a webview action, the URL is loaded in the webview.

```
{
    /**
     * The button text.
     */
    text: string,
    /**
     * Type of the action
     */
    type: 'webview',
    /**
     * Value indicating whether the action is the default action for a message item.
     */
    default: boolean,
    /**
     * Flat object containing any custom properties associated with the action.
     */
    metadata?: any,
    /**
    * The webview URI. This is the URI that will open in the webview when clicking
the button.
    */
```

```
    uri: string,
    /**
    * The webview fallback URI. This is the link that will be opened when not
support webviews.
    */
    fallback: string,
    /**
    * Controls the webview height.
    */
    size?: 'compact' | 'tall' | 'full',
    /**
    * Extra options to pass directly to the channel API.
    */
    extraChannelOptions?: any
}
```

Share Action

A share button.

```
{
    /**
     * The button text.
     */
    text: string,
    /**
     * Type of the action
     */
    type: 'share',
    /**
     * Value indicating whether the action is the default action for a message item.
     */
    default: boolean,
    /**
     * Flat object containing any custom properties associated with the action.
     */
    metadata?: any
}
```

Message Item

```
{
    /**
     * The image URL to be shown in the carousel/list item.
     */
    mediaUrl?: string,
    /**
     * The text description, or subtitle.
     */
    description?: string,
    /**
     * The title of the carousel item.
     */
    title: string,
    /**
     * If a mediaUrl was specified, the media type is defined here, for example
image/jpeg. If mediaType is not specified, the media type will be resolved with the
mediaUrl.
     */
    mediaType: string,
    /**
     * Array of action buttons. At least 1 is required, a maximum of 3 are allowed.
```

```
link and postback and share actions are supported.
     */
    actions: IBotsSDKMessageAction[],
    /**
     * The size of the image to be shown in the carousel/list item
     */
    size: 'compact' | 'large'
}
```

Display Style Options

You can style the UI by adding these options after `Bots.init`.

Option	Description	Default Value	Required

`displayStyle`	How the messenger widget appears on your website. This is defined as either a button or tab. You can style the button's size and icon:
	• `buttonIconUrl`
	• `buttonWidth`
	• `buttonHeight`
	For example:
	`Bots.init({ appId: appId, embedded: true });`
	`// ...`
	`displayStyle: "button", buttonIconUrl: https://myimage.png, buttonWidth: '90', buttonHeight: '90' ... }).then(function() { // Your code after init is complete });`
		`button`	No

`buttonIconUrl`	The URL that points to the button icon. The icon image must be: • At least 200 x 200 pixels • JPG, PNG, or GIF format		No
`buttonWidth`	The button width, in pixels.	`8px`	No
`buttonHeight`	The button height, in pixels.	`58px`	No

Option	Description	Default Value	Required
`businessName`	The business name. For example: `businessName: “Oracle”`		No

`businessIconUrl`	The URL that points to the business’ icon image. This image must be:
	• At least 200 x 200 pixels
	• JPG, PNG, or GIF format
	For example:
	`Bots.init({ appId: appId, embedded: true });`
	`//...`
	`businessName: "Acme Corporation",`
	`businessIconUrl: "https://example.com/image/thumb/thatimage.jpg/1200x630bb.jpg"`
	`//...`
	`}).then(function() {`
	`// Your code after init is complete`
	`});`
			No

Option	Description	Default Value	Required

`backgroundImageUrl`	The URL that points to the image that appears in the background of the conversation. This image is tiled to fit the chat window.
	For example:
	`Bots.init({ appId: appId, embedded: true });`
	`// ...`
	`backgroundImageUrl: "https://a-nice-texture.png"`
	`// ...`
	`}).then(function() {`
	`// Your code after init is complete`
	`});`
			No

`integrationOrder`	An array of integration IDs. When set, only integrations from this list will be displayed. If the array is empty, then , no integrations will be displayed. **Note**: Listing an integration in the array doesn't guarantee that it will be displayed in the widget.		No
`customColors`	The colors used in the Web Messenger UI.	The three-to-six character hexadecimal colors used for the `brandColor`, `conversationColor`, and `actionColor` options.	No
`brandColor`	The color used in the messenger header and for the button or tab in idle state	`65758e`	No
`conversationColor`	This color used for customer messages, quick replies and actions in the footer.	`0099ff`	No

Option	Description	Default Value	Required
`actionColor`	The color used to change the appears of selected actions inside your messages, like tapped buttons or links. This color is also used for the Send button when it is in active state.	`0099ff`	No

11

Quality Reports

Bots that can distinguish between its intents easily will have fewer intent resolution errors and better user adoption. The quality reports can help you reach these goals.

You can use these reports when you're forming your training data and later on, when you've published your bot and want to find out how your intents are fielding customer messages at any point in time.

How Do I Use the Data Quality Reports?

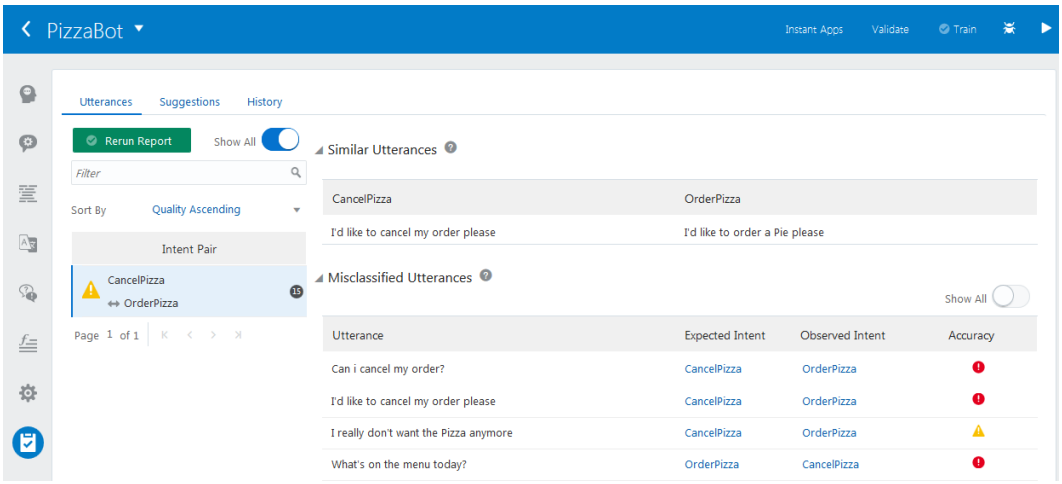
Using the Utterances, Suggestions, and History pages, you can find out if your bot has a sufficient number of intents in the first place, and if so, if these intents overlap, need editing, or if they're behaving as expected in a production environment.

- Utterances—Assigns quality rankings to pairs of intents as follows:
  - High—The intents are distinct.
  - Medium—The intents have similar utterances.
  - Low—The intent pairs aren't differentiated enough.

  You can edit or delete utterances from this page.
- Suggestions—Tells you if your bot is viable. You can find out if you've added enough intents and if you defined a sufficient number of utterances for each intent.
- History—Shows your bot's resolution history, so that you can identify when the intents worked as expected and when they didn't. You can use this feedback to retrain your bot.

Utterances

When you're building your training corpus, you can gauge how distinct your intents are from one another by running an utterance quality report. This report shows you different combinations of intent pairs, each rated on the similarity of their respective utterances. It generates these results by randomly splitting the utterances into two sets: training and testing. It builds and trains a model from 80% of the utterances and then uses the remaining 20% to test this model. If you don't already have a lot of training data, you can build high-quality intents by combining this report with the utterance guidelines.

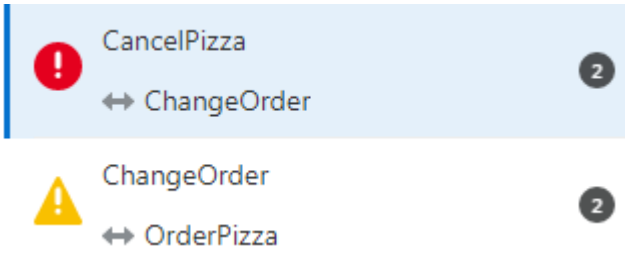


Run an Utterance Quality Report

Use this report to find out which utterances are too alike or can be potentially misclassified (associated with the wrong intent).

> **Important:**
>
> Before you run a report, you need to train your bot with Trainer Tm.

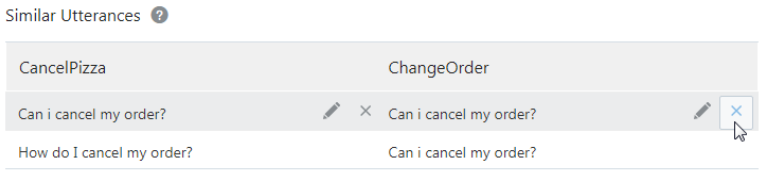
1. After the training completes, click **Quality** ( ) in the left navbar.
2. Click **Run Report**. The report scores the intent pairs in terms of their utterances that are too alike.



Score	What Does this Mean? (And What Do I Do?)
High	While your bot can easily distinguish between these intents, they still may have utterances that are too alike, so you should continue to edit and add training data.
Medium	The utterances are so similar that they potentially blur the meanings of these intents. Because your bot may have trouble distinguishing between these intents, edit or delete these utterances.

Score	What Does this Mean? (And What Do I Do?)
Low	The utterances are too alike, so the bot can't distinguish between them. To fix this, edit or delete these utterances and then retrain the bot. You can also add more utterances to your intent.

3. If needed, click **Show All**. By default, this switch is toggled off ( ), so the report shows only the medium and low-ranking intent pairs. Keep in mind that just because a report ranks an intent pair as high quality, doesn't mean that your corpus is complete, or doesn't need more utterances.
4. If needed, choose a sorting option to view the intent pairs.
5. Click an intent pair to see the similar utterances. By hovering over an utterance, you an edit or delete it.



6. If needed, adjust the utterances.

   Based on the score of the similar entities, your typical course of action is:

   - Adding new utterances.
   - Modifying the similar utterances.
   - Deleting similar utterances.
   - Leaving the utterances alone, even if they clash.
   - Collapsing the two intents into a single intent if they have too many utterances in common. This common intent uses entities (such as list value entities with synonyms) to recognize the distinctions in the user input.

   Important:

   Keep your sights on how modifying the data improves your bot's coverage, not the report results. While you can increase accuracy within the context of this report by adding similar utterances to an intent, you should instead focus on anticipating real-world user input maintaining a diverse set of utterances for each intent. If you pad your intents to suit the report, your bot won't perform well.

7. After you've made your changes, retrain your bot again and then click **Rerun Report**.

Troubleshooting Utterance Quality Reports

Why does the report show similar utterances for high-quality intent pairs?

The report not only compares utterances, but also looks at an intent as a whole. So if most of the utterances for an intent pair are distinct, a low number of similar ones won't detract from the overall quality rating. For example, if you have two intents called FAQ and Balances which each have 100 utterances each. They're easily distinguished, but there are still one or two utterances that belong to each.

Why doesn't the report show similar utterances for low-quality pairs?

This can happen because, on the whole, the report can't distinguish between the intent pair even though they don't share any utterances. Factors like a low number of utterances, or vague and general wording can cause this.

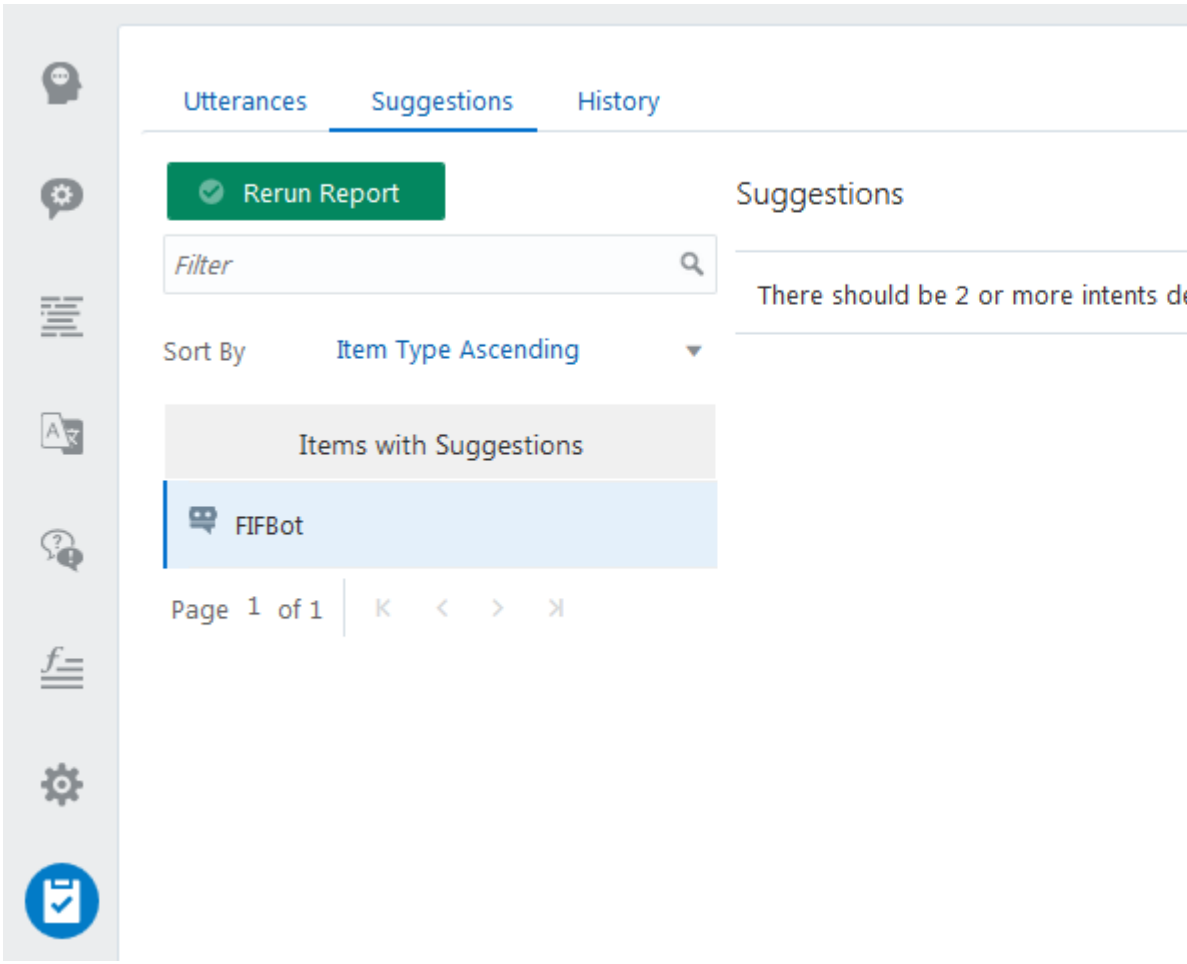
Why does the report continue to show my utterances as similar, even after I edited them?

Whenever you delete or edit and utterance, you need to retrain the bot with Trainer TM before you run the report again.

Suggestions

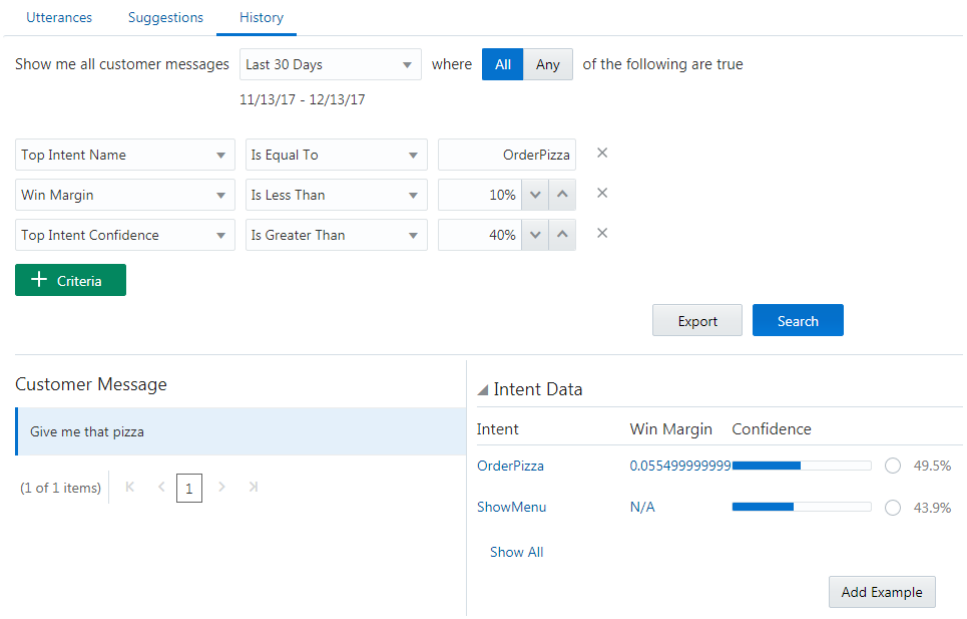
When you're starting out with your data set, check the Suggestions page to find out if your bot meets the minimum standards of having at least two intents, each of which

has two or more utterances.



History

While the Utterance and Suggestion pages help you evaluate your bot as you develop it, you'd use the History page when your training data is robust. The reports that you run from this page return user messages along with the intents that resolved them ranked by win margin and confidence level.

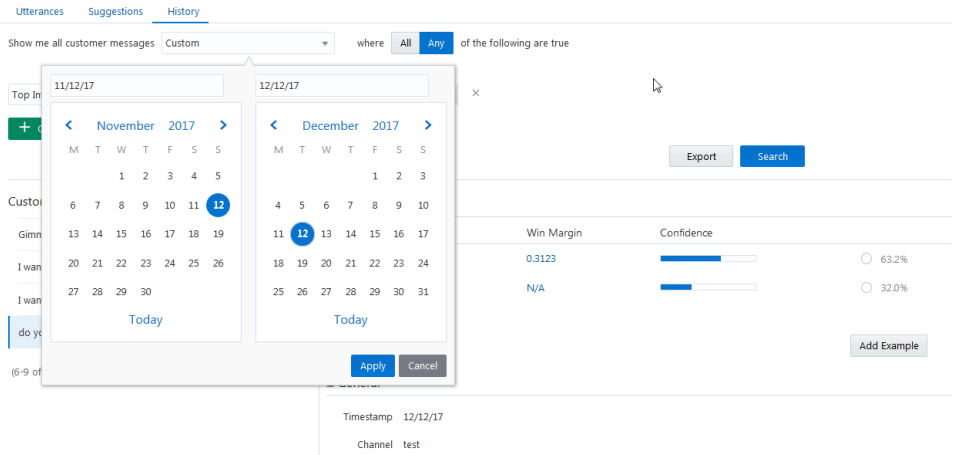


The report is designed to help you look for:

- Complete failures (unresolved intents)—When your bot can't classify the user comment to any of its intents.
- Potentially misclassified user messages—When the top intent is separated from the second intent by only a narrow margin.
- Low confidence levels—When the intended intent resolves the message, but just barely, as indicated by a low confidence level.

How Do I Run a History Report?

1. Choose the time period. You can use one of the preset periods, like Today, Yesterday, or Last 90 Days, or add your own by first choosing **Custom** and then by setting the collection period using the date picker.



> **Tip:**
>
> If you don't want to filter this data any further, then just delete the filter criteria ( ) and then click **Search**.

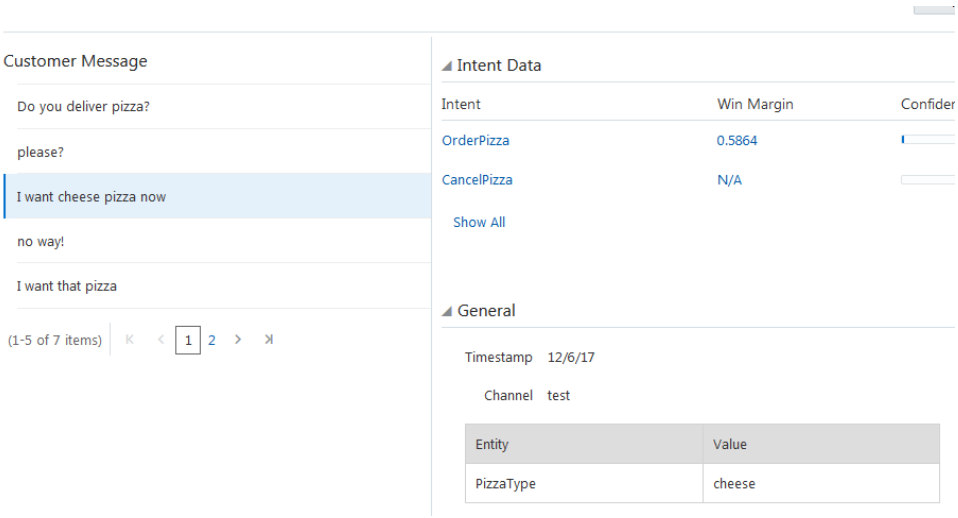
2. If you want to use the report to find out about intents that resolve the messages correctly but with only a low confidence level or by a thin margin, first chose one of the operators (**All** or **Any**) and then apply your search criteria.

3. Click **Search**. For each message within the time frame, the report shows you which intent your bot used to resolve the message along with the second-runner up. To reflect the intent ranking, the report shows you the intents' confidence ranking and, for the top intent, it's win margin, the difference, in terms of confidence, between it and the second intent.

> **Tip:**
>
> In general, you set the win margin at around 10%.

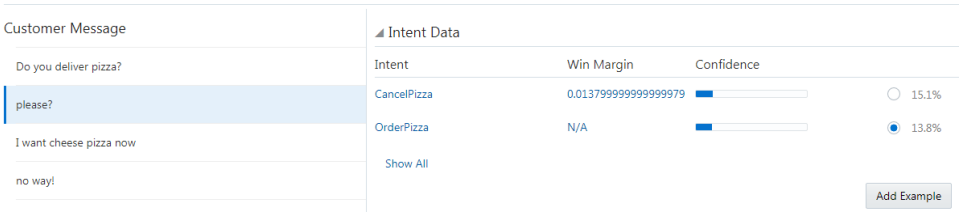
If you click **Show All**, you can see the lower-ranking intents (if any).

By expanding the General section of page, you can see which entities played a role in resolving the message and the channel (which you can set as a filter).



4. If you think the message improves your corpus by say, widening the win margin between the top two intents, select the intent's confidence level radio button and

then click **Add Example**. Remember that since you've now added a new utterance to your corpus, you need to retrain your bot.



Running Failure Reports

To identify all of the messages that your bot treated as unresolved because the resolution fell below the confidence threshold, set **Top Intent Confidence** to a value lower than the one set for the `System.Intent`'s `confidenceThreshold` property. You can add the messages returned by this report to an existing intent, or if they indicate that users want your bot to perform some other action entirely, you can use them to define a new intent.

Running Low Confidence Reports

When the top intent resolves the message, but only with a low confidence might indicate that you might need to revise the utterances that belong to the intents because they're potentially misclassified. To run a report of low confidence intents, set **Top Intent Confidence** equal to a value that's just above the one set for the `System.Intent`'s `confidenceThreshold` property.

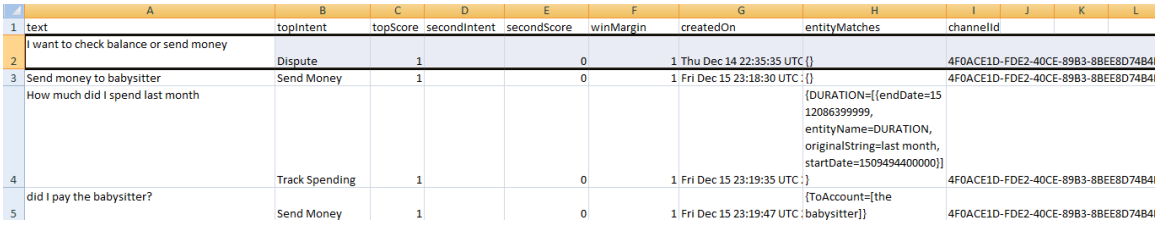
Troubleshooting Narrow Win Margins

Thin win margins might indicate where user messages fall in between your bot's intents. Review these messages to make sure that they are getting resolved by the right intent. You can also configure the `System.Intent`'s confidenceWinMargin property to help your bot respond to vaguely worded or compound user messages.

Viewing the Resolution History as a CSV File

If you prefer to analyze the report results using spreadsheet, then you can export the report as a CSV file when you click **Export**. The CSV sorts the report data for the top two intents using these columns:

- text
- topInent
- topScore
- secondIntent
- secondScore
- winMargin
- createdOn
- entityMatches
- channelId



	A	B	C	D	E	F	G	H	I	J	K	L
1	text	topIntent	topScore	secondIntent	secondScore	winMargin	createdOn	entityMatches	channelId			
2	I want to check balance or send money	Dispute	1		0	1	Thu Dec 14 22:35:35 UTC	{}	4F0ACE1D-FDE2-40CE-89B3-8BEE8D74B4			
3	Send money to babysitter	Send Money	1		0	1	Fri Dec 15 23:18:30 UTC	{}	4F0ACE1D-FDE2-40CE-89B3-8BEE8D74B4			
4	How much did I spend last month	Track Spending	1		0	1	Fri Dec 15 23:19:35 UTC	{DURATION=[{endDate=1512086399999, entityName=DURATION, originalString=last month, startDate=1509494400000}]}	4F0ACE1D-FDE2-40CE-89B3-8BEE8D74B4			
5	did I pay the babysitter?	Send Money	1		0	1	Fri Dec 15 23:19:47 UTC	{ToAccount=[the babysitter]}	4F0ACE1D-FDE2-40CE-89B3-8BEE8D74B4			

12

Bots Analytics

Use the Analytics Collector API to send events to an analytics app via a Bots custom component.

Adding Analytics to the PizzaBot Sample Bot

OMCe platform APIs, including the Analytics API, are available to a custom component via the `conversation.oracleMobile` object.

Use `conversation.oracleMobile.analytics.postEvent()` to send data about your bot and how it's being used to OMCe Analytics. You just need to make sure that the OMCe Analytics application is associated with the backend that's running the custom component, and that you configure a custom schema for the Analytics application to handle the data.

To demonstrate how to add Analytics to a bot, we'll use the PizzaBot sample bot.

Setting up the PizzaBot Analytics Application

Set up the Analytics application in OMCe Analytics and attach it to the backend that runs the custom component.

1. Open OMCe Analytics and create an application with the name *PizzaAnalytics*.
2. Add a new property of type Metric, and give it the name *age*.

> **Note:**
>
> For more information about custom schemas, see Custom Schemas in *Developing Applications with Oracle Mobile Cloud Enterprise*.

3. Click **Settings** then click **Application** and copy the Application ID.
4. Open OMCe Mobile Apps and open the backend that runs the custom components for the PizzaBot bot.
5. Click **Settings** and in the Applications ID field, paste the Application ID that you copied earlier.

After the Analytics application and the backend are configured, set up your custom code to post an event to the Analytics application. You do this by modifying the existing `age_checker.js` custom component code.

Setting up the PizzaBot Custom Component

Add analytics to the AgeChecker custom component by modifying the code in `age_checker.js`. You add code that takes the data that comes from the bot, and uses the Analytics API to send it to the analytics application.

1. Open OMCe and go to **Mobile Apps** then **APIs**.
2. Open the API that contains the custom components that the PizzaBot sample bot uses. If you're not sure which API it is, open the backend that runs the custom components for the bot, and expand the Dependencies section where you'll see the API that contains the custom components.
3. Click **Implementation**. When the Implementation panel opens click **bots_samples** and save the downloaded `api_implementation.zip` file in a convenient location on your hard drive. Keep this page open because you'll need it in a later step when you upload the modified implementation.
4. Extract the files from `api_implementation.zip` and open `api_implementation/sample_bots/pizza/age_checker.js`.
5. Add the following two functions, postEvent and postCustomAnalyticEvents, to the bottom of `age_checker.js`:

```
/**
 * Posts a single custom analytics event, with a single custom property.
 * @param {object} analytics the custom code SDK analytics object,
    usually obtained from conversation.oracleMobile.analytics
 * @param {string} eventName the name of the custom event
 * @param {string} customProperty the name of the custom property
 * @param {string} customValue the value of the custom property
 * @returns {object} a Promise
 */
var postEvent = function (analytics, eventName, customProperty, customValue) {
    const timestamp = (new Date()).toISOString();
    return postCustomAnalyticEvents(
        analytics,
        {
            "name": eventName,
            "type": "custom",
            "timestamp": timestamp,
            "properties": { [customProperty] : customValue.toString() } //
custom values must be passed as String
        },
        timestamp
    );
};

/**
 * Posts custom analytics events.
 * @param {object} analytics the custom code SDK analytics object,
    usually obtained from conversation.oracleMobile.analytics
 * @param {object} customEvents either a single custom event
    object or an Array of custom event objects
 * @param {string} sessionStartTimestamp ISO formated String
    representation of a Date object
 * @param {string} [sessionEndTimestamp] ISO formated String
    representation of a Date object
 * @returns {object} a Promise
 */
var postCustomAnalyticEvents = function (analytics, customEvents,
                                         sessionStartTimestamp,
sessionEndTimestamp ) {
    const events = [];
    events.push(
        {
            "name": "sessionStart",
            "type": "system",
```

```
            "timestamp": sessionStartTimestamp
        }
    );
    Array.isArray(customEvents) ? Array.prototype.push.apply(events,
customEvents) : events.push(customEvents);
    events.push(
        {
            name: "sessionEnd",
            type: "system",
            "timestamp": sessionEndTimestamp ? sessionEndTimestamp :
sessionStartTimestamp
        }
    );
    return analytics.postEvent(events);
};
```

> **Tip:**
>
> The functions are generic enough that you can use them unchanged in any custom code, not just the PizzaBot bot.

6. Next, modify the invoke method in `age_checker.js` to call the postEvent function that you just added. Notice the use of `conversation.oracleMobile.analytics` in `postEvent()`.

```
invoke: (conversation, done) => {
    // Parse a number out of the incoming message
    const text = conversation.text();
    const matches = text.match(/\d+/);
    var age = 0;
    if (matches) {
        age = matches[0];
    }

    console.info('AgeChecker: using age=' + age);

    // Set action based on age check
    conversation.transition( age >= 18 ? 'allow' : 'block' );

    // Original code until here. Analytics logic below.

    // captures age in a custom analytics event.
    postEvent(
        conversation.oracleMobile.analytics,
        "pizzaEvent",
        "age",
        age
    ).then(
        function (result) {
            done();
        },
        function (error) {
            console.warn('AgeChecker: error posting analytics.',
                         error.statusCode, error.error);
            done();
        }
    );
}
```

7. Zip up the `api_implementation` folder. Make sure the zip file has the same internal structure as the one that you downloaded.
8. On the Implementation panel, click **Upload an implementation archive** and upload the new `api_implementation.zip` file.

After the implementation is successfully uploaded, the bot is ready for testing.

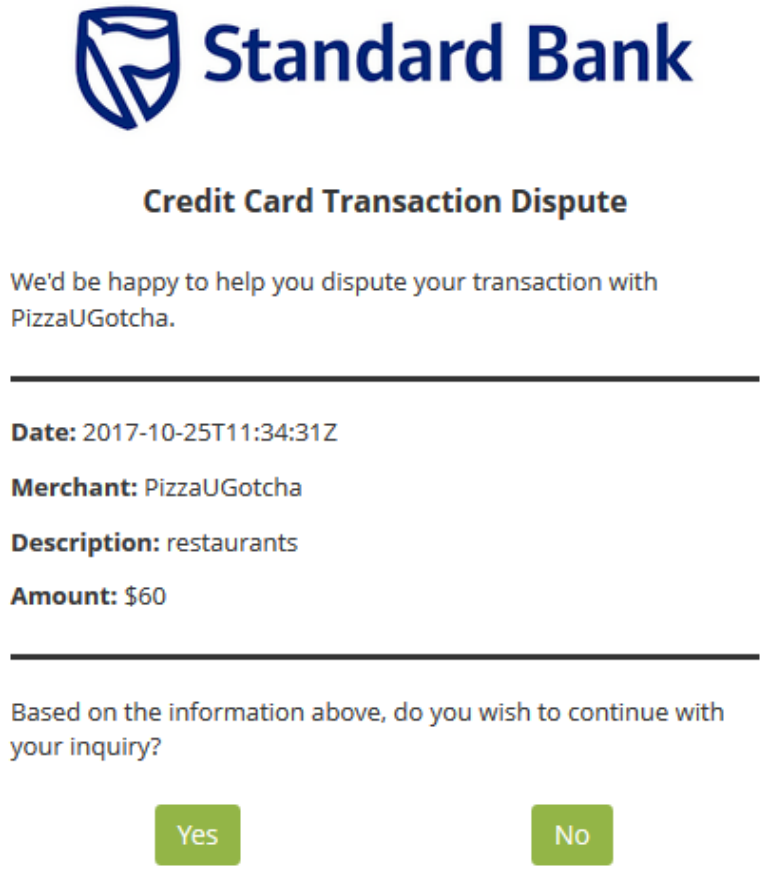
13

Instant Apps

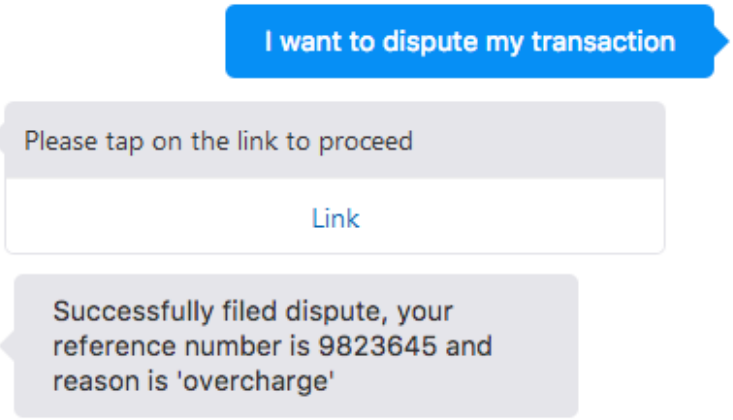
Natural language conversations are, by their very nature, free-flowing. But they may not always be the best way for your bot to collect information from its users. For example, some situations, like entering credit card or passport details, require users to enter specific information (and enter it precisely). To help your bot's users to enter this type of information easily, your bot can call an instant app, which provides forms with labels, options, choices, check boxes, data fields, and other UI elements.

The FinancialBot calls an instant app for the fictitious Standard Bank that walks users through a series of steps to resolve disputed charges. The FinancialBot and the instant app show you how your bot transitions to an instant app, how bots pass variable values to an instant app, and how the instant app returns the user to the bot. See System.Interactive to find out how to embed an instant app in the dialog flow.

You can try this out this wizard-like app using the phrases like the ones defined for the bot's Dispute intent. For example, after querying the checking account balance, you can enter "I want to dispute a charge" to receive a link which in turn opens the instant app in a webview. When the instant app opens, it's populated with values that are passed from the bot: the date, merchant, amount, and description.

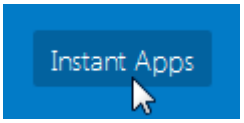


The instant app itself provides a wizard-like experience. When you've finished with it, the instant app executes a callback to relocate you from the webview to your bot, where it displays a confirmation message detailing your transaction. The confirmation message includes the reason for the dispute and the dispute number, both of which are values returned from the instant app.

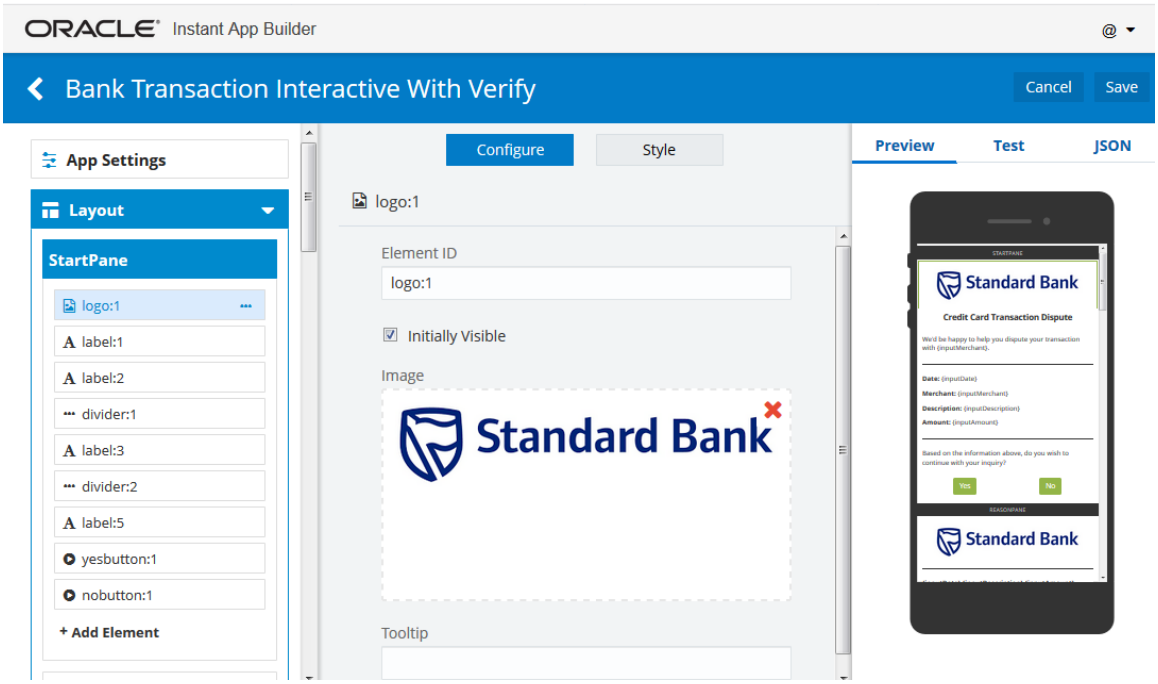


The Instant App Builder

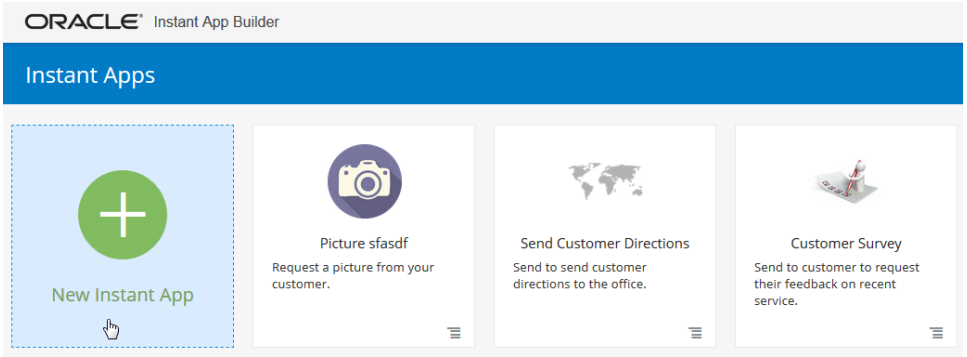
You can build these apps using the Instant App Builder, which you access by clicking **Instant Apps** in the Bots landing page.



Instant apps are made up of sets of panes, which display one at a time. You populate these panes with various elements that can display charts or images and collect customer data using widgets like checkboxes, radio buttons, and file upload functions.



To get you started, you can customize the templates that display in the landing page. You can also use the Standard Bank instant app, which is invoked by the FinancialBot, as a reference. You can also start from scratch by clicking the **New Instant App** tile. See System.Interactive to find out how to integrate your instant app into the OBotML definition.



Creating an Instant App from Scratch

App Settings

App Settings is where you manage general information about your instant app.

Name

The instant app name is your internal way of identifying this instant app among all your others on the instant apps main page. The name can include letters, numbers, and special characters. The name is not exposed to the end user, as you can see in the image Internal Description below.

ID

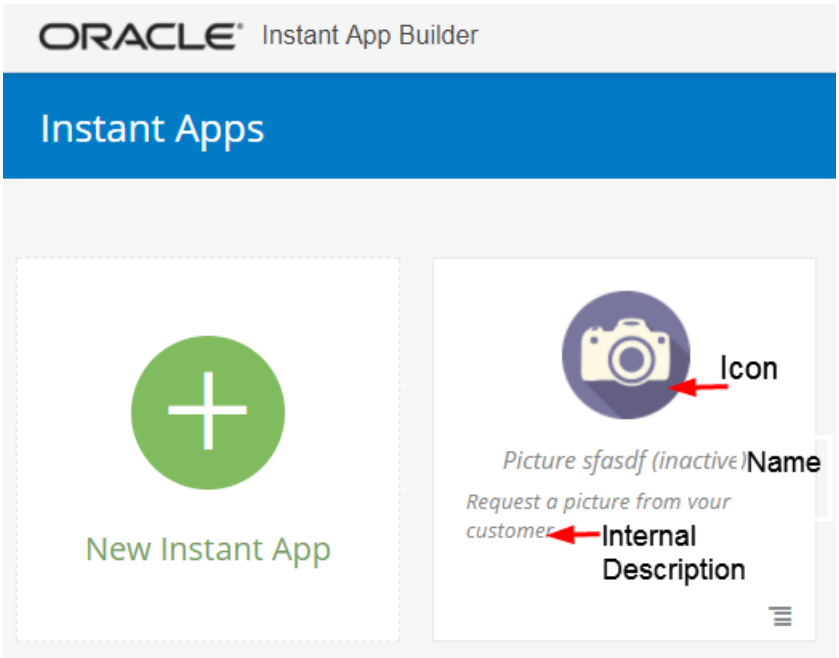
The instant app ID is how you reference the instant app if you need to call it from somewhere, like from a bot, an API, or a JavaScript Snippet. When you create a new instant app, the ID itself is derived from the instant app name that you enter. The ID cannot contain special characters or spaces. You can edit the ID at any point, but if you do change it, you will need to update any references to the previous ID.

Icon

An icon is the image that shows up on the instant app tile on the main instant apps page. Remove unwanted icons by clicking on the red X in the top right corner. Then you can drag and drop an icon, add an icon via regular file lookup, or input a URL.

Internal Description

The internal description is what shows up on the instant app's tile on the main instant apps page as a reminder of the particular instant app's function.

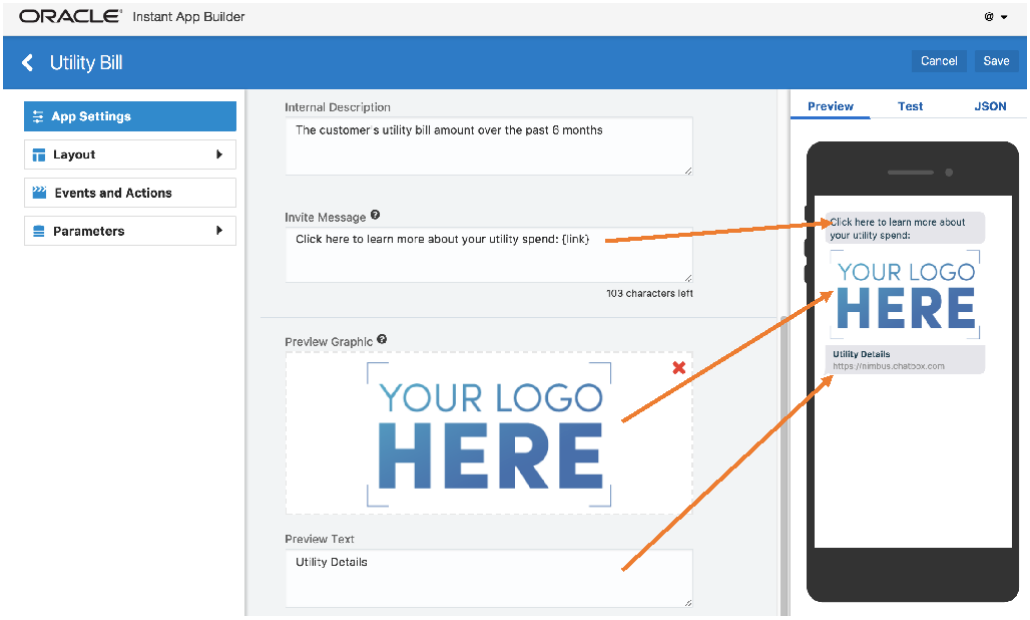


Initially Active

When you create a new instant app, you can set the instant app to be **Initially Active** before you save it. If you set it to active, the instant app can be activated from the bot. If you do not set it to Initially Active, then you can always set it to Active from the instant app tile on the main menu. You can see which apps are inactive by the **Inactive** display next to the instant app name.

Invite Message and Link

The Invite Message is a preconfigured message that is sent to customers inviting them to use the instant app, and it is the first thing a customer sees. Include the {link} in the position where you want the instant app link, and do not change anything else. The message, including the link, cannot exceed 160 characters.



Laying Out an Instant App

Laying out an instant app includes selecting panes, elements, and adding identifying information such as Pane IDs with the instant app builder. Instant app layouts are highly customizable to suit your business needs.

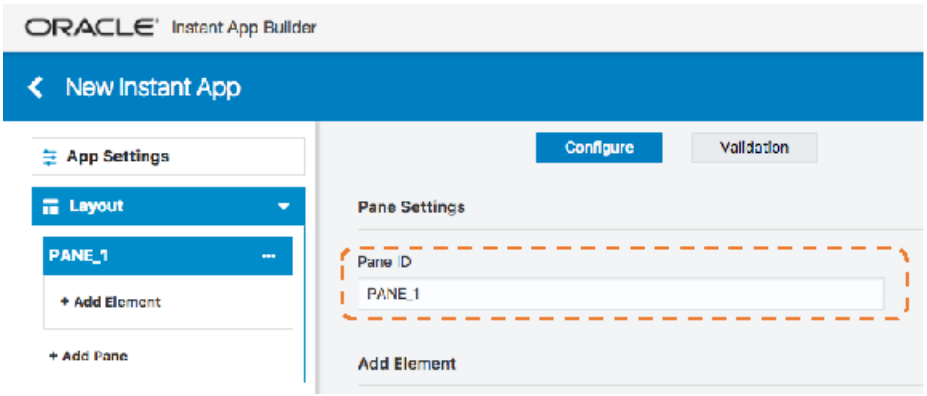
Panes

Panes are essentially the pages of your instant app. Some scenarios require just a single pane experience, where a customer clicks into the instant app, engages, and then is taken back to the bot conversation. More complex instant apps will have multiple panes to minimize the amount of information on a single screen. Panes are flexible and allow you to optimize your customers' experience based on the content you're delivering

When you first click to build out the layout of your instant app, you will see a single pane as a starting point. Before jumping into adding Elements, make sure to edit **Pane**

ID. Renaming **Pane ID** to describe the function or purpose of the pane will make it easier to identify when modifying your instant app in the future.

> **Note:**
>
> Pane IDs can only contain letters and numbers. No spaces or special characters are allowed.



Elements

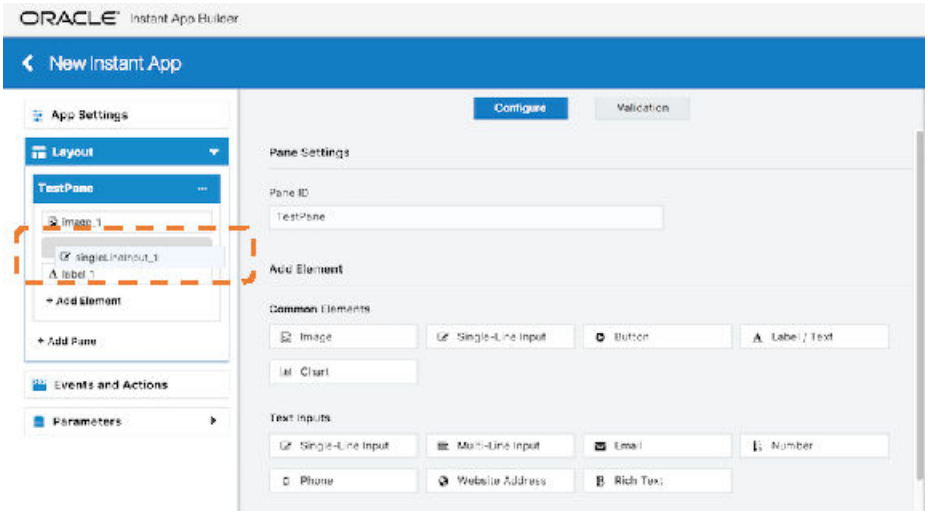
Elements are the components within each pane that make up your instant app. Element types are essentially templates for the types of components in an instant app.

All elements and functions are highly configurable with JSON. Element types are really just templated JSON. They can capture everything from text inputs and labels, to image galleries, photo uploads, and signature captures. This section includes tips on using elements followed by a look into each element type.

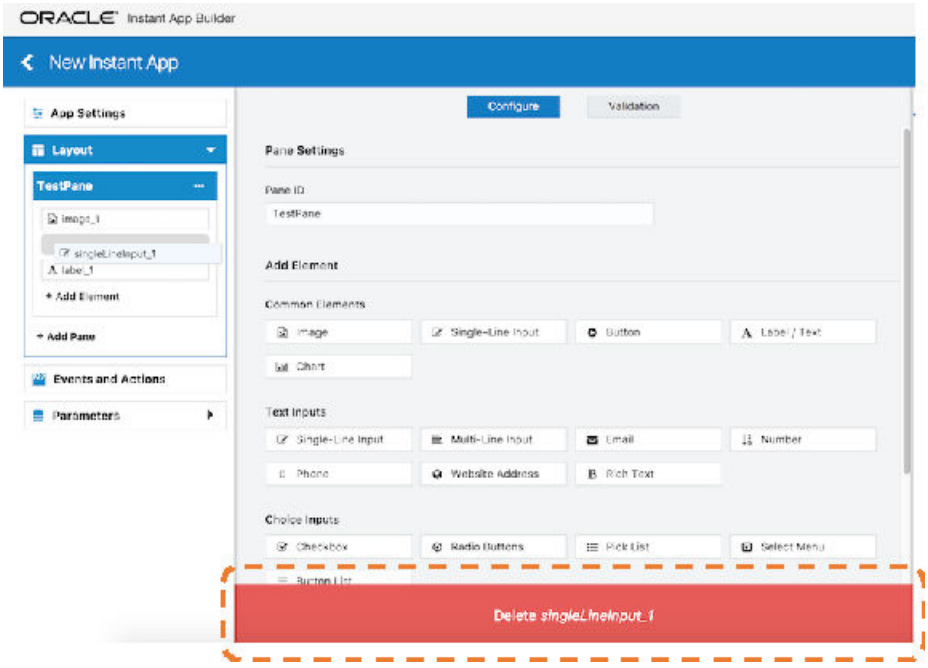
Adding, Moving, and Deleting Elements

To add a single element, click on the element. When you add a single element, you are taken into the configuration menu options for that element.

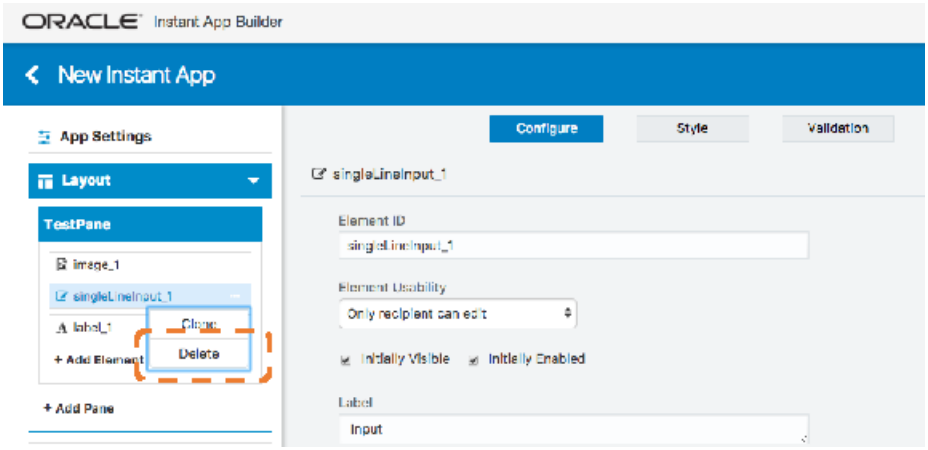
To move an element, click on it and drag and drop to the desired location. Elements can be moved within a pane or across panes



There are two ways to delete an element. When you click on an element and drag it, you will notice a Delete bar on the bottom of your screen. Drag and drop the element to the Delete bar to delete it.



OR, click on the ellipsis next to the element name and click **Delete**.



Cloning Elements

If you're using the same element multiple times across an instant app, you can easily clone the element. To do so, click on the ellipsis to the right of the element name, and choose **Clone**.

> **Note:**
>
> When an element is cloned, all associated configuration, style, and validation is cloned to the new element, but any associated events and actions are not cloned.

Shortkey for Adding Multiple Elements

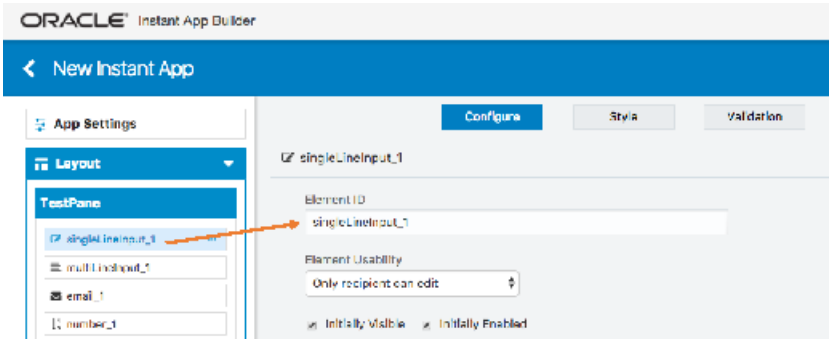
While building your Instant App, there may be times when you know what Elements you want to include and want to add them quickly in bulk. To do this, use Command + Shift and click on each Element you want to add. This will add Elements to your active Pane in the order they're clicked. From there, you can go through each Element to configure it as needed.

Common Configuration

Many elements have some common configurations. Here are the commonalities between elements, followed by specifics for each element type.

Element ID

The Element ID is the name on the left list of elements. This ID is referenced anywhere you are using JavaScript to affect this element.



Initially Visible

You can make each element visible or invisible when the instant app pane loads. If it's invisible when loaded, then a user would not see it unless you create an action to make it visible.

Initially Enabled

You can make an element enabled or disabled when the instant app pane loads. If it's not enabled when loaded, then a user could not input anything into or affect the element. If you would like a user to be able to interact with it, then you would need to set up an action to enable the element.

Label

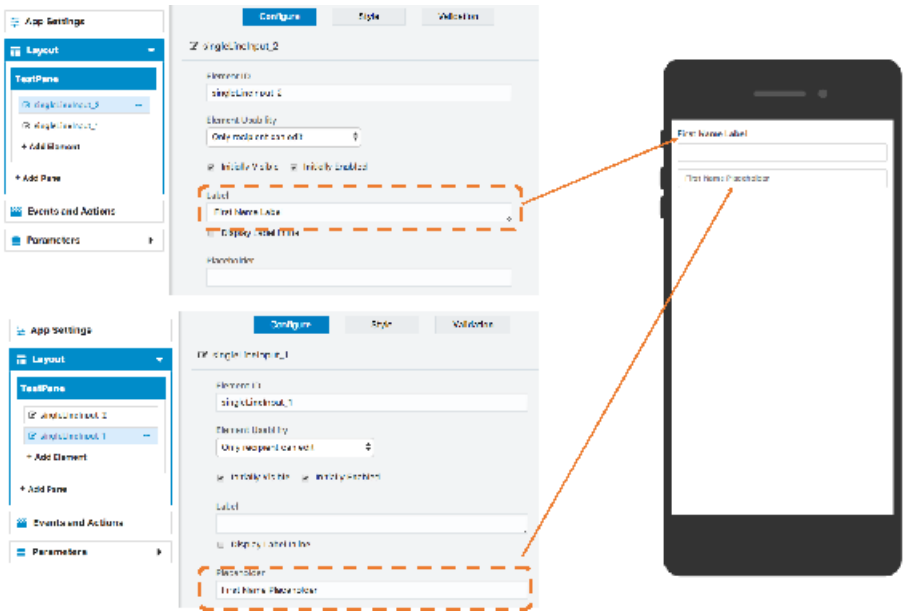
Generally, a label is what you would see above an element that describes what it is or its function. A few elements have unique label behavior which are called out in the specific element. Labels can use HTML.

Display Label Inline

For Elements where the label is listed above it, you can select **Display label inline**, to make the label appear on the same line as the element.

Placeholder

You will see a configuration for placeholder in a few different elements, primarily the text input elements. Any text configured here will show in gray, inside the text input box. Once a customer inputs any text into the field, the placeholder will disappear. A common use for placeholders is to configure an empty label, but be sure to set a label/ description as the placeholder.



Tool Tip

Tool tips can be used to show help text. The tool tip will show when users interact with the element.

Styles

Styles are fairly similar across elements, with only a few variations.

Font

Text fonts and features within elements can be adjusted in these ways:

- **Size**: In Pixels
- **Weight**: Normal, Bold, Light
- **Style**: Normal, Italic, Oblique
- **Alignment**: Left, Center, Right, Justified

Layout

The Layout section of the Style Tab varies between elements. Input for the following options can be given in pixels or as a percent

- Element Width: This is the width of the entire element. Different elements have different defaults. For example, a label defaults to 100%. Setting the element width to a smaller number allows for more than one element to show up on the same line.
- Top Margin: The distance between the previous element and the element you are editing.
- Left Margin: The space between the left edge of the instant app and where the element begins.
- Right Margin: The space between the right edge of the instant app and where the element ends.

- Bottom Margin: The distance between the element you are editing and the element which follows.

The **Inner Dimensions** control the used portion of the width of an element, given the constraints you've created on element width, left and right margins, and the height of the element. For example, if you set the element width of a button to 75 percent and then set the Inner Dimension width to 100 percent, the button will expand to fill the entire 75 percent. As another example, if you set the element width of a Label to be 50 percent the left margin to be 20 px, and the Inner Dimension width to be 50 percent it would result in the label starting 20 px to the right, and only filling up half of the text box, or 25 percent of the width of the instant app since the element width had already been set to 50 percent (50 percent of 50 percent = 25 percent).

Foreground Color and Background Color

The **Foreground Color** generally controls the text in an element and the **Background Color** generally controls the area of the element behind the text. They can differ somewhat between elements because of the variety of element types, so specifics are called out in each element. However, for any of the elements that have buttons embedded in them (like Upload, Location, Signature), the button color can't be changed.

Border

For most elements, you can control whether there is a border around the element or not, and how the border looks. You can adjust the width, whether it is solid, dotted, or dashed, the shape of the corners, and the color. In some element types, for example Images, if you increase the corners sufficiently and adjust the Layout, you can create oval and round shapes.

Element Types

Element types influence validation options described here:

Text Inputs

Single Line Input

Single Line Input is one of the most common elements, and it can also be found in the Common Elements section of your instant app layout pane. This element is used for any text input and can accept letters, numbers, and special characters. A common use for this element is to collect a customer's name. Here are some things to remember about Single Line Input:

- These elements have a maximum length of 256 characters.
- They can have several forms of validation. They can be *required*, have min/max character validations, match a regular expression, or have JavaScript validation. If you want to match a regular expression, add it in the **Regular Expression** field, and if it fails, the user will receive the failure message. You can find more help with Regular Expressions at https://regex101.com/

Multi-line Input

Use a Multi-line Input element for lengthier text. This element is similar to the Single Line Input but with a few different configurations:

- Multi-line Input elements have a maximum input character length of 1000.

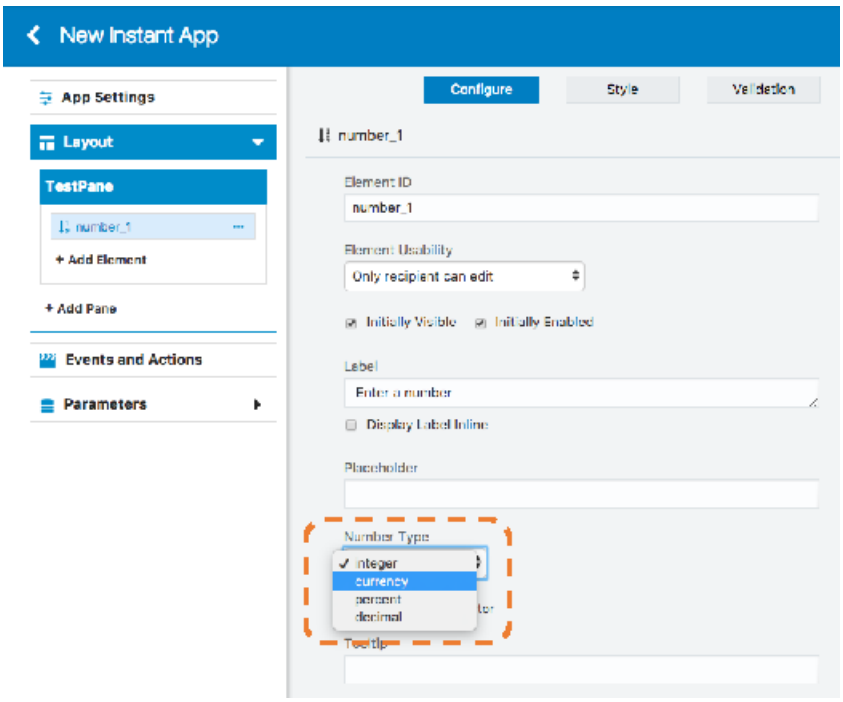
- You can specify how many rows are shown in the display of this element.
- These elements can have several forms of validation. They can be required, have min/max character validations, match a regular expression, or have JavaScript validation. If you want to match a regular expression, add it in the **Regular Expression** field. If it fails, the user will receive a failure message.

Email

Use this element to capture an email address. A customer must include @ and a completed domain to avoid an error message. Email elements can have several forms of validation. They can be required, match a regular expression, or have JavaScript validation. If you want to match a regular expression, add it in the regular expression field.

Number

Use this element to collect integers, currency values, percentages, or decimals. Phone numbers have their own element, described below. The type of number you choose affects the element display in the Instant App and enforces the number type during text input. You can include, or exclude, a common separator for the thousands' place. Number elements can have several forms of validation. They can be required, match a regular expression, or have JavaScript validation.



Phone

This element provides a stylized input field for phone numbers. Customers can use the drop-down menu to set the default country code before entering their phone number. Phone number elements can have several forms of validation. They can be required, match a regular expression, or have JavaScript validation.

Website Address

Use this element when you need to collect website information from your customers. Website inputs must begin with http:// or https:// to be valid, otherwise users will see a malformed url error message. Website address elements can have several forms of validation. They can be required, match a regular expression, or have JavaScript validation.

Rich Text

The Rich Text Element is similar to the Multi-Line Input element, but it allows agents and customers to add HTML formatting inside the text field. Below the element are three icons: the edit, preview, and help icons.

The help icon displays supported formats and samples. These formats include:

-
 line break
- **bold text**
- *italic text*
- <u>underlined text</u>
- centered text
- <h1>header text</h1>

Rich Text Elements can have several forms of validation. They can be required, have min/max character validations, match a regular expression, or have JavaScript validation.

Choice Inputs

Checkbox

The Checkbox Element is typically used for acknowledgement and offer a way for the user to provide confirmation. For multiple confirmations, you can add multiple checkbox elements or use the Pick List Element. The validation used is required/not required.

Radio Buttons

Radio buttons allow users to specify a choice from a list of options. Configure the options available for the user to choose from and specify the Value associated with the Option, which you can link to an action. You should have a minimum of two options, but you can add more as needed.

If you check the box to **Load Options with Action**, you need to specify the values by either adding an Execute JavaScript Snippet action in the App Sent event, or by retrieving the values from a call to an external web API. In either case, the values must be expressed as a JSON object with the following structure

```
{"options":
[{ "label": "Label One", "value": "value1" },
{ "label": "Label Two", "value": "value2" },
{ "label": "Label three", "value": "value3" }
]}
```

The validation supported is required/not required.

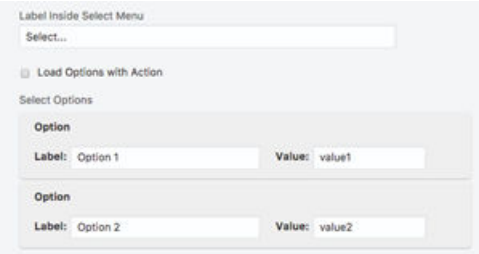
Picklist

A Picklist Element is used when you want a user to choose multiple options. You can configure as many items in the Picklist as needed. If you check the box to **Load Options with Action**, you need to specify the values by either adding an Execute JavaScript Snippet action in the App Sent event, or by retrieving the values from a call to an external web API. In either case, the values must be expressed as a JSON object with the following structure:

```
{"options": [
{ "label": "Label One", "value": "value1" },
{ "label": "Label Two", "value": "value2" },
{ "label": "Label three", "value": "value3" }
]}
```

Select Menu

Select Menu Elements allow users to choose from a drop-down menu. In addition to setting the options for the menu, you also have the ability to set a label inside of the Select Menu box.



If you check the box to **Load Options with Action**, you need to specify the values by either adding an Execute JavaScript Snippet action in the App Sent event, or by retrieving the values from a call to an external web API. In either case, the values must be expressed as a JSON object with the following structure:

```
{"options": [
{ "label": "Label One", "value": "value1" },
{ "label": "Label Two", "value": "value2" },
{ "label": "Label three", "value": "value3" }
]}
```

The validation supported is required/not required.

Special Inputs

Buttons

Buttons are one of the most common elements in instant apps and are typically used at the end of a pane, linked with an event and/or an action. They help a user navigate within an instant app, or to end an instant app experience. You can style your button and adjust button size in style layout. The inner dimension of a button should be set to 100 percent if you want the button to be the width of the device screen.

Upload Photo or File

The Upload Photo or File element allows users to upload an image or a file into the instant app. Here is what the choices mean:

- **Upload Label**: If you enter text in the upload label, you will see it appear above the Waiting for Upload message. This message can not be changed.
- **Button Text**: Set the upload text here that will appear on the button.
- **Show Filename**: Select this option if you want the user to see the name of their uploaded file.
- **File Size Limit**: You can set the limit of the file size you would like to accept. You must enter it in bytes and it defaults to 10000000 (10MB).
- **Show a Preview of the Uploaded Image**: Select this if you would like your user to see their image after it is uploaded. Note: When you are building the Instant App and using either the preview or test mode, you aren't able to actually upload an image. During those simulations, you will just see a placeholder for an image/file.
- **Preview Height Max** and **Preview Width Max**: you may set the size of the preview in pixels. The default is height 200 px, width 300 px .
- The size, weight, and style of the upload label, the message *Waiting for Upload*, and the button text are controlled by the size, weight, and style in the style tab. The **Foreground Color** controls the color of the upload label text and the Waiting for Upload message. The color of the button and button text cannot be changed. The **Background Color** controls the area of the upload element, behind the upload label, Waiting for Upload message, and the button.

- The validation supported is required/not required.

Date

The Date element lets users easily select a date from a dropdown calendar. Since the date is not manually input you don't need to worry about formatting; it will always be formatted the same.

- **Placeholder**: That is the text that will appear in the area a user would touch/click into to select the date. It will be replaced by the date when selected.
- **Get Time**: Select Get Time, if you would like the user to be able to select a time as well as the date.
- **Minimum Date** and **Maximum Date**: When these are set, a user will not be permitted to select a date earlier than the Minimum or later than the Maximum. If you are manually entering the date in the Instant App Builder instead of using the drop-down calendar, it must be in the format MM/DD/YYYY.
- The size, alignment, weight, and style of the placeholder text is controlled on the Style Tab. Placeholder text color cannot be adjusted. The width and margins of the Date Element can be adjusted using pixels or percentage, but if you make the width too small, the user will not be able to see the full date and time. The area of the Date Element behind the placeholder text is controlled by the background color.
- Date elements can be required/not required and you can also write a JavaScript snippet to validate the element.

Signature

The Signature element allows you to capture the user's signature. The user can sign and then clear or confirm their signature. You can adjust the text on the Confirm button (using the Confirm Button Label), but not the Clear button. Both the Clear and Confirm buttons will always be initially disabled and are enabled when a user writes a signature. If the user clears the signature, then both buttons will become disabled again.

- The style aspects of the Signature element that can be controlled through are the background color of the signature box and the foreground color of the signature .
- The validation supported is required/not required.

Star Rating

The Star Rating element allows you collect a rating from your user. This element is very flexible. You are able to set the number of stars, and can change them from stars to any other Font Awesome icon. You are also able to set icons to go at the left and right of the stars; they default to frowning and smiling faces, but they can be replaced by any Font Awesome icon, or you can remove them altogether.

- From a style perspective, the font size controls the size of the Font Awesome icon, foreground color controls the color of the Font Awesome icon, and the background color controls the color behind the icons.
- The validation supported is required/not required.

Slider

The Slider element allows a user to move a selector along a numeric spectrum and pick a number within the range. You can set the minimum value (Min Value input field)

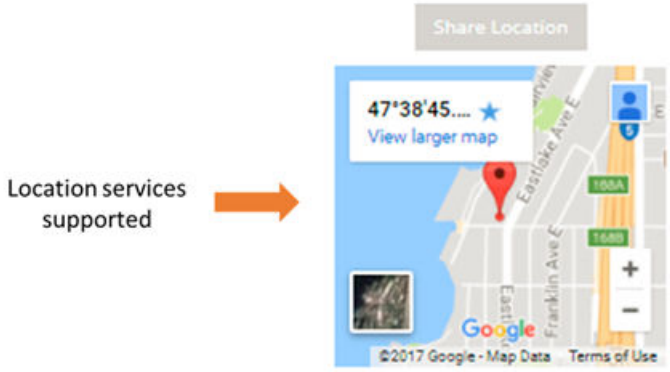
and the maximum value (Max Value input field); both must be integers and the Max Value must be equal or greater than the Min. You may also set the amount by which the slider will increment as a user slides it along (Step input). For example, if your Min Value is set to 0, your Max Value to 10, and your Step value to 2, then as the user moves the slider along, it would increase from 2 to 4 to 6, and continuing on.

- To change the right, left, top, and bottom margins, you can enter the value as a percentage or in number of pixels. The area behind the Slider can be adjusted by using the Background Color.
- There is no validation for a slider.

Location

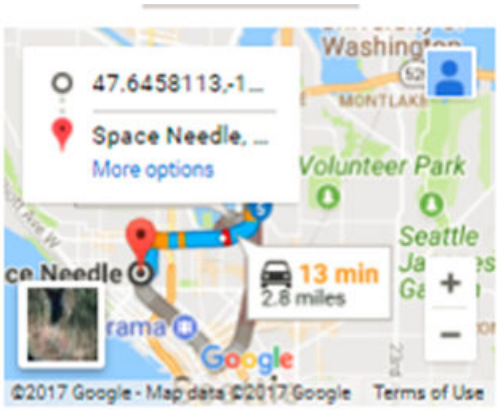
The Location element can be used to capture the location of a user. In order for the Location element to be set up, you need to have a Google Maps API Key. For it to work with a user, their device needs to support location services and be enabled.

- The Location element can be used to capture the location of a user. In order for the Location element to be set up, you need to have a Google Maps API Key. For it to work with a user, their device needs to support location services and be enabled.
- The Location element can be used to capture the location of a user. To set up the Location element, you need to have a Google Maps API Key. For it to work with a user, their device needs to support location services and must be enabled.
- **Show Map**: If enabled and the user's location is received, the user will see a map of their location.



- **Google Maps API Key**: If you are currently have a usable Google Maps API key, paste that key in the indicated input field. If you do not have a Google Maps API key and would like to use the location feature, you can go to the Google Maps site. Once you have the key, you can paste it in the input field.

- **Destination**: If you provide a destination in this field, then, after the location of the user has been discovered, the map will render with directions from the user's location to the destination. For example:



Directions to
Space Needle, Seattle, WA

- **Font Size**, **Foreground Color**, **Weight**, and **Style** control the size, color, weight, and style of the label text on the button; **Background Color** controls the color of the button itself. To change the right, left, top, and bottom margins, you can enter the value as a percentage , for example 10%, or in number of pixels, for example, 20px.
- The only supported validation for Location Elements is required.

Barcode Entry

The Barcode element allows a user to take a picture of a barcode. From that image, the barcode numbers are extracted and displayed. If a barcode is not detected in the uploaded image, there's a message that alerts the user and they can re-upload. Note that in Preview and Test Mode, a real image will not be uploaded; you will only see a placeholder image.

- **Instructions**: The Instructions are what displays within the barcode box, above the button. Note that the text on the button, "Select Image", and the text beneath the button can't be changed.
- **Allow Manual Barcode Entry** and **Manual Input Label**: If you select **Allow Manual Barcode Entry**, a user could enter in the barcode number if they decide not to upload a photo of the barcode or if there are challenges in character recognition from the photo. The **Manual Input Label** controls the text that is seen for the input if you do want the manual entry option to be available.
- **Clear Button Label**: After an image is uploaded, a "Clear" button appears. This is used to clear an exiting image if the image isn't readable or the user wants to provide a different image. The **Clear Button Label** sets the text on that button. Its color cannot be changed.
- **Show Preview Image**: When checked, this will show the uploaded image.
- **Error Message: Barcode Not Detected**: If an image is uploaded where no barcode can be detected, this message will display. This message can be customized.

- The **Foreground Color** controls the instructions text and *or Drag and Drop* text. The **Background Color** and **Border** control the background color and the border of the barcode box where the instruction text, Select Image button and or *Drag and Drop* text are.
- Barcode elements can have several forms of validation. They can be required, the barcode entry can be set to match a regular expression, and you can also set a JavaScript validation function to validate it.

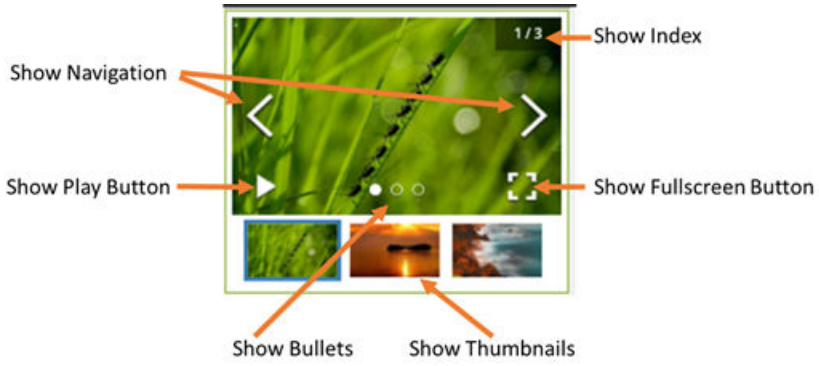
Images and Layout

Image

You can upload a static image to your instant app. Do this by dragging and dropping any image on your desktop, or by entering an image URL to upload. Styling is limited on images and is restricted to layout, where you can adjust dimensions and border color.

Image Gallery

An Image Gallery allows users to see a set of images in a carousel. You can add images by drag and drop or by looking up a file. You can control the way the carousel looks and interacts via a series of checkboxes, for example, Show Index, and Show Play Button; see image below for a complete set. You can also set the rate at which the images advance (in milliseconds) by setting the **Play Interval**. There are no style or validation options with the Image Gallery.



Divider

The Divider Element creates a line in the instant app. Through the Style Tab, you can control the height of the bar (in pixels only) as well as its color. The divider is a fixed length, which is the full width of the instant app. There are no validation options.

Content

Label / Text

The Label element allows you to add static text anywhere within your instant app. This element is often used to create headers for sections or to write lines of text between other elements. You can format this element using markdown.

HTML

The HTML element gives you a place to write your own HTML, using a limited subset of supported HTML:

-
-
-
- <u></u>
-
-
-
-
- Font Awesome icons can also be used. If you click out of the HTML box, you see your text appear in the preview.
- The options in the style tab will not override your HTML. For instance, suppose you write this HTML:

```
<center> Hello !</center>
<br>
<b> How are you? </b>
```

- On the style tab, if you select **weight light**, only “Hello!” is light. If you select **alignment right**, only “How are you?” is aligned right. If you select italic, both are italicized, but “How are you?” remains bold.
- There are no validations for the HTML element.

Social Buttons

The Social Buttons Element provides you the option to display Facebook, Twitter, LinkedIn, and Instagram icons in the instant app. When the user clicks on one of the icons, it takes them to the URL you provided on the Configure Tab. If you do not input a link for a particular social network, the icon does not show up. For instance, if you do not want to have an icon for Instagram show up on your instant app, simply leave the Instagram input field blank.

- On the Style tab, the background circle and the actual icon, for example the bird for Twitter, are controlled separately. The icon’s size and color are controlled by the font size and foreground color. The color of the circle around the icon is controlled by the background color. A circle is the default shape because the inner dimensions are set to equal numbers (50px, 50px). If those are not equal, it will result in an oval. Also, the border of the circle has rounded corners set to 50%. If that is changed, you create shapes other than a circle, depending on the percentage set and on the inner dimensions.
- There are no validations for social buttons.

YouTube

The YouTube element will show the user an image link to a YouTube video. When the user clicks on a video, it plays within the instant app; the user is not be taken out of the instant App.

- **Label**: The label displays text above the image link to the video
- **Video URL**: In the video URL field, you input the URL to the YouTube movie.
- **Show URL**: If you enable show URL, the video's URL displays below the image link.
- For the style, you can configure the top, bottom, right, and left margins of the image of the video using pixels or percentage.
- There are no validations for the YouTube element.

Embedded Website

The Embedded Website Element allows you to iFrame a website to your instant app with a fixed height and width. You might do this, for example, if you want to be able to process a payment for a user. You would input the URL in the website URL and if you want the URL to be visible (it renders above the iFrame), you select **Display URL**. For privacy or other reasons, uncheck **Show the Website to Sender** if the user is the only one who should be able to see the website. If the website you are embedding has data that you want to pass back to the instant app, you should select **Append Callback URL**. Following our example of processing a payment, if a payment succeeds, you may want the user to have one experience, and if it fails, you may want the user to have a different experience. In order to set element values and drive different instant app behaviors, select **Append Callback URL**. These instructions detail how callback URLs work:

- You can set the fixed height of the iFramed website by adjusting the inner dimension height in pixels, and the width by the left margin and right margin (in pixels or percentage).
- There are no validations for the embedded website element.

PDF Viewer

The PDF Viewer Element allows your user to read through a PDF within the instant app. You can provide the file via drag and drop or file upload, and also set the page number you want displayed first. The user can navigate to the document using the next and previous buttons. However, these buttons are not configurable.

- You can adjust the size of the PDF display, but if you input an inner dimension greater than 100%, you risk not displaying parts of the file, because there is no horizontal scroll bar. The default inner dimension is 90%.
- There is no validation for the PDF viewer.

Chart

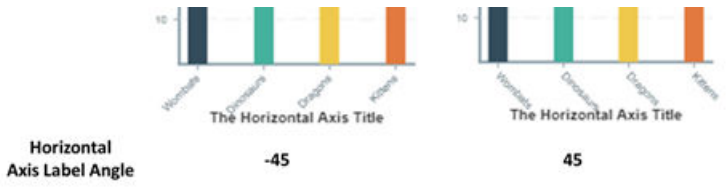
The Chart Element allows you to create four different chart types (bar, pie, line, and scatter chart) that render in your user's instant app. For each, you can manually input data in the chart element UI or dynamically pull in data from an external source using JavaScript snippets or parameters.

> **Note:**
>
> If you do use JavaScript or parameters, those override any static values you have entered.

- Single series only is permitted and each chart type has a specific data format and maximum number of recommended data points.
- When you first look at the chart element, each chart type is pre-populated with static values. This allows you to switch between the different chart types to see how a chart renders. If you want to use JavaScript or parameters to create the chart, that data does not render a chart in preview or test mode. To see how your chart will look with dynamic data, configure and style it with static data and then uncheck **Enter static values manually** . If you change the static values of a bar, pie, or line chart, the changed values are maintained if you switch between those chart types. Scatter charts have a different data input format; any changes to the static values of bar, pie, or line charts are not preserved if you change to a scatter chart.

Bar Chart

- Bar chart values must be entered as: Label,Value. Labels cannot contain a comma. Values are numbers that contain "+", "-", and ".", but no commas, spaces, or other special characters.
- Bar charts support a maximum of 40 data values, but render best with 20 or fewer. The colors of the bars can be set on the style tab using the five color palette options.
- Data labels: You can chose to whether to show or hide data labels. If you show them, the number value for each bar shows on the graph. Via the style tab, you can adjust the label placement of the data label up or down in pixels, and you can control the label color with the label fill color.
- The chart title, horizontal (x) axis title, and vertical (y) axis title are set on the configuration tab and their size, weight, and color are controlled on the style tab via size, weight, and chart and axis title color. Their style is the same; you cannot control them separately.
- The labels default to horizontal alignment along the horizontal (x) axis. To make them angled and permit more labels, use the style tab's horizontal axis label angle. The angle can be entered as a positive or negative value:



- By default, the vertical (y) axis minimum and maximum will be inferred from the data. if you want to set them yourself, you can uncheck **Infer from Data** and enter your own minimum and maximum values. These values can be positive or negative.
- You can toggle to show or hide the legend using the **Display Chart Legend** checkbox.

Line Chart

- Line chart values must be entered as: Label, Value. Labels cannot contain commas. Values are numbers that can contain "+", "-", and ".", but no commas, spaces, or other special characters are accepted.

- Line charts support a maximum of 40 data values, but render best with 20 or fewer. The color of the line can be set on the style tab using the data color option.
- Data labels: You can chose to show or hide data labels on your chart. If you show them, you see the number value for each point on the graph. Via the style tab, you can adjust the label placement of the data label in pixels, and you can control the label color with **Label Fill Color**.
- The chart title, horizontal (x) axis title, and vertical (y) axis title can be set on the configuration tab and their size, weight, and color are controlled on the style tab via size, weight, and the chart and axis title color options. The title style is the same; you cannot control title styles separately.
- the labels default to a horizontal display along the horizontal (x) axis. To make them angled and permit more labels, use the style tab's horizontal axis label angle. As in the bar chart, the angle can be entered as a positive or negative value.
- By default, the vertical (y) axis minimum and maximum will be inferred from the data provided. If you want to set them yourself, uncheck **Infer from Data** and enter your own minimum and maximum values. These can be positive or negative values.

Scatter Chart

- Scatter chart values must be entered as: xValue,yValue. Values are numbers that can contain "+", "-", and ".", but no commas, spaces, or other special characters.
- Scatter charts support a maximum of 200 data value pairs, but render best with 100 or fewer. The color and size of the points can be set on the style tab using the data color and scatter size options.
- Data labels: You can chose to show or hide data labels on your chart. If you show them, the Y-value for each point shows on the graph. Via the style tab, you can adjust the label placement of the data label up or down, and you can control the label color with the label fill color.
- The chart title, horizontal (x) axis title, and vertical (y) axis title are set on the configuration tab and their size, weight, and color are controlled on the style tab via size, weight, and chart and axis title color. Their style is the same; you cannot control them separately.
- The labels themselves default to horizontal placement along the horizontal (x) axis. To make them angled and permit more labels, use the style tab's horizontal axis label angle. As in the bar chart, the angle can be a positive or negative value.
- By default, the vertical (y) axis min and max and the horizontal (x) axis min and max are inferred from the data provided. If you want to set them yourself, you can uncheck **Infer from Data** and enter your own minimum and maximum values. These can be positive or negative values.

Pie Chart

- Pie chart values must be entered as: Label,Value. Labels cannot contain a comma. Values are numbers that can contain "+", "-", and ".", but no commas, spaces, or other special characters are allowed.
- Pie charts support a maximum of 20 data values, but render best with 10 or fewer. The colors of the chart components can be set on the Style Tab using the five Color Palette options. The Style Tab also allows you to create a donut chart by adjusting the inner radius and add padding between the slices by using **Pad Angle** . By default, the pie chart will be 360 degrees. However, you can adjust

these angles using the start angle and end angle by entering the desired degrees (positive or negative).

- Data labels: You can chose to show or hide data labels on the pie chart slice. If you show them, you can select any combination of the value itself (the value part of the Label, Value), the percent that the value represents of the whole, and/or the slice name (the value part of the Label, Value). On the configuration tab, you can adjust the data label placement, moving it in or out from the center of the pie. Via the style tab, you can control the label color with the label fill color.
- The chart title is set on the configuration tab and its size, weight, and color are controlled on the style tab via size, weight, and chart and axis title color.
- You can toggle the show or hide the legend using the **Display chart legend** checkbox.

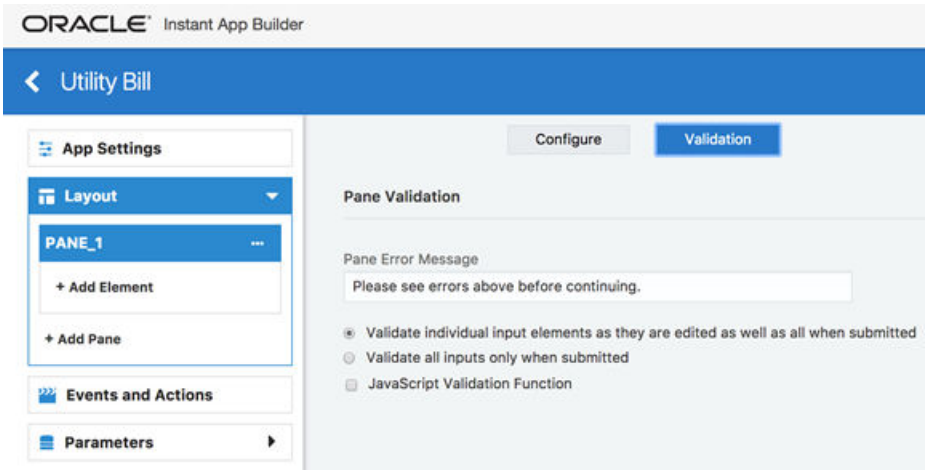
Pane Validation

Each pane in an instant app has two main validation options:

1. **Validate individual input Elements as they are edited as well as all when submitted**—This means that if a customer enters data that's not valid (for example, a phone number that does not conform to the prescribed regex), then they would immediately see an error message. The instant app will also check that everything is valid before moving on to the next pane.
2. **Validate all inputs only when submitted**—When you set this condition, a customer doesn't see an error message immediately after entering data that's not valid. In this case, the customer could continue to fill out the other fields and wouldn't be confronted with the error message until the problematic data gets submitted.

Add the messages that display under either of these conditions in the **Pane Error Message** field. This message appears directly above the element (typically, a button) that triggered the submission of the invalid data. Using the Validation tab for an individual element, you can create an additional error message that displays next to the error-causing element(s).

You can also validate a pane by writing your own JavaScript. The message included in the code is triggered when the pane is submitted. See The Validator Object.



The Validator Object

Use the Validator Object to set and clear element errors for complex validation scenarios. For example:

```
if (element.value == "Phil") {
    validator.displayElementError("input", "Can't be Phil");
} else {
   validator.clearElementError("input");
```

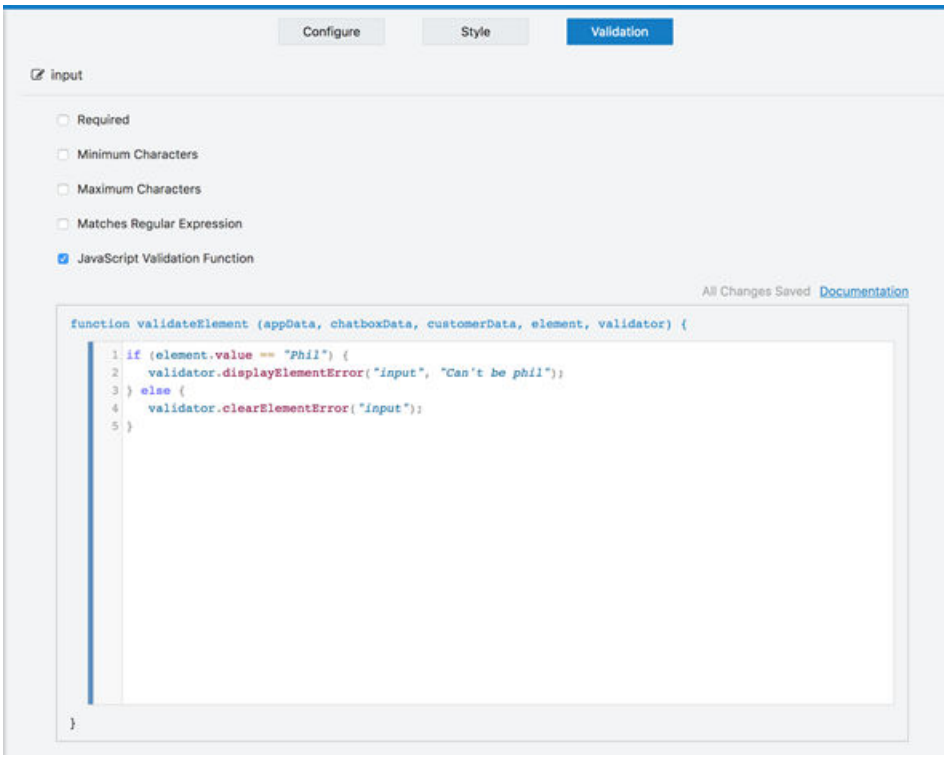
As shown in this snippet, the object surfaces two functions:

- `validator.displayElementError(<ElementId>,<errorMsg>`

> **Note:**
>
> If the snippet does not call `displayElementError` during execution, then that element is considered valid.

- `validator.clearElementError(<ElementId>);`

You can define Validator object from the Validator tab when you select the **Execute JavaScript Validation** option for an input element.

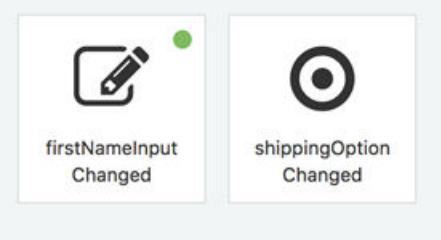


Events and Actions

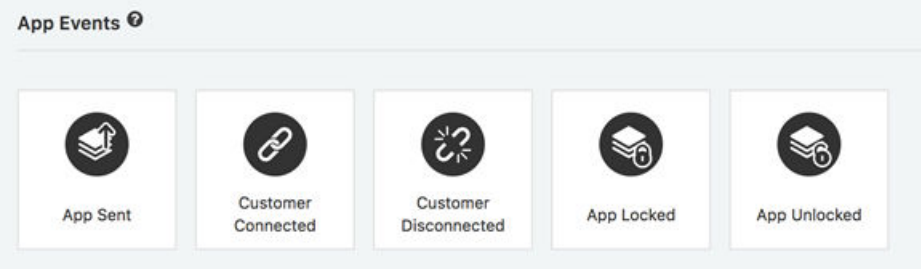
In the Events and Actions section for an instant app, you specify actions that will occur when the events fire. For instance, when the customer clicks a radio button, the instant app fires the event associated with the radio button's changed event. Or, when the

instant app is sent to the customer, the App Sent event is fired. This event might be used to set up the initial element values, hit an external web API to collect data, or make certain elements invisible or disabled. In summary, the Events and Actions section is where you make your instant apps dynamic.

Events with associated actions are displayed with a green dot in the upper right corner. Events without any associated actions will not have a dot. For example, in the following illustration, the input element *firstNameInput* has configured actions, while the *shippingOption* radio button element does not.

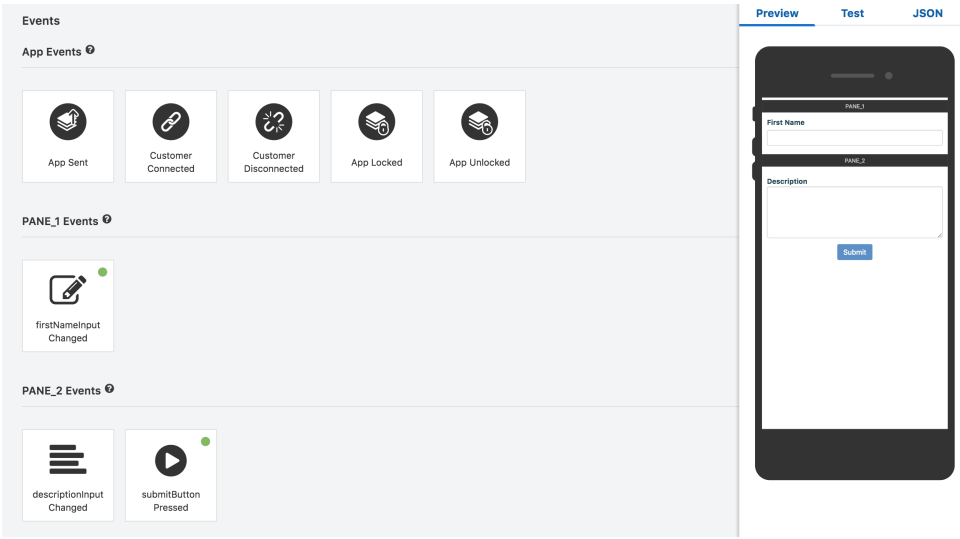


App Events

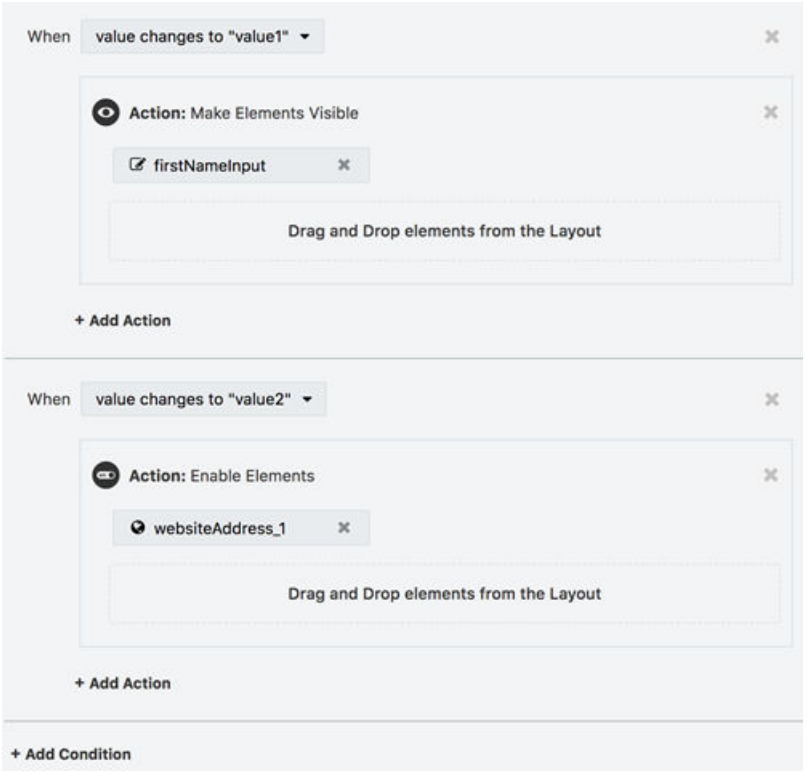


- **App Sent Event**—This event and any associated actions are fired the very first time the instant app is sent to the recipient (a customer). Typically, you use this event for disabling or hiding elements, calling an external web API to retrieve data that is used in the instant app, and for instantiating input elements with their initial values.
- **Customer Connected Event**—This event and any associated actions are fired every time the customer opens the instant app. Most often, you can use this event to refresh data from an external web source, ensure consistency and validity of the various values, and to reset the active pane.
- **Customer Disconnected Event**—This event and any associated actions are fired when the recipient disconnects from the app. In general, you would use this event is to pipe partially completed data to an external web source or to the bot.
- **App Locked and Unlocked Events**—This event and any associated actions are fired when the instant app is locked or unlocked. The instant app can be locked or unlocked using actions, a JavaScript snippet, or by a manual action by the sender.
- **Input Value Changed and Button Pressed Events**—When instant Apps have input or button elements, an event is automatically created for that element. An element's input events are arranged by pane. For example, the following illustration shows the events that were created for two separate panes. On the first pane (PANE_1 Events), there is an event for the single line input element named *firstNameInput*. This element has a green dot, which indicates that actions have been configured for it. On the second pane (PANE_2 Events), there are two

events: one for the *descriptionInput* , a multi-line Input element and another for the *submitButton* Button Press event.



The Choice elements (Radio Buttons, Checkbox, Pick List, Select Menu, Button List) support Conditional Action Lists. For these events, you can build action lists that execute when a specific condition is met. For example, the following illustration shows a radio button with Conditional Action Lists. The conditions are tested and then executed sequentially. One of the options for the Condition is **value changes**. By selecting this option, you enable the actions to fire whenever the value changes for the element.



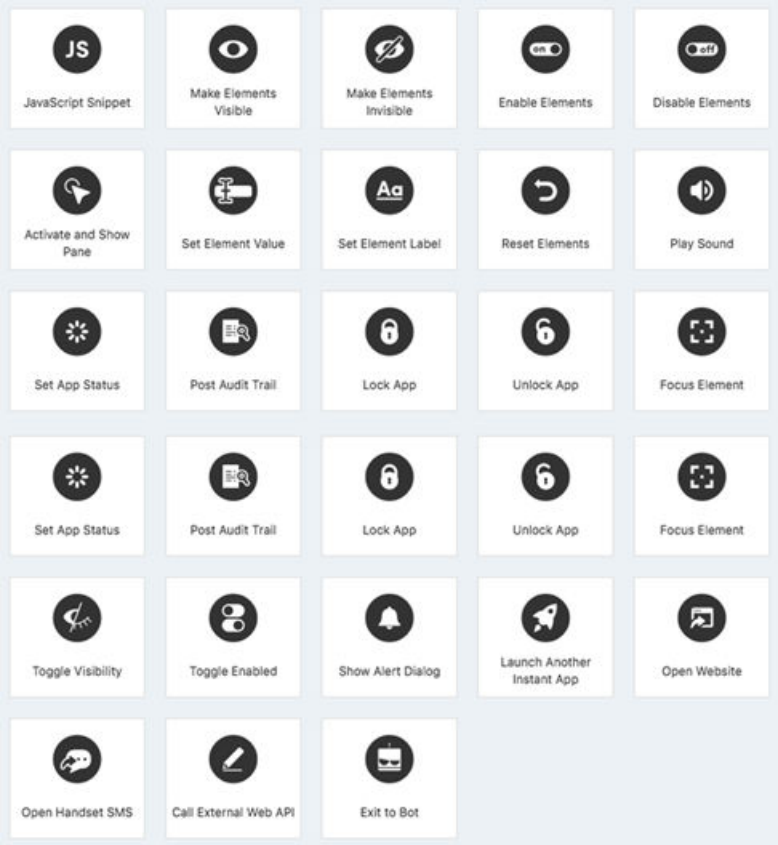
You can delete a condition by clicking the **X** icon in the upper right corner associated with the condition. Deleting a condition will delete any and all associated actions.

Actions

Actions can be configured for each event. These actions can modify the state of the app, for example, by setting an element value, showing or hiding elements or activating a pane. You can also use them to call external web APIs or to execute JavaScript snippets.

For actions that require an element to operate on (for instance, the action), you specify the element by dragging and dropping them from the Layout section. See Adding, Moving, and Deleting Elements.

You can create actions using the Instant App Builder, or specify them programmatically and execute them as a JavaScript Snippet.



Event	Action	JavaScript Function Signature
Computation	JavaScript Snippet	N/A
Element Operations	Set Element Value	`app.setElementValue(ElementId, value)`
N/A	Make Elements Visible	`app.makeElementsVisible(Elements)`

Event	Action	JavaScript Function Signature
N/A	Make Elements Invisible	`app.makeElementsInvisible(Elements)`
N/A	Enable Elements	`app.enableElements(Elements)`
N/A	Disable Elements	`app.disableElements(Elements)`
N/A	Toggle Visibility	`app.toggleVisibility(Elements)`
N/A	Toggle Enabled	`app.toggleEnabled(Elements)`
N/A	Set Element Label	`app.setElementLabel(ElementId, textOrHTML)`
N/A	Set Element Properties	`app.setElementProperties(elementId, properties)`
External Data	Call External Web API	`chatbox.callExternalWebAPI(dataKey, eventName, method, URL)` and other variants.
Pane Operations	Activate and Show Pane	`app.activatePane(PaneName)`
N/A	Reset Elements	`app.resetElements(ElementName)` and `app.resetElements(PaneName)`
Interaction	Play Sound	`app.playSound(soundNameOrURL, volume)`
N/A	Show Alert Dialog	`app.showAlertDialog(dialogTextOrHTML)`
N/A	Focus Element	`app.focusElement(ElementName)`
N/A	Open Website	`chatbox.openWebsite(url, dialogMessage)`
N/A	Open Handset SMS	`chatbox.openHandsetSMS(message, phoneNumber, dialogMessage)`
Audit/Console	Set App Status	`app.setAppStatus(statusString)`
N/A	Post Audit Trail	`app.postAuditTrail(string);`
App Lifecycle	Lock App	`app.lock()`
N/A	Unlock App	`app.unlock()`
N/A	Launch Another Instant App	`chatbox.launchInstantApp(schemaId, [params])`
N/A	Exit to Bot	`app.exitToBot({key: value, key: value})`

JavaScript Snippet

This action, which executes JavaScript on the server, broadens your options for configuration and error checking. You can also use it to create dynamic customer experiences.

Each JavaScript Snippet is instantiated with a function signature.

- Button Press Events (for Button Elements)—`function buttonPressed(app, chatbox, customer, element) {}`
- Value Changed Events (for Input Elements)—`function valueChanged(app,chatbox, customer, element, oldValue, newValue) {}`
- App Sent Event—`function appSent(app, chatbox, customer) {}`
- Customer Connected Event—`function customerConneted(app, chatbox, customer, count) {}`
- Customer Disconnected Event—`function customerDisconnected(app, chatbox, customer) {}`
- App Locked Event—`function appLocked(app,chatbox,customer) {}`
- App Unlocked Event—`function appUnlocked(app.chatbox.customer) {}`
- Input Validation Event (on an Input Element's Validation tab)—`function validateElement(appData, chatboxData, customerData, Element, validator) {}`

When the function is called on the server, function parameters are instantiated with objects that can be accessed and used along with functions that perform actions that are the equivalent to the actions that you configure with the Instant App Builder.

Function Parameters	Description
`app.appId`	The unique identifier for this instance
`app.schemaId`	The schema number for this Instant App
`app.createdTimestamp`	The number of milliseconds since 1/1 1970

Function Parameters	Description
`app.status`	A string denoting the app's current state as set by `app.setAppStatus(str)`.
`app.locked`	A boolean indicating if app is locked/unlocked
`app.activePane`	A string indicating name of the current Pane
`app.tags`	
`app.parameters`	A key-value list of parameters passed to the Instant App at launch

`App.Elements`
`[`
`{`
`id`
`type`
`visible`
`enabled`
`label`
`properties`
`value`
`}`
`]`
	A list of elements and all their associated information.
	• `id`—String indicating Element ID or name
	• `type`—Element Type (`button`, `chart`, etc.)
	• `visible`—Boolean indicating current visibility
	• `enabled`—Boolean indicating current enablement
	• `label`—String indicating current label
	• `properties`—Element properties (type varies by Element)
	• `value`—Element value (type varies by Element)

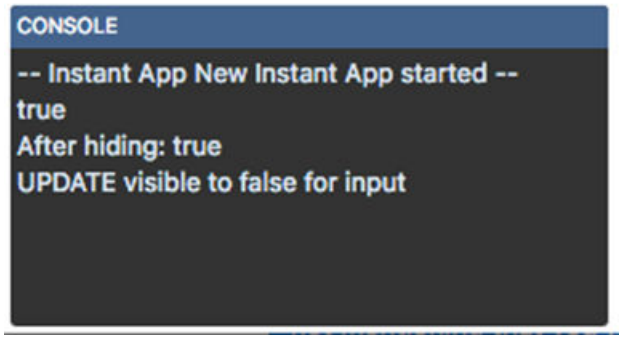
`customer.customerId`	A unique customer ID for current customer
`customer.name`	The customer name, if known

JavaScript Snippet Execution

- Limited Run time and Length—JavaScript Snippets are limited to a total server run time of 1s. If your snippet exceeds that time, an error will be posted and the snippet will fail to execute. Snippets are limited to 10000 characters in total length.
- External Resources—The only way to access external resources is to use the `app.callExternalWebAPI()` function, which executes asychronously and returns data to a named callback event for further processing.
- Asynchronous, Server-side Execution—The code that is executed on the server-side. No client side injection is possible and each snippet is sandboxed to avoid data integrity issues. Snippets execute asynchronously; blocking operations are not allowed or supported.
- Actions Performed Post-Execution—Values that are changed through a function call are not immediately reflected in the objects that were passed to the function. For instance, calling `app.hideElements("input")` will not affect the value of `app.elements.input.visible` within the scope of the function. For example:

  ```
  print(app.elements.input.visible);
  app.hideElements("input");
  print("After hiding: " + app.elements.input.visible);
  ```

  The following illustration shows how the Console reflects this action, with the visibility remaining unchanged. See Test Mode.



Commonly Used JavaScript Snippets

Here are some common JavaScript snippets and guidelines.

How to Change the Value of an Element

```
function valueChanged(app, chatbox, customer, element, oldValue, newValue) {

  if (newValue) {
    var capitalized = newValue[0].toUpperCase() + newValue.slice(1);
    app.setElementValue(element.id, capitalized);
  }
}
```

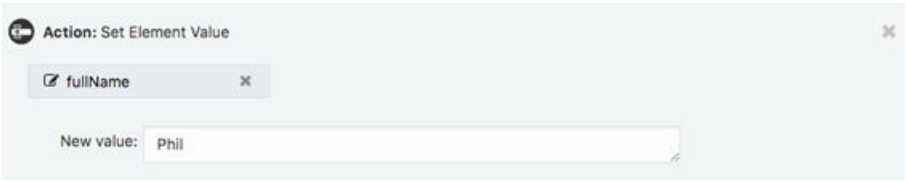
How to Change the Behavior of the Instant App Based on a Selection of a Radio Button or Select Element

```
function valueChanged(app, chatbox, customer, element, oldValue, newValue) {
  switch(newValue) {
     case "shippingIssue":
        app.showElements("description");
        app.hideElements("starRating");
        break;
     case "return":
        app.showElements("starRating");
        app.hideElements("description");
        break;
     case "changeAddress":
        app.activatePane("changeAddressPane");
        break;
  }
}
```

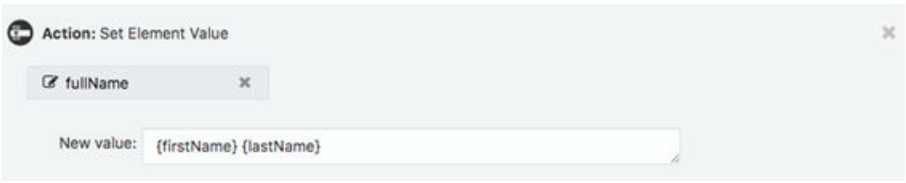
Set Element Value

This action sets the value of an input element. To configure this action, drag and drop the element and then specify a value. You can define this as a static value, but you can access other element values or parameters using brace notation. For example:

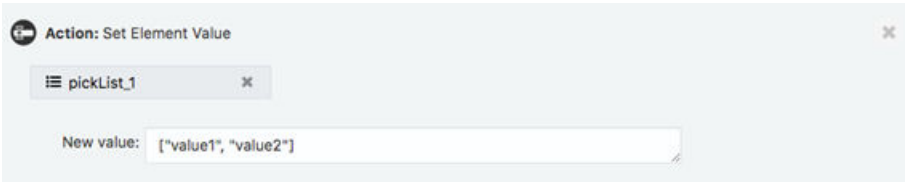
- The following illustration shows setting the fullName element to a static value.



- In this illustration, the fullName element is set to the value of the `{firstName}` and `{lastName}` values.



- This illustration shows setting value of Picklist element to check options corresponding to `value1` and `value2`.



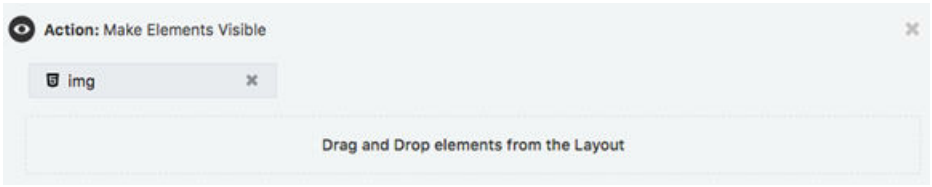
JavaScript Action

```
app.setElementValue(“ElementId”, value);
```

> **Important:**
>
> You’ll get an error if you use `setElementValue` to change the label or other properties of an element.

Make Elements Visible

This action displays one or more hidden elements.

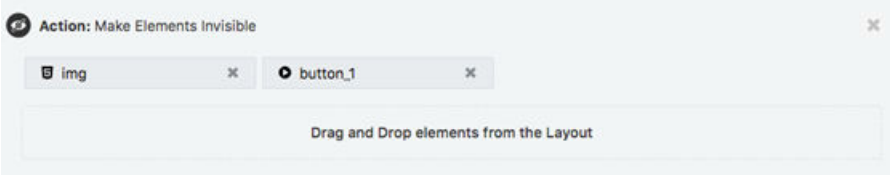


JavaScript Action

```
app.showElements("ElementName"); // single element
app.showElements("ElementName", "ElementName2"); // comma-separated Elements
app.showElements(["ElementName", "ElementName2"]); // list of Elements
```

Make Elements Invisible

This action hides one or more hidden elements.

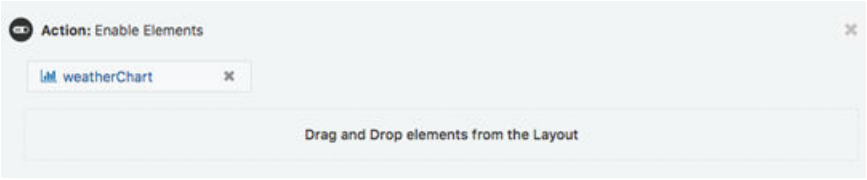


JavaScript Action

```
app.hideElements("ElementName"); // single element
app.hideElements("ElementName", "ElementName2"); // comma-separated elements
app.hideElements(["ElementName", "ElementName2"]); // list of elements
```

Enable Elements

This action enables the input of the elements that you drag and drop from the Layout category. The elements enabled by this action still respect the **Element Usability** setting, meaning that the element remains disabled when you've enabled an element for a target that can't use it. See Common Configuration.

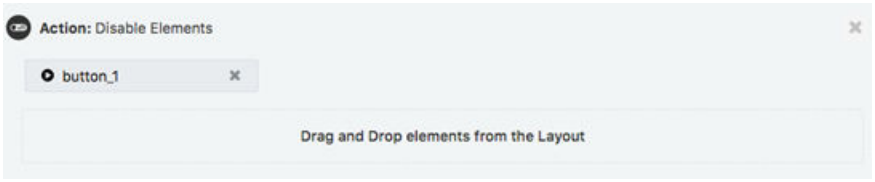


JavaScript Action

```
app.enableElements("ElementName"); // single element
app.enableElements("ElementName", "ElementName2"); // comma-separated elements
app.enableElements(["ElementName", "ElementName2"]); // list of elements
```

Disable Elements

Use this action to disable input for the elements that you drag and drop from the Layout category. When you disable an element, it's grayed out and in this state, can't accept a customer's input or manual updates.

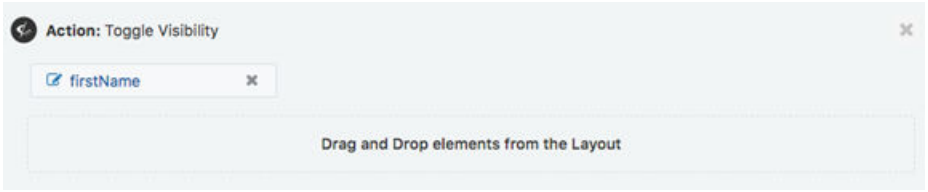


JavaScript Action

```
app.disableElements("ElementName"); // single element
app.disableElements("ElementName", "ElementName2"); // comma-separated elements
app.disableElements(["ElementName", "ElementName2"]); // list of elements
```

Toggle Visibility

This action toggles the visibility for the elements that you drag and drop from the Layout category. If an element is invisible, it will become visible (and vice versa).

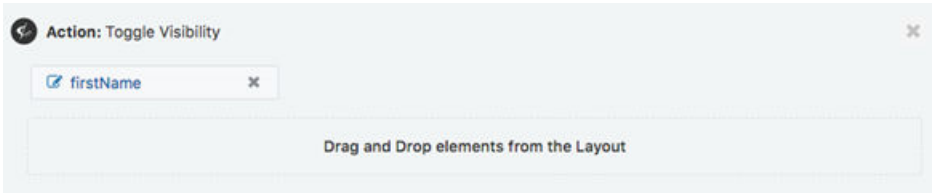


JavaScript Action

```
app.toggleVisibility("ElementName"); // single element
app.toggleVisibility("ElementName", "ElementName2"); // comma-separated elements
app.toggleVisibility(["ElementName", "ElementName2"]); // list of elements
```

Toggle Enabled

This action toggles the input state for the elements that you drag and drop from the Layout category. If an element is enabled, it will become disabled. Likewise, if an element is disabled, it will become enabled.



JavaScript Action

```
app.toggleEnabled("ElementName"); // single element
app.toggleEnabled("ElementName", "ElementName2"); // comma-separated elements
app.toggleEnabled(["ElementName", "ElementName2"]); // list of elements
```

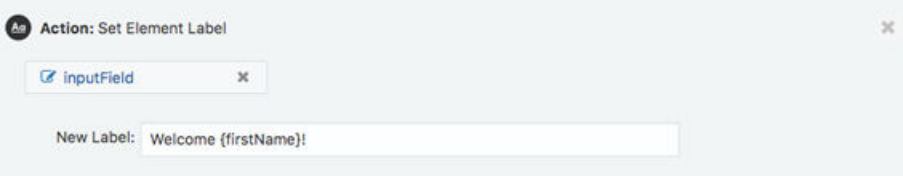
Set Element Label

Use this action to set a label value for an element. To configure this action, drag and drop the element from the Layout category and then specify the value. You can add a

static value, a brace notation (if you need the label to access values from other elements and parameters) or a combination of both. You can format the label using a subset for HTML.

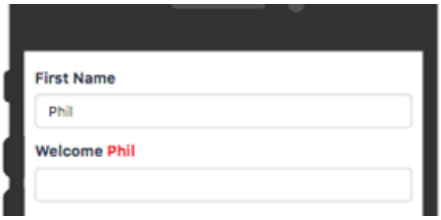
- As shown by the InputField Element in this image, you can set the set the label value as a static value (`My new label`).

- This image shows using static values and brace notation for stored values (`Welcome {firstName}`).



- This image shows a combination of a static value, HTML tags and value replacement through brace notation (`Welcome <span style='color: red'>{firstName}</span>`).

The following image shows how this combination renders the label at runtime.



JavaScript Action

```
app.setElementLabel("ElementId", "new Label (supporting {} and HTML)");
```

> **Note:**
>
> Using `setElementLabel` on a Checkbox Element also changes the text that's next to the checkbox.

Set Element Properties

This action is currently only used to set the properties for the Chart Element. The way chart data is read is `app.chartElement.properties.data`. This is set as:

```
app.setElementProperties("chartElement", { "data": d });
```

`app.element.properties` is an object that stores properties that are settable with the `app.setElementProperties()` function. It supports only the `data` property. For example:

```
var chartData = app.elements.chartElement.properties.data
```

`app.setElementProperties(elementId, properties)` sets a collection of properties on the given element. The `properties` parameter is a key-value map of the properties to update and their updated value. It supports only the data property. For example:

```
app.setElementProperties("chartElement", { data: chartData })
```

Call External Web API

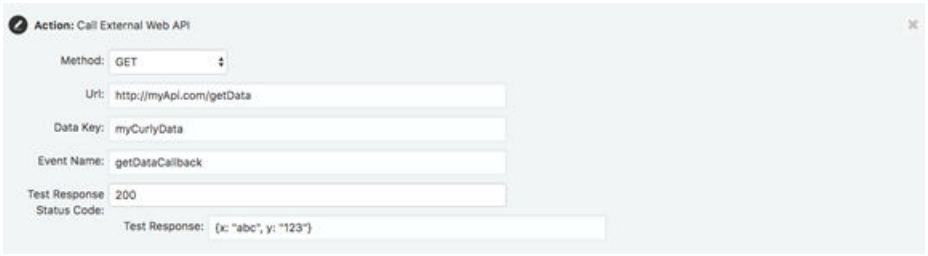
Use this action to call an external API endpoint. To configure this action, specify the API method (GET, DELETE, PATCH, POST, or PUT) and the URL endpoint.

Data from the external web API will be sent to the event that you specify in the **Event Name** field. To process the returned values, you'll need to create a Named Callback Event whose name matches the value that you enter in this field.

> **Note:**
>
> All external web API calls are asynchronous in that they don't block and wait. Instead, they execute any subsequent actions immediately. When the data is returned from the API call, the specified Named Callback Event will then fire and execute.

If you provide a value in the **Data Key** field, use brace notation to make the API return data addressable. For example, if your Data Key is called *myCurlyData*, you can add `{myCurlyData.x}` to the `Elements` in the JavaScript that support the curly notation format.

To test various scenarios without actually having to hit the specified endpoint, you can also specify test response and the status code. In Test Mode, when the action is executed, this test response and status code are sent to the named callback event. See Test Mode.



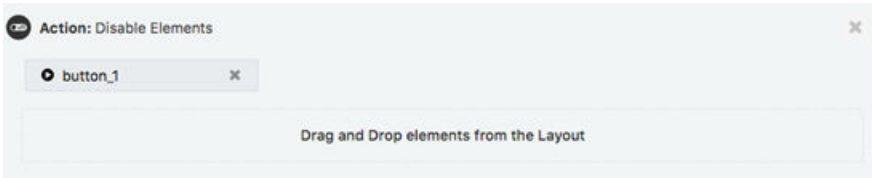
JavaScript Action

There are six variants of the `chatbox.callExternalWebAPI` Action:

```
chatbox.callExternalWebAPI(dataKey, eventName, method, url);
chatbox.callExternalWebAPI(dataKey, eventName, method, url, value);
chatbox.callExternalWebAPI(dataKey, eventName, method, url, value, contentType);


// value and contentType are optional (needed where method equals POST or PUT)
// contentType defaults to application/json if not specified.


chatbox.callExternalWebAPI(dataKey, eventName, method, url, testModeStatusCode,
testModeResponse);
chatbox.callExternalWebAPI(dataKey, eventName, method, url, value,
testModeStatusCode, testModeResponse);
chatbox.callExternalWebAPI(dataKey, eventName, method, url, value, contentType,
testModeStatusCode, testModeResponse);
```

Activate and Show Pane

This action switches the pane that's currently active. Using this action, you can switch context and also add wizard-like behavior to your app.



To configure this action, drag and drop a pane from the Layout category.

JavaScript Action

```
app.activatePane("PaneName");
```

Reset Elements

Use this action to reset elements (along with their visibility, label, and value attributes) to their original state. For example, you can configure this action for a form reset button by dragging and dropping the elements that the button resets.

JavaScript Action

```
app.resetElements("ElementName"); // single element
app.resetElements("PaneName"); // all elements on the specifiedPane
app.resetElements(["ElementName", "ElementName2"]); // list of elements
```

Play Sound

This action enables your instant app to play a specified sound at a specified volume.

You can configure this action by selecting one of the built-in sounds or, by choosing the **Specify URL** option, you can use an external sound file that's accessed via `https`. For the latter, you need to enter the URL where the file is hosted (`https://example.com/sound.mp3`, for example).

> **Note:**
>
> There are a few things to keep in mind if you opt for an external sound file:
>
> - The instant app can only access the file through `https`.
> - The instant app can't play a sound if you don't specify a sound file, or if the file is unavailable or can't be reached.
> - The file types vary by browser. Typically, browsers support mp3, mpeg, opus, ogg, oga, wav, aac, caf, m4a, mp4, weba, webm, dolby, and flac.
> - For the broadest browser coverage and compatibility, use an mp3–formatted file.

You can test any sound (built-in and external file) by clicking **Preview** see Preview Mode.

Built-in Sounds

- AscendingTone
- BellTwinkle
- DoubleBlip

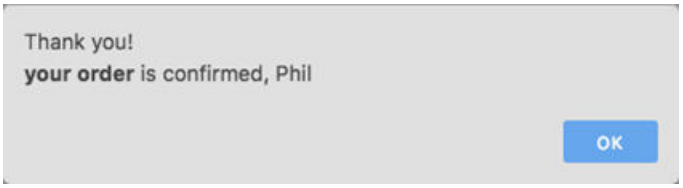
- HappyChime
- OptionChange
- PercussionBlipsNotic
- PercussionBlipsPop
- SlipperyStuttering
- SuccessAle
- SuccessChime

JavaScript Action

```
// supports all the tones from the drop down menu
app.playSound("AscendingTone", 100); // play at full volume

// play a sound from an external website at half volume
app.playSound("https://example.com/sounds/yahoo.mp3", 50);
```

Show Alert Dialog

This action enables your instant app to display a modal dialog message to its customers.



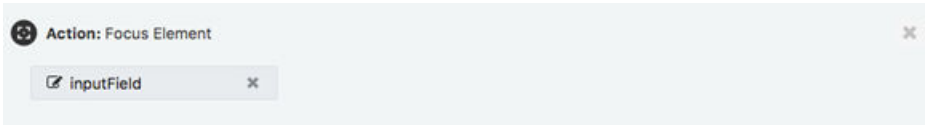
Your message definition can include both a limited set of HTML tags and brace notation. For example, `Thank you!<br/><b>your order<b> is confirmed, {firstName}`.

JavaScript Action

```
app.showAlertDialog("Thank you!");
app.showAlertDialog("Thank you {firstName}"); // use {} value replacement
app.showAlertDialog("<b>Thank you!</b>"); //use of limited HTML
```

Focus Element

Use this action to move the keyboard focus to a specified element.

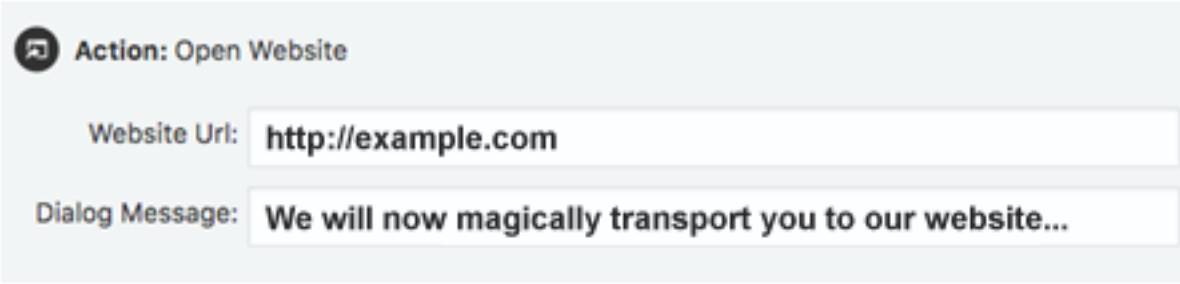


JavaScript Action

```
app.focusElement(“ElementName”); // unlock the app and allow input changes
```

Open Website

This action enables your app to launch a confirmation dialog, one that opens a URL in a new window after the customer clicks the confirmation button.



To configure this message, enter a message and a website URL using one of the following schemes:

Scheme	Example
`http`	`http://example.com`
`https`	`https://example.com`
`sms`	`sms:+12065551212`
`mailto`	`mailto:support@acme.co`
`tel`	`tel:+12065551212`
`ftp`	`ftp://myftp.com`

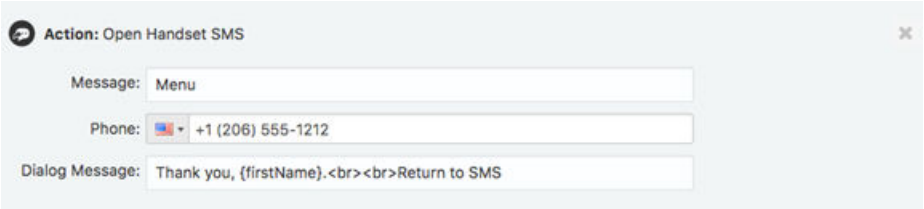
> **Note:**
>
> Some URLs, like `sms` and `tel`, are not supported by desktop browsers while others (`ftp`) are not supported by mobile browsers.

JavaScript Action

```
chatbox.openWebsite("http://example.com", "Let us take you to example.com");
```

Open Handset SMS

Use this action to display a message dialog, one that opens a phone number when the mobile device user taps a confirmation button with a pre-populated message. You can use brace notation in both the **Message** and **Dialog Message** fields to access element values (`Thank you, {firstName}.<br><br>Return to SMS`).



> **Note:**
>
> Some URLs, like `sms` and `tel`, are not supported by desktop browsers while others (`ftp`) are not supported by mobile browsers.

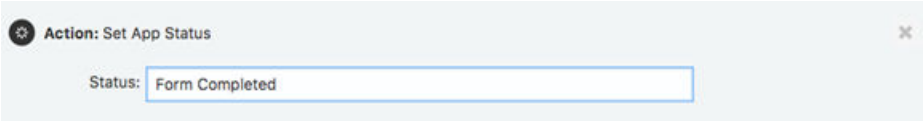
> **Important:**
>
> Some older mobile phones may not fully support the pre-instantiation of the message

JavaScript Action

```
chatbox.openHandsetSMS(“MENU”, “12065551212”, “Return to SMS?”);
```

Set App Status

Use this action to post the app status to the customer record’s audit trail. Each instant app uses a string to indicate its current state. For instance, suppose you have an instant app that is a form. For this, you can add two Set App Status actions, one with a *Form Sent* string in the **Status** field and the other with a *Form Completed* string in its **Status** field. Users can’t see these status strings, but they get posted to the audit trail and can be searched for analytics.



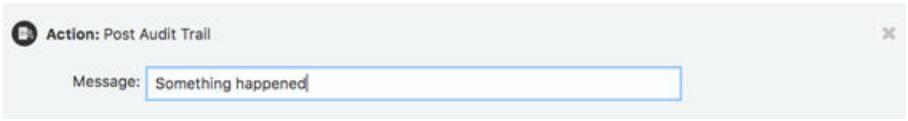
> **Tip:**
>
> Your string definitions can include brace notation (`{ElementOrParameterName}`).

JavaScript Action

```
app.setAppStatus("Form Submitted");
```

Post Audit Trail

Use this action to post a timestamped line of text to the customer record.



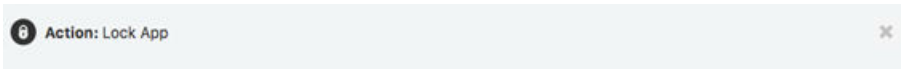
Your string that specifies the audit trail message can access element or parameter values using the brace notation (`{ElementOrParameterName}`). For example, `First Name Updated {firstName}`.

JavaScript Action

```
app.postAuditTrail("Something happened");
app.postAuditTrail("First Name Updated: {firstName}");
```

Lock App

Use this action to lock the instant app and prevent users from making any further changes. When this action is fired, the instant app displays an indicator that the instant app has been locked. At this point, your users can no longer click buttons or change any input field. After this action is called, the App Locked Event will fire if any actions are configured for that event.



JavaScript Action

```
app.lock(); // lock the app from further changes
```

Unlock App

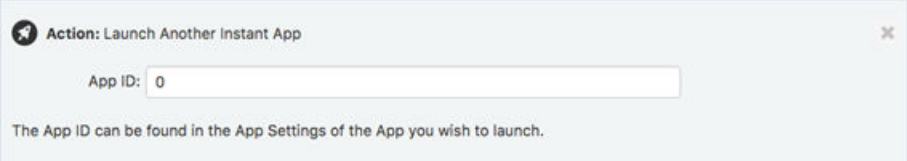
Use this action to unlock the instant app and allow users to resume with the instant app. In other words, your users can continue to click buttons and change and input values. After this action is called, the App Unlocked Event will fire if any actions are configured for that event.

JavaScript Actions

```
app.unlock(); // unlock the app and allow input changes
```

Launch Another Instant App

Use this action to immediately launch another instant app. To configure this action, specify the ID of target instant app.



> **Tip:**
>
> Use the value in the App ID field in the App Settings page.

JavaScript Action

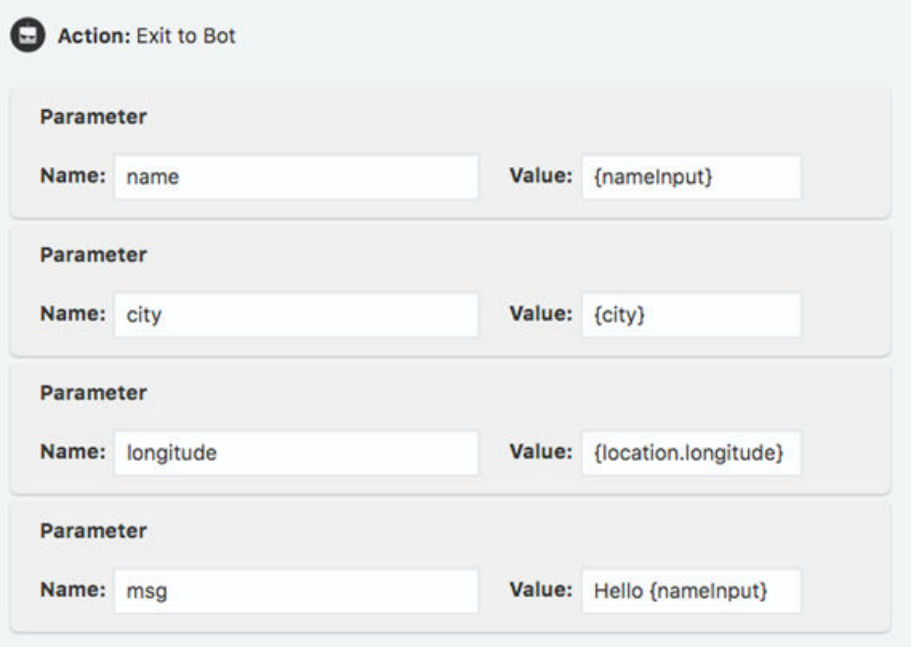
```
// launch an instant app with no parameters specified
chatbox.launchInstantApp("schemaId",  null);

// launch with specified parameters comma-separated
chatbox.launchInstantApp(
  "schemaId",
  { name: "paramName1", value: "paramValue1" },
  { name: "paramName2", value: "paramValue2" }
);


// launch with specified parameters in an array
var array = [];
array.push({ name: "paramName1", value: "paramValue1" });
array.push({ name: "paramName2", value: "paramValue2" })

chatbox.launchInstantApp("schemaId",  array);
```

> **Important:**
>
> You can't launch another instant app with parameters.

Exit to Bot

Use this action to enable the instant app to return values to the bot. The instant app remains visually active for the customer even after it returns its values to the bot unless you set actions that enable it to behave otherwise.

> **Note:**
>
> Returning values to the bot does not alter the visual state or usability of the bot in any way.



While you can define these values as static values, you're more likely to specify them using curly bracket notation like `Hello {nameInput}` or `{location.longitude}` so that they can return values from other elements. For example, the payload returned to the bot would look like this because of the value substitution:

```
{
"name": "Phil Gordon",
"city": "Seattle",
"longitude": "-32.33221156",
"msg": "Hello Phil Gordon"
}
```

JavaScript Action

```
app.exitToBot(); // no parameters
app.exitToBot({"name":"value", "name":"value"}); // exit and send data
```

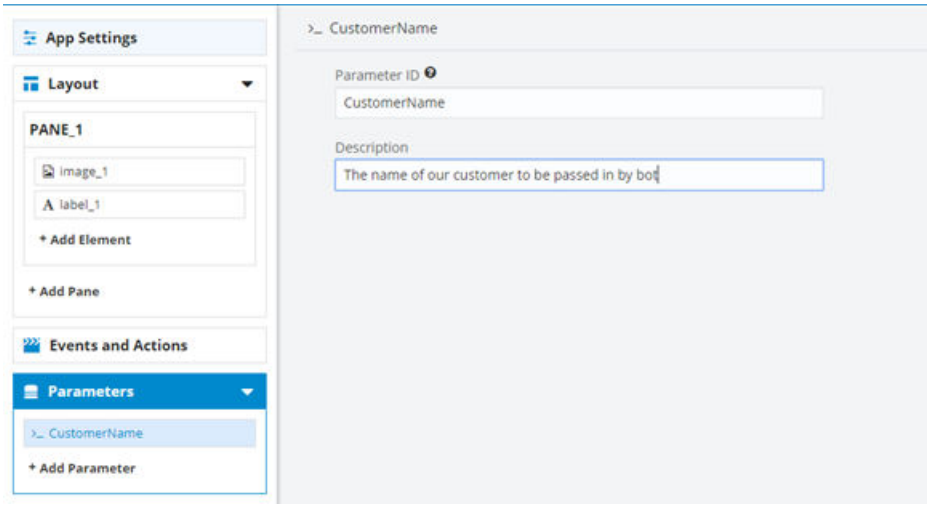
Parameters

Parameters are used to pass data from a bot into an instant app. This data can then be used both in elements and as part of the JavaScript snippets.

You can define a parameter in the Parameters section along the left, using the **+Add Parameter** function. When you create a parameter, you need to give it an ID, which is used as follows:

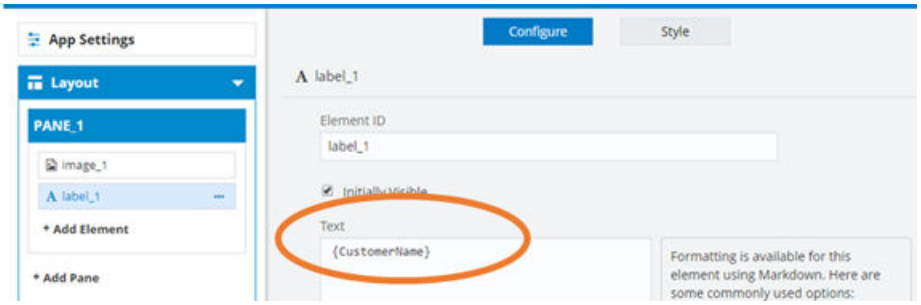
- When passing in a parameter to the instant app. This happens when an instant app is launched from a bot, or when you enter the parameters manually as part of testing. See Test Mode.
- Inside an instant app with the `{parameterID}` notation.
- Inside an instant app (and within a JavaScript snippet through `app.parameters.parameterName`).

To give your parameter some context, you can add a description, which is optional. The text that you add in the Description field is for internal use only. It doesn't display in the instant app and isn't passed in the parameter.

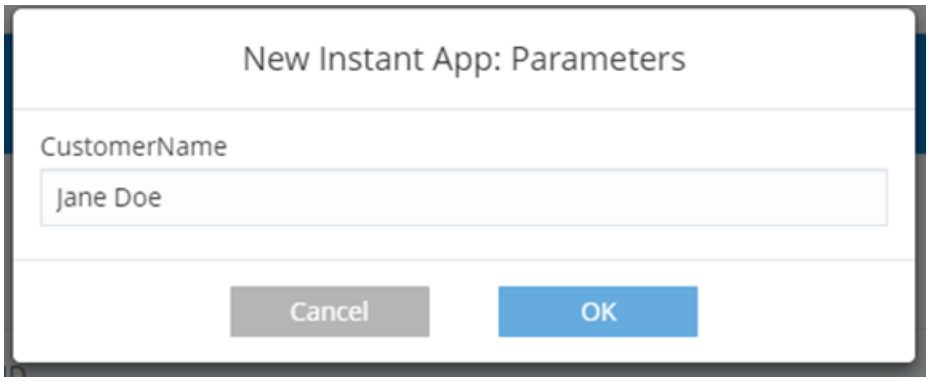


Here is an example of how to use a parameter to set a label and how to see it work in Test Mode.

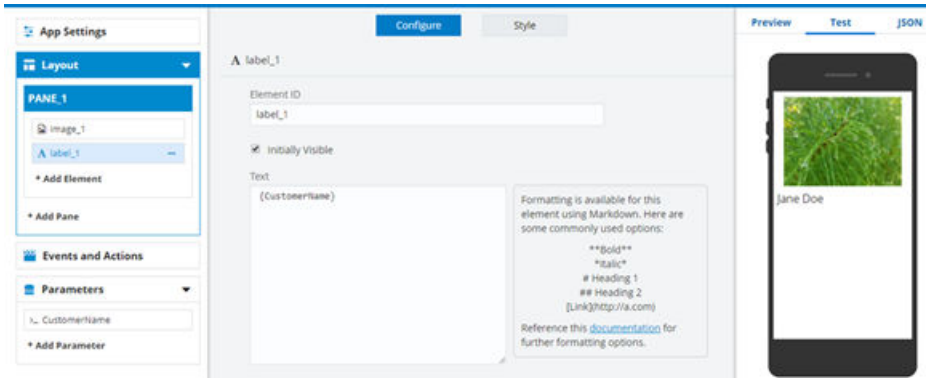
1. Create the parameter. In this case, it's called *CustomerName*.
2. Add a Label element. In the Text field, enter `{CustomerName}`.



3. On the right side of the Instant App Builder (where the phone is), select **Test** and then click **Start as Recipient**.
4. As the admin, you'll see the Parameter popup, where you can add a name like Jane Doe. Click **OK**.



You'll see Jane Doe show up on your instant app as your Label element text.



Using Brace Notation in Element and Parameter Values

When specifying a value for an action, you can both access and use parameters and other element values using brace notation:

- `{ElementName}`
- `{parameterName}`

When elements have complex object values, you can access their individual key values using the `{ElementName.key}` notation.

Barcode Element

- `{barCodeName.barcode}` —The numeric barcode
- `{barCodeName.barcode.type}`—The type of the barcode (AZTEC, CODABAR, CODE_39, CODE_93, or CODE_128)
- `{barCodeName.url}`—The URL of the barcode image
- `{barCodeName.html}`—The barcode and image as a HTML fragment

Upload Element

- `{uploadElementName.url}`—The URL of the uploaded data
- `{uploadElementName.html}`—The uploaded data as a HTML fragment
- `{uploadElementName.filename}`—The uploaded data's filename

Location Element

- `{locationElementName.latitude}`
- `{locationElementName.longitude}`
- `{locationElementName.url.google}`—The URL for a Google Maps location
- `{locationElementName.url.bing}`—The URL for a Bing location
- `{locationElementName.url.openStreetMap}`—The URL for an Open Street Map location
- `{locationElementName.url.hereMap}`—The URL for a Here Map location

Date Element

- `{dateElementName.day}`
- `{dateElementName.month}` — 0 = January, 11 = December
- `{dateElementName.year}`
- `{dateElementName.hour}`
- `{dateElementName.minute}`
- `{dateElementName.epoch}` —The number of milliseconds that have elapsed since 1/1/1970

> **Tip:**
>
> The epoch value is the easiest way to convert back to the JavaScript date:
>
> ```
> var d = new Date(app.Elements.date.value.epoch)
> ```

Modes

While you're developing your instant app, you can see your work in progress and find out how your instant app behaves at runtime using the following viewing modes that are located at the left of the Instant App Builder:

- Preview Mode
- Test Mode
- JSON

Preview Mode

The Preview mode provides you with the following visual guidelines and functions:

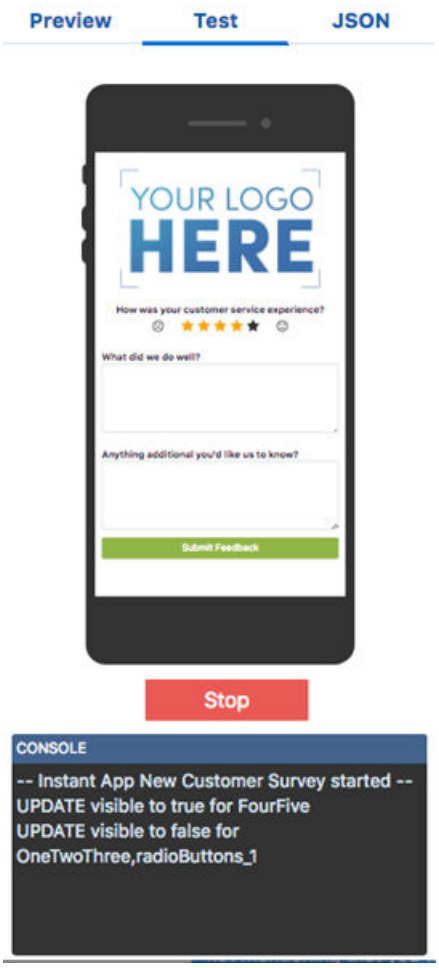
- The elements that you configure as initially invisible are shown with the hatch marks.
- The current element that you've selected or are editing is outlined in green.
- Panes are separated by name.

- Clicking an element in the Preview highlights that element in the Layout section and opens the element's editor.

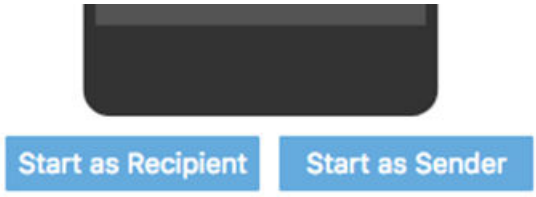
> **Note:**
>
> Changing values in Preview mode has no effect on the instant app and will not trigger events to fire.

Test Mode

In Test Mode (which you activate by clicking **Test**), you'll see a full preview as well as the Console, which is a running commentary on events and actions as they occur.



There are two choices when Test Mode starts: **Start as Recipient** and **Start as Sender**. Because an app can behave differently depending on sender or recipient role, you need to pick one of these options before testing the app so that you can assess a particular user experience. For example, you can configure an element's usability setting that limits the ability to edit a field to only the customer (the recipient).



Print to Console in JavaScript Snippets

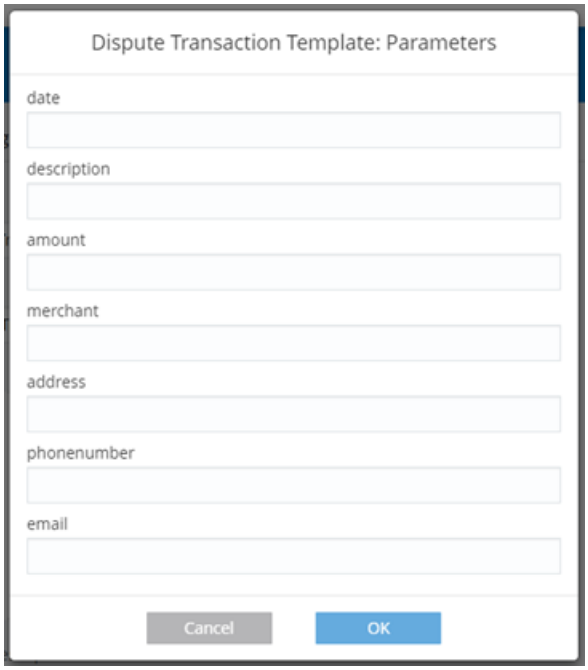
The Test Mode execution comes to a halt whenever you update the instant app (for example, when you edit JavaScript, change an element property or style, etc.), so you'll need to restart.

```
print("Hello World");
```

Limitations of Test Mode

There are few things to keep in mind while using Test Mode:

- The Call External Web API action can't call the target resource. Instead, it uses test data that you've provided in the **Test Response** field. You also need to provide a response code.
- You can't use the Lock App and Unlock App actions.
- If the instant app requires any parameter data, you'll be prompted it when Test Mode starts. See Parameters.



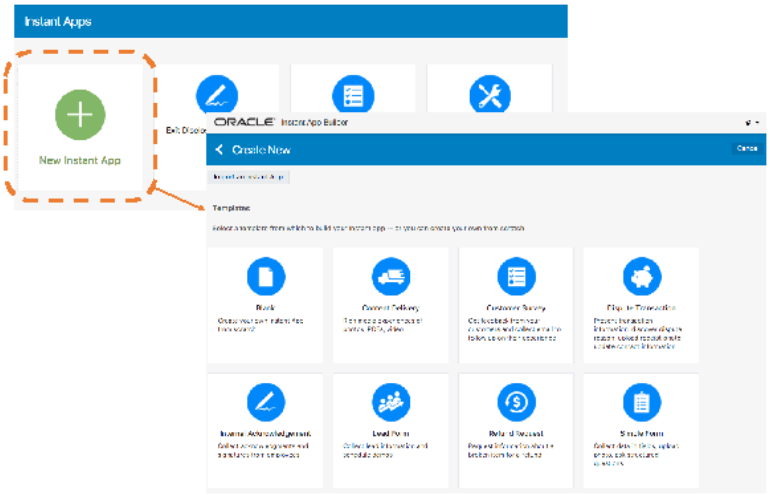
JSON

The JSON tab displays the complete specification for this instant app as expressed in JSON. While you can export this JSON, modify it using any code editor that you want, and then import it to the Instant App Builder, you can avoid the validation errors and other problems (like creating or improper or inconsistent schemas) the you might encounter if you take this route if you create the app entirely with the Instant App Builder. See Exporting and Importing.

Starting an Instant App from a Template

If you're not sure exactly where to start on your instant app, you can use template that you can customize to your business needs.

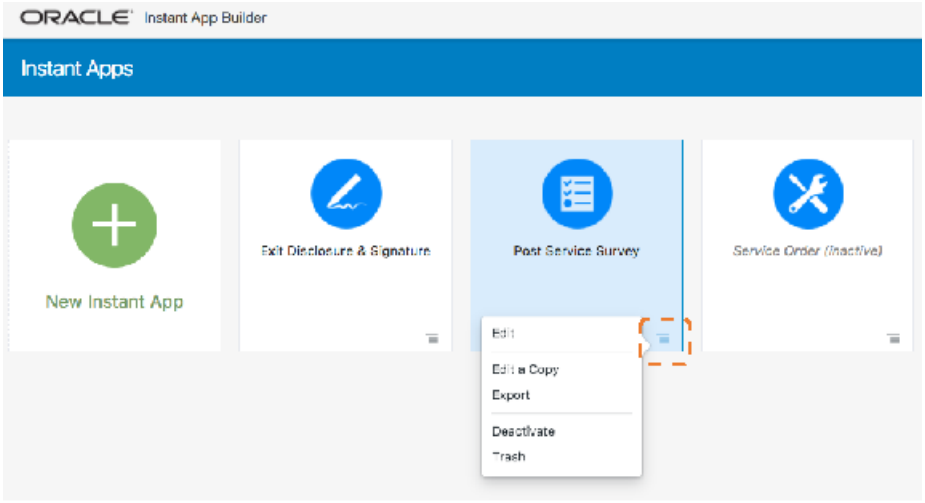
Similar to starting a new instant app from scratch, you click the New Instant App tile. But instead of choosing a blank instant app, you can scroll through the templates and choose the one that's most relevant to your use case. From there, the Instant App Builder opens for the template, allowing you to both customize it and save it to your library.



Instant App Lifecycle

You can manage your library of instant apps by activating and deactivating them, deleting them, making clones that you can edit, or by editing the app directly.

You access these management functions from hamburger menu on the bottom right of an instant app tile.



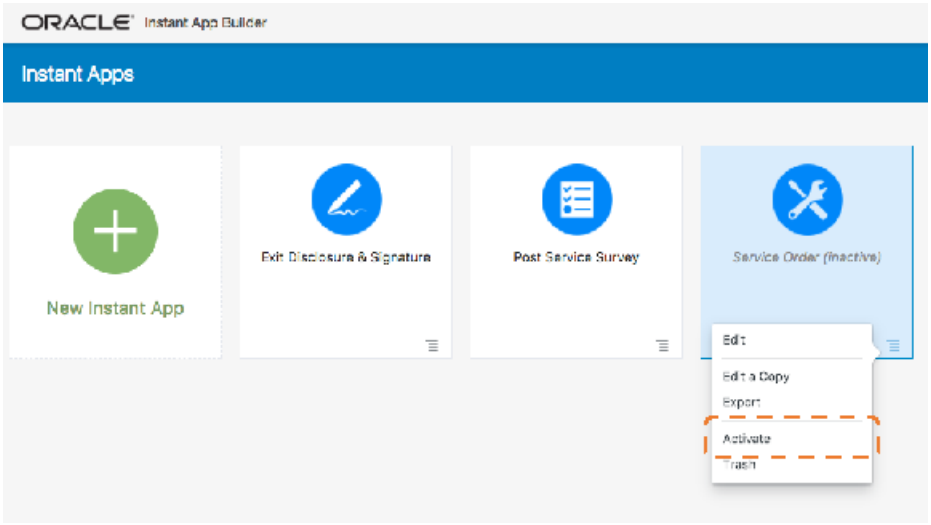
Editing

You can edit any instant app in your library in one of two ways: simple **Edit**, or **Edit a Copy**.

Simple editing takes you into the existing instant app to make changes. Once you've completed your changes, clicking **Save** overwrites the existing version. If you choose to **Edit a Copy**, your instant app will be cloned. Any changes that you make will be saved to a new instant app, with the default name *Copy of (original instant app name)*. Rename, make edits, and save your new instant app.

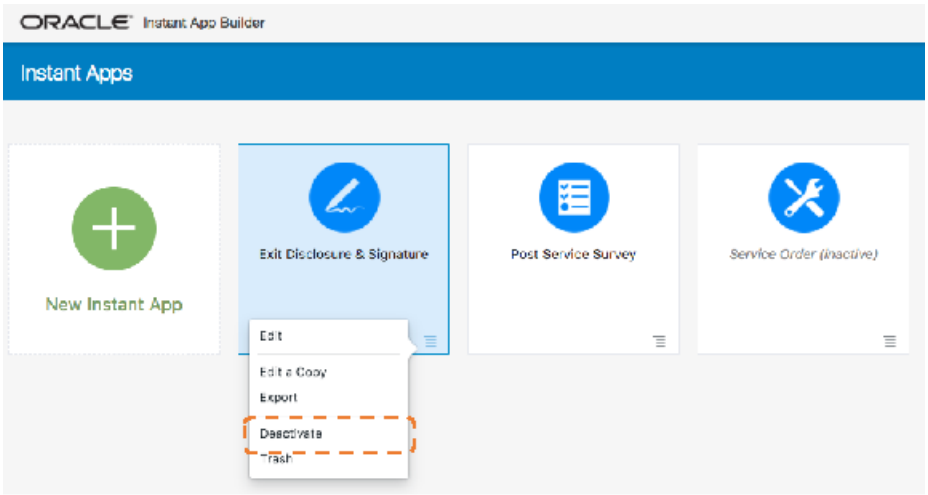
Publishing

Any inactive instant app displays with its name and description grayed out and italicized. If you an inactive instant app in your library, you can reactivate them by choosing **Activate** from the menu in the tile.



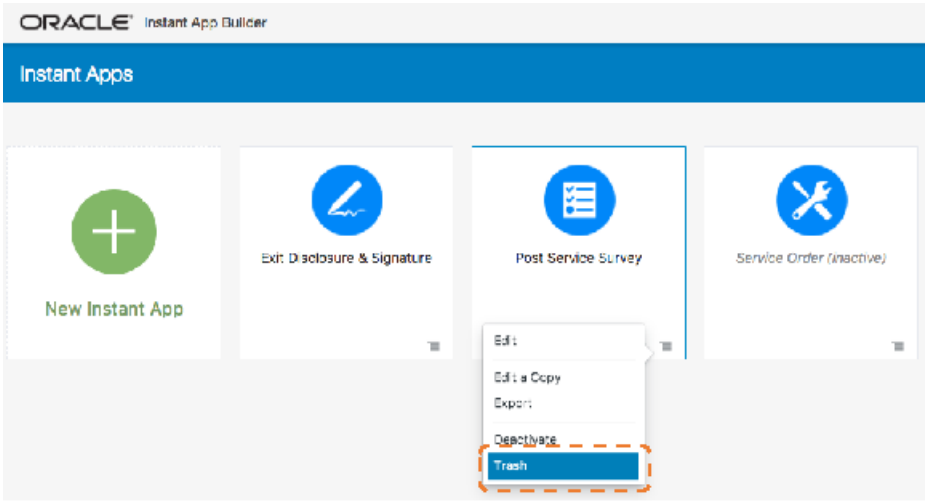
Deactivating

Similar to activating an instant app from your library, you can use the hamburger menu to deactivate an active instant app.

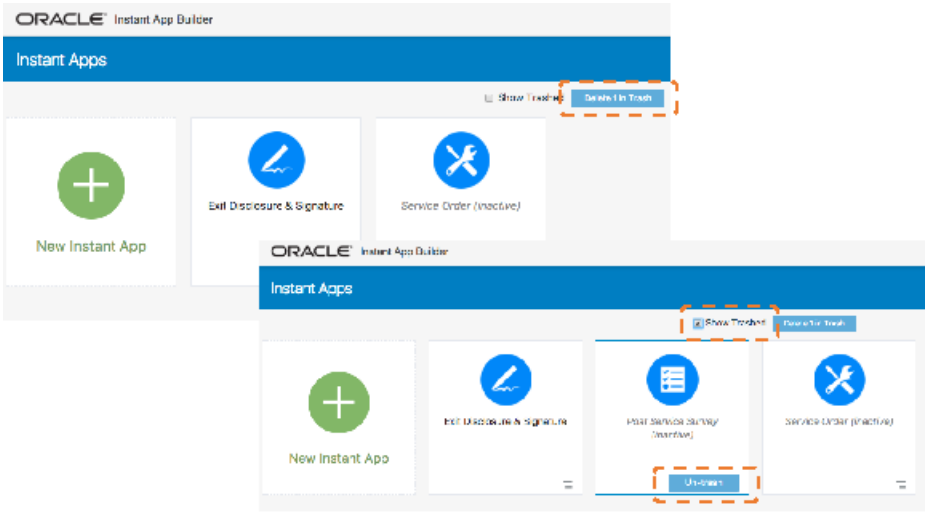


Deleting and Restoring

You can remove an instant app from your library by choosing **Trash** from the instant app's tile menu.



The **Trash** option doesn't permanently delete an instant app. Once you've trashed it, you will see options related to trashed items display at the top of the instant app library.

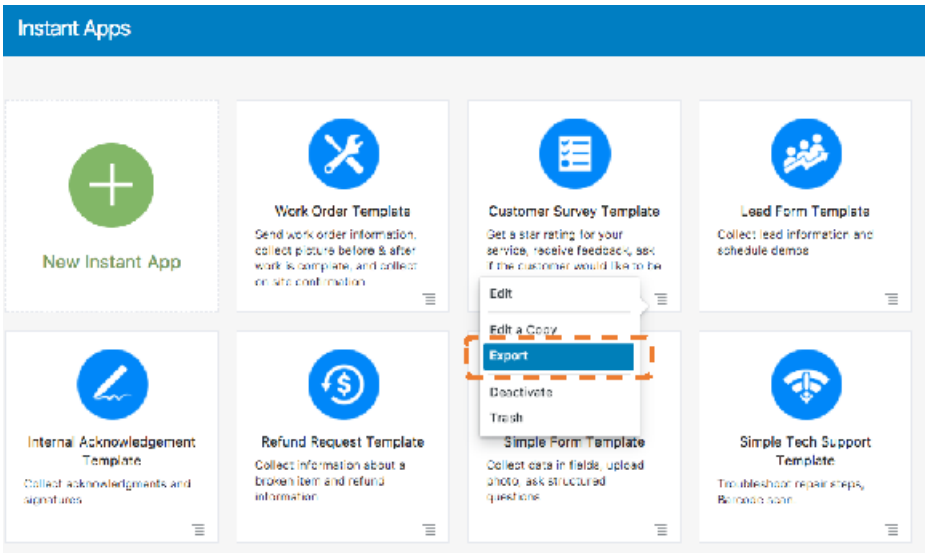


Clicking the **Delete (#) in Trash** permanently deletes any instant apps in your trash.

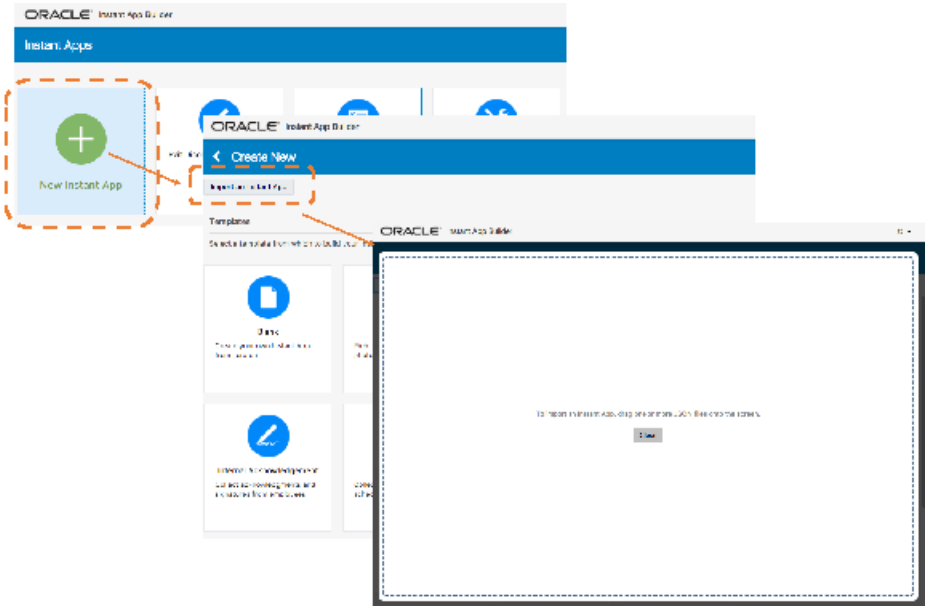
You can restore an instant app anytime by first selecting **Show Trashed**, identifying the instant app that you want to restore, and then clicking **Un-trash**.

Exporting and Importing

You can export any instant app into a JSON file. By doing this, you can edit the JSON directly using some other framework instead of the Instant App Builder. You can also export and import across different instances.



To import a JSON file into the Instant App Builder, click the green plus icon (**+**) to add a new instant app. Next, click **Import an Instant App**. From there, drag and drop the JSON file into the Instant App Builder, where you can modify and save your instant app.



14

Reference

- Built-In Components: Properties, Transitions, and Usage
- Apache FreeMarker Reference
- The SDK Helper Methods
- Navigation with keepTurn and transition
- The Custom Component Payload

Built-In Components: Properties, Transitions, and Usage

- Control Components
- Language
- Security
- User Interface Components
- Variable Components

Control Components

The control components route the flow based on whether the user input matches a predetermined value.

System.ConditionEquals

Use this component to check if the variable matches a value that has been passed in. To route the dialog according to the value, define the transitions key using `equal` and `notequal` actions.

Properties	Description	Required?
`variable`	The first value to be compared.	No

Properties	Description	Required?
source	In place of the variable property, you can name the source to be compared against the values properties. You can define this using a FreeMarker expression that compares a specific property of the system entity with the values property. For example: conditionEquals: component:"System.ConditionEquals" properties: source: "${addressVar.value.state} - ${addressVar.value.country}" value: "CA - USA" transitions: actions: equal: goCalfifornia notequal: goSomewhereElse Using FreeMarker expressions enables you to use System.ConditionEquals for other types of comparisons. For example: conditionEquals2: component: "System.ConditionEquals" properties: source: “<#if age.value gt 18>true<#else>false</#if>" value: true transitions: actions: equal: oldEnough notequal: tooYoung	No

Properties	Description	Required?

value	The second value to be compared. You can define this property with a FreeMarker expression:
	
	`verifyCode:`
	`  component:`
	`"System.ConditionEquals"`
	`  properties:`
	`  variable: "code"`
	`  value: "$`
	`{userEnteredCode.value}"`
	`  transitions:`
	`  actions:`
	`  equal: "wrongCode"`
	`  notequal: "$`
	`{flow.value}"`
		Yes

How Do I Use This?

This example shows how the `System.ConditionEquals` component can fork the dialog based on the value.

```
main: true
name: "Shoppingbot"
context:
  variables:
    yesnoVar: "YES_NO"

...

 confirmBuy:
    component: "System.ConditionEquals"
    properties:
      source: "${yesnoVar.value.yesno}"
      value: "YES"
    transitions:
      actions:
        equal: "deviceDone"
        notequal: "cancelOrder"
  deviceDone:
    component: "System.Output"
    properties:
      text: "Your ${devices.value} is on its way."
    transitions:
      return: "done"
  cancelOrder:
    component: "System.Output"
    properties:
      text: "Thanks for your interest."
    transitions:
      return: "done"
```

System.ConditionExists

Use this component to check for the existence of a specified variable. To route the dialog according to the value, define the transitions key using `exists` and `notexist` actions.

Properties	Description	Required?
`variable`	The name of the variable	Yes
`value`	The value that the Dialog Engine checks for.	Yes

```
main: true
name: "HelloKids"
context:
  variables:
    foo: "string"
    lastQuestion: "string"
    lastResponse: "string"
states:
  intent:
    component: "System.Intent"
    properties:
      variable: "iResult"
      confidenceThreshold: 0.4
    transitions:
      actions:
        Talk: "checkUserSetup"
        unresolvedIntent: "checkUserSetup"
  checkUserSetup:
    component: "System.ConditionExists"
    properties:
      variable: "user.lastQuestion"
      value: "unneccessary"
    transitions:
      actions:
        exists: "hellokids"
        notexists: "setupUserContext"
  setupUserContext:
    component: "System.CopyVariable"
    properties:
      from: "lastQuestion,lastResponse"
      to: "user.lastQuestion,user.lastResponse"
    transitions: {}

...
```

System.Switch

Use this component to switch states based on variable value.

Similar to the `System.ConditionEquals` component, you can define the state that you want to navigate to as a variable. See System.ConditionExists

Property	Description	Required?

`variable`	The name of the variable that's compared against the `values` properties. For example:
	
	`switchOnCategory:`
	`  component:`
	`"System.Switch"`
	`  properties:`
	`  variable: "category"`
	`  values:`
	`  - "Vehicle"`
	`  - "Property"`
	`  - "Other"`
	`  transitions:`
	`  actions:`
	`  Vehicle:`
	`"getVehicleQuote"`
	`  Property:`
	`"getPropertyQuote"`
	`  Other:`
	`"getOtherQuote"`
		No

Property	Description	Required?
source	In place of the variable property, you can name the source to be compared against the values properties. You can define this using a FreeMarker expression that compares a specific property of the system entity with the values property. For example: switch1: component: "System.Switch" properties: source: "$ {yesnoVar.value.yesno}" values: - "YES" - "NO" transitions: actions: YES: goYes NO: goNo switch2: component: "System.Switch" properties: source: "$ {startDate.value.date? string('dd-MM-yyyy')}" values: - "17-12-2017" - "18-12-2017" transitions: actions: ¿17-12-2017": goToday ¿18-12-2017": goTomorrow	No
values	The list of values that the Dialog Engine checks for.	Yes

Language

- System.Intent
- System.MatchEntity
- System.DetectLanguage
- System.TranslateInput
- System.TranslateOutput

System.Intent

This component detects the user intent and extracts all of the entities and then triggers a subsequent state.

Property	Description	Required?
`variable`	Holds the value that the Intent Engine resolves from the user input. You can define this property as `variable=iResult` (with `iResult: "nlpresult"` defined as one of the `context` variables). The response from the Intent Engine is stored in the `iResult` variable.	Yes
`confidenceThreshold`	The minimum confidence level required to match an intent. When your bot's confidence in matching any of its intents with the user message falls below this minimum value, the component triggers its `unresolvedIntent` action.	Yes
`optionsPrompt`	The title for the list of intents when the `confidenceWinmargin` is set. By default, this string value is `Do you want to`.	No
`confidenceWinMargin`	Sets the maximum level for the win margin, which is the delta between the respective confidence levels for the top intents that bot uses to resolve vague or compound user requests. The value that you set for this property should be greater than or equal to this delta. The intents separated by this delta are presented in a select list. To be included in the list, the intents must exceed the value set for the `confidenceThreshold`. The default value for the `confidenceWinMargin` property is 0.0.	No

Property	Description	Required?
`botName`	The name of the bot that resolves the intent. Use this property when you have a reusable bot that holds all of the intent and entity definitions. To support multiple languages, you can define this property with a variable expression that evaluates to a bot name based on the current language.	No
`sourceVariable`	The NLP engine resolves the intent using the `sourceVariable` as the input. You can combine this with the `System.TranslateInput` component and assign its value to a variable that's used as the input to the NLP engine. See System.TranslateInput to find out how.	No
`translate`	You can override the boolean value of the `autoTranslate` context variable here. If `autoTranslate` is not set, or set to `false`, you can set this property to `true` to enable autotranslation for this component only. If the `autotranslate` context variable is set to `true`, you can set this property to `false` to exclude this component from autotranslation.	No

How Do I Use This?

This component can be used to detect the user intent from free text input and can be used anywhere in the flow, as shown in the following snippet:

```
metadata:
  platformVersion: "1.0"
main: true
name: "FinancialBotMainFlow"
context:
 ...

states:
  intent:
    component: "System.Intent"
    properties:
      variable: "iResult"
      confidenceThreshold: 0.4
    transitions:
      actions:
        Balances: "startBalances"
```

```
        Transactions: "startTxns"
        Send Money: "startPayments"
        Track Spending: "startTrackSpending"
        Dispute: "setDate"
        unresolvedIntent: "unresolved"
...

  askPaymentAmount:
    component: "System.Text"
    properties:
      prompt: "What's the payment amount?"
      variable: "paymentAmount"
      maxPrompts: 1
    transitions:
      actions:
        cancel: "intentCheck"
...

  intentCheck:
    component: "System.Intent"
    properties:
      variable: "iResult2"
      confidenceThreshold: 0.4
    transitions:
      actions:
        Balances: "startBalances2"
        unresolvedIntent: "askPaymentAmount2"

...
```

The confidenceThreshold Property

When you add the `confidenceThreshold` property, you can steer the conversation by the confidence level of the resolved intent, which is held in the `iResult` variable.

If the intent's ranking exceeds the `confidenceThreshold` property (which, by default is 40%), then the action defined for that intent is triggered, setting the path for the Dialog Engine. In the opposite case—when the value for the `confidenceThreshold` property is higher than the ranking for the resolved intent—the Dialog Engine moves to the state defined for `System.Intent`'s `unresolvedIntent` action. See The Intent Tester.

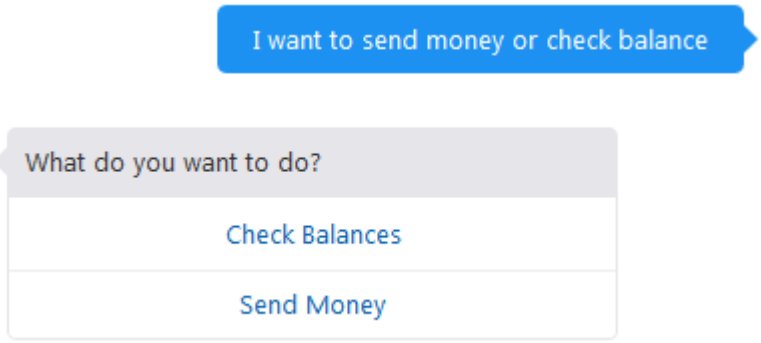
Taking the PizzaBot as an example, testing its intents with *I want to order pizza* resolves to 100%. When you enter the same phrase in the tester's Bots tab, however, the bot replies with *How Old Are You?*, a seemingly inappropriate response. Within the context of the PizzaBot dialog flow definition, however, this is the expected response for an intent whose ranking (100%) exceeds the confidence threshold (40%). When you enter 18, the `checkage` state's `allow: "crust"` action directs the Dialog Engine to the `crust` state. (Because there were no entities to extract from the initial user input, the Dialog Engine bypassed the `resolveSize` and `resolveCrust` states and ended up here after the age confirmation instead of completing the order.)

If you entered a wholly inappropriate phrase for the PizzaBot like *I want to buy a car* , the intent testing window will rank the top intent at only 25%, which is below the 40% threshold. Because neither the OrderPizza nor the CancelPizza intents can resolve the user input satisfactorily, the Dialog Engine moves to the state defined for the `unresolvedIntent` action (`unresolvedIntent: "unresolved"`). As a result, the bot responds with "I don't understand, what do you want to do?"

```
  unresolved:
    component: "System.Output"
    properties:
      text: "I don't understand. What do you want to do?"
    transitions:
      return: "unresolved"
```

The confidenceWinMargin Property

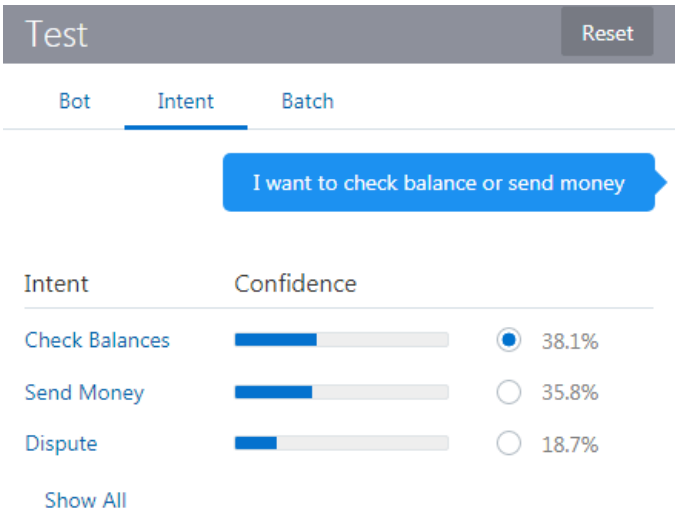
Adding the `confidenceWinMargin` property enables your bot to prompt users to pick an intent when it can't pick one. For example, if a user asks the FinancialBot, "I want to check balance or send money," the bot responds with a select list naming the top intents, Check Balances and Send Money.



The bot offers these two intents because its confidence in them exceeds 30% (`confidenceThreshold: 0.30` in the following snippet) and they're separated by a win margin—the difference between their respective confidence levels— that's within 15% (`confidenceWinMargin: 0.15`).

```
states:
  intent:
    component: "System.Intent"
    properties:
      variable: "iResult"
      optionsPrompt: "What do you want to do?"
      confidenceThreshold: 0.30
      confidenceWinMargin: 0.15
```

For example, the bot's confidence for the Check Balances intent is 38.1%. For the Send Money intent, it's 35.8%. The win margin is 2.3%, well within the 15% configured for the `System.Intent` component.



System.MatchEntity

The `System.MatchEntity` calls the Intent Engine to extract entity information from the text held by the `sourceVariable` property. If a match exists for the variable entity type, the variable is set with this entity value.

Property	Description	Required?
`sourceVariable`	The variable that holds the input value.	Yes
`variable`	The name of the context variable. The value of this variable can be used in a subsequent `System.SetVariable` component to extract a specific entity using a FreeMarker expression. For example, to extract an `EMAIL` entity value: `${userInputEntities.value.entityMatches['EMAIL'][0]}`	Yes

This component also has two predefined transitions, match and nomatch

Transition	Description
`match`	Directs the Dialog Engine to go a state when the entities match.
`nomatch`	Defines the Dialog Engine to go to a state when the entities don't match.

In the following snippet, `System.MatchEntity` component matches the user-provided value stored in the `mailInput` variable against the EMAIL entity type that's been defined for the `mailEntity` variable. If the user input satisfies the entity type by being an e-mail address, then the `System.MatchEntity` component writes this value to the `mailEntity` variable that's echoed back to the bot user (`"You entered $`

{mailEntity.value.email}"). When the values don't match, the Dialog Engine moves to the nomatch state.

> **Note:**
>
> The System.MatchEntity component resolves a single value.

```
context:
  variables:
    iResult: "nlpresult"
    mailInput: "string"
    mailEntity: "EMAIL"
states:
  intent:
    component: "System.Intent"
    properties:
      variable: "iResult"
      confidenceThreshold: 0.4
    transitions:
      actions:
        displayMailAdresses: "askMail"
        unresolvedIntent: "dunno"
  askMail:
    component: "System.Text"
    properties:
      prompt: "Please provide a valid email address"
      variable: "mailInput"
    transitions: {}
  matchEntity:
    component: "System.MatchEntity"
    properties:
      sourceVariable: "mailInput"
      variable: "mailEntity"
    transitions:
      actions:
        match: "print"
        nomatch: "nomatch"
  print:
    component: "System.Output"
    properties:
      text: "You entered ${mailEntity.value.email}"
    transitions:
      return: "done"
  nomatch:
    component: "System.Output"
    properties:
      text: "All I wanted was a valid email address."
    transitions:
      return: "done"
  dunno:
    component: "System.Output"
    properties:
      text: "I don't know what you want"
    transitions:
      return: "done"
```

System.DetectLanguage

Use this component to detect the user's language.

This component stores the language detected in a variable named `profile.languageTag`, so, if you want to find out which language has been detected, you can use `${profile.languageTag}` or `${profile.languageTag.value}`. Because this is a string variable, you don't necessarily need to add the `value` suffix.

```
context:
  variables:
    autoTranslate: "boolean"
    translated: "string"
    someTranslatedText: "string"
states:
  setAutoTranslate:
    component: "System.SetVariable"
    properties:
      variable: "autoTranslate"
      value: true
    transitions: {}
   detect:
    component: "System.DetectLanguage"
    properties: {}
    transitions: {}
```

System.TranslateInput

Use this component when you've activated a translation service, but you want to explicitly translate user input and not rely on the `autotranslate` facility.

This component takes the user input, translates it to English and then stores the translated text into a variable.

Property	Description	Required?
`variable`	The variable that holds the translated text.	Yes.
`source`	Specifies the text values to be translated.	No

In the following code snippet, this variable that holds the text string is called `translated`. It holds the English translation of the user's input, which the NLP engine uses as the source for the Intent resolution. Note that the `autoTranslate: "boolean"` context variable, which is required for autotranslation services, is defined.

```
context:
  variables:
    autoTranslate: "boolean"
    translated: "string"
...

states:
  translate:
    component: "System.TranslateInput"
    properties:
      source: "${somevar.value}" or // "Besoin de pizza"
```

```
      variable: "translated"
    transitions: {}
```

Using the sourceVariable Property

Because the `System.Intent`'s `sourceVariable` property holds the value processed by the component, you can use it with the `System.TranslateInput` component to insert translated text. The following snippet shows assigning the `translated` variable value so that it can be processed by the NLP engine.

```
translate:
    component: "System.TranslateInput"
    properties:
      variable: "translated"
    transitions: {}
  intent:
    component: "System.Intent"
    properties:
      variable: "iResult"
      sourceVariable: "translated"
      confidenceThreshold: 0.4
```

System.TranslateOutput

The `System.TranslateOutput` component allows you to translate text manually.

The `System.TranslateOutput` component takes the value defined for the `source` property. It translates the text into the language detected by the `System.DetectLanguage` component and then stores it in the `variable` property.

Properties	Description	Required?
`source`	The text to be translated, or a FreeMarker expression that references a variable whose value needs to be translated.	Yes
`variable`	Holds the translated text.	Yes

In this example, the System.Output component, which would otherwise display autotranslated text, still outputs translated text, but here it outputs the translation of the text defined for the `source` property.

```
  unresolvedTranslate:
    component: "System.TranslateOutput"
    properties:
      source: "Sorry I don't understand"
      variable: "someTranslatedText"
    transitions: {}
  unresolved:
    component: "System.Output"
    properties:
      text: "${someTranslatedText}"
    transitions:
      return: "unresolved"
```

Security

System.OAuthAccountLink

The `System.OAuthAccountLink` component enables the bot to make calls to a third-party service on behalf of the user.

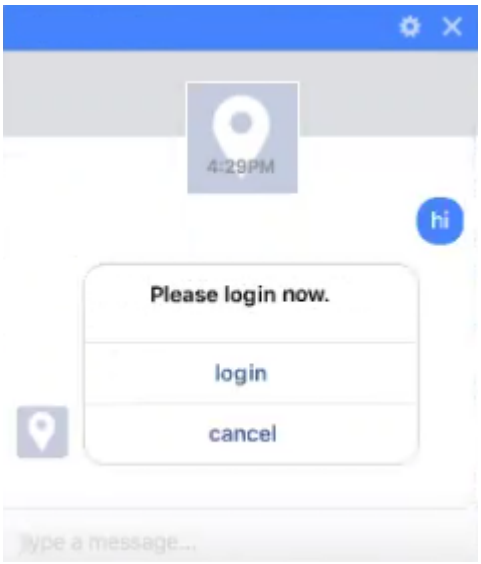
This component first directs a user to a third-party OAuth provider's login page. After a successful login, the bot then receives the access token, which it stores as a variable that's used by the custom component API. To handle the subsequent calls made through the channel, the custom component API exchanges the access token and a client secret for an OAuth 2 token. It makes another REST call to the OAuth2 provider, which accesses the secured API.

Property	Description	Required?
`prompt`	A text string that prompts the user to login.	Yes
`authorizeURL`	The login URL. See The authorizeURL Property.	Yes
`translate`	You can override the boolean value of the `autoTranslate` context variable here. If `autoTranslate` is not set, or set to `false`, you can set this property to `true` to enable autotranslation for this component only. If the `autotranslate` context variable is set to `true`, you can set this property to `false` to exclude this component from autotranslation.	No
`variable`	The name of the variable. You can declare it in the context node as a variable, a string, or another supported variable type. It can also be a user variable.	Yes

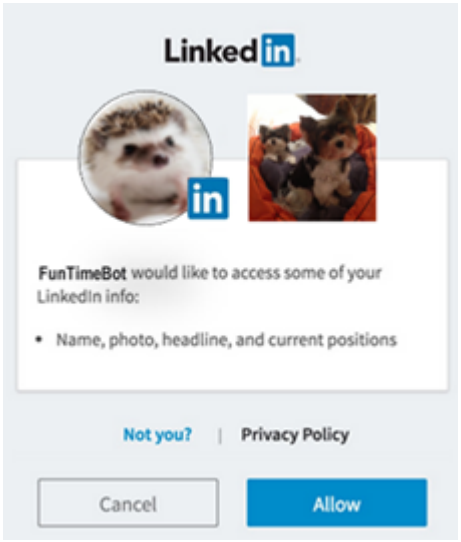
You can use this component to return access tokens from OAuth providers like Twitter, Google, Microsoft, or as shown in the following example, LinkedIn. This example shows the required properties that you need to define for the `System.OAuthAccountLink` component: `prompt`, which outputs the message, `variable`, which holds the returned code, and `authorizeURL` which defines both the provider's OAuth URL and the redirect URL that receives the token used by the bot to access the user's LinkedIn profile.

```
login:
  component: "System.OAuthAccountLink"
  properties:
    prompt: "Please login now."
    authorizeURL: "https://www.linkedin.com/uas/oauth2/authorization?
response_type=code&client_id=75k0vq4&scope=r_basicprofile&redirect_uri= https://
myBotsinstance/connectors/v1/callback"
    variable: "code"
  transitions:{}
```

When the Dialog Engine encounters this component, the bot prompts users to with two links, **Login** and **Cancel**.



The channel then renders the OAuth provider's login page or its authentication dialog as a webview.



> **Note:**
>
> The test window doesn't render webviews, so you need to cut and paste the link text into your browser.

The authorizeURL Property

To configure this property, you begin with the OAuth provider URL, such as `https://www.linkedin.com/oauth/authorization/` in the example. Next, you need to append the following OAuth parameters to this URL:

1. `response_type`—Set to `code` since the bot expects an authorization code.
2. `client_id`—An API key value that you get when you register your app with the OAuth provider.
3. `scope`—A list of permissions to access resources on the user's behalf. These are the permissions that you set when you register your app with the provider. They can vary by provider: for LinkedIn, these include `r_basicprofile` (shown in the example) and `r_emailadress`; for Microsoft, they're defined using `openid email` and `openid profile`.
4. `redirect_uri`—This is the redirect URI that you used to register your app with the OAuth provider that tells it where it needs to redirect users. This parameter, which is the Bots service host name appended with `connectors/v1/callback`, is the endpoint that receives the OAuth provider's token and then associates it with the active channel. The `redirect_uri` property is based on the Webhook URL that's generated when you create a channel. See Channels

> **Important:**
>
> Be sure that the value you enter for the `redirect_uri` matches the redirect URI that you provided when you registered your app exactly. In both instances, the URI must be appended with `connectors/v1/callback`.

User Interface Components

Use these components to display text:

- System.Text—Prompts the user to enter text.
- System.List—Prompts the user with a list option.
- System.Output—Displays a message.
- System.CommonResponse—Outputs content-rich messages.
- System.Interactive—Integrates your bot with an Instant App.

System.Text

The `System.Text` component enables your bot to set a context or user variable by asking the user to enter some text.

When the Dialog Engine enters a `System.Text` state for the first time, it prompts the user to enter some text. When the user enters a value, the Dialog Engine returns to this state. The component processes the user response and if it can convert the user input to the variable type, it stores the value in the variable. The Dialog Engine moves on to another state when this variable has a value.

> **Note:**
>
> The Dialog Engine skips over the `System.Text` state of the variable already has a value.

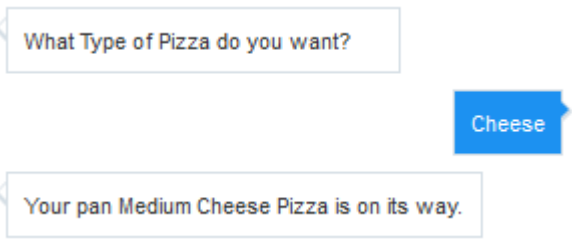
Property	Description	Required?
`prompt`	A text string that describes the input required from the user. You can dynamically add values to it using a value expression. For example: `Hello ${profile.firstName}, how many pizzas do you want?`	Yes
`variable`	The name of the variable, which can be either a user variable or one of the variables declared in the `context` node.	Yes
`nlpResultVariable`	Use this property when the `variable` property references some type of entity (custom or built-in). If the `variable` property initially holds a null value but the `nlpResultVariable` holds an entity that matches the entity type that you've set for the `variable` property, then `variable` property is set with this entity value and the dialog flow will then transition to the next state. You can obtain this entity match by simply adding the `nlpResultVariable` property; you don't need to create a separate `SetVariable` state to set the entity value.	No
`maxPrompts`	The number of times that component prompts the user for valid input. See Limiting the Number of User Prompts.	No
`translate`	Use this property to override the boolean value that you've set for the `autotranslate` context variable. If you haven't set this variable, or if you set it to `false`, then you can set this property to `true` to enable autotranslation for this component only. If you set the `autotranslation` variable is set to `true`, you can set this property to `false` to exclude this component from autotranslation. See Autotranslation.	No

See Transitions for the predefined action types that you can use with this component.

How Do I Use This?

In this example, the `type` variable holds the values expected by the `PizzaType` entity, like `cheese`, `Veggie Lover`, and `Hawaiian`. When this information is missing from the user input, the bot can still get it because its dialog flow transitions to the `type` state, whose `Text` component prompts them to explicitly state what they want. Keep in mind that even at this point, the user input still needs to resolve to the `PizzaType` entity to transition to the next state.

```
main: true
name: "PizzaBot"
parameters:
  age: 18
context:
  variables:
    size: "PizzaSize"
    type: "PizzaType"
    crust: "PizzaCrust"
    iResult: "nlpresult"

...

  type:
    component: "System.Text"
    properties:
      prompt: "What Type of Pizza do you want?"
      variable: "type"
    transitions: {}
```



System.List

Your bot can use the `System.List` component to set a user or context variable or set a transition action. The mode depends on whether a value can be set for the component's `variable` property (or if you configure a `variable` property in the first place).

When the Dialog Engine enters a `System.List` state for the first time, your bot displays a message containing a list of options. When the user clicks one of these options, the Dialog Engine returns to the `System.List` state to process the user response. If the component can convert the selected option to a user variable or one of the variables that you've defined in the `context` node, the `System.List`'s `variable` property it sets the `variable` property with this value. When this property can't be set (or hasn't been defined), the Dialog Engine triggers a transition action instead.

Property	Description	Required?
`options`	You can specify the `options` using comma-separated text strings, FreeMarker value expressions, and as a list of maps.	Yes
`prompt`	The text string that prompts the user.	Yes
`variable`	The name of the context or user variable that's populated when the user enters free text as a response instead of choosing a list option. When the user taps on a button rather than entering free text, the button payload determines which variable(s) are set and this property is ignored. When the Dialog Engine enters this state and the variable already has a value, then the state is skipped.	Yes (for value lists only)
`maxPrompts`	The number of times that component prompts the user for valid input. See Limiting the Number of User Prompts.	No
`nlpResultVariable`	Set this property when the `variable` property references an entity. If the referenced variable is null and this property has an entity match of the same type as the `variable` property, then the `variable` will be set with this entity value and the dialog flow will then transition to the next state. To get the match for this entity, add this property. You don't need to create a separate state for the System.SetVariable component to set the entity value.	No

Property	Description	Required?
`translate`	Use this property to override the boolean value that you've set for the `autotranslate` context variable. If you haven't set this variable, or if you set it to `false`, then you can set this property to `true` to enable autotranslation for this component only. If you set the `autotranslation` variable is set to `true`, you can set this property to `false` to exclude this component from autotranslation. See Autotranslation.	No

See Transitions for the predefined action types that you can use with this component.

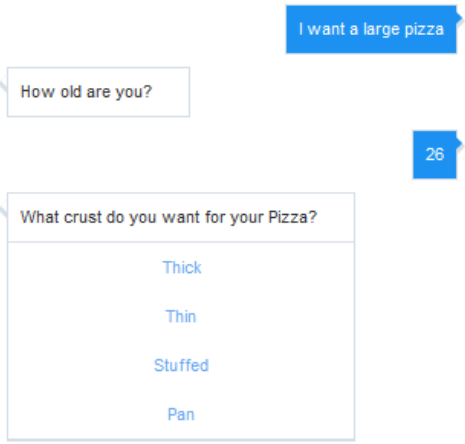
Value Lists

You can use the `System.List` component to return a value that satisfies a context variable that's defined as a primitive (like `greeting: "string"` in the dialog flow template) or as an entity, as shown in the following snippet. In this dialog flow, the `options: "Thick,Thin,Stuffed,Pan"` definition returns a value that matches `crust` variable. The `options` property defined for size is a value expression (`${size.type.enumValues}`) that returns the Large, Medium, Small, and Personal list values as options. See Accessing Variable Values with Apache FreeMarker FTL.

This example also shows how the `nlpResultVariable` property's `iResult` definition allows the component to set the entity values for the `variable` properties for the `crust` and `size` states when these values haven't been previously set. Like the `Text` component, the `System.List` component doesn't require any transitions (`{}`).

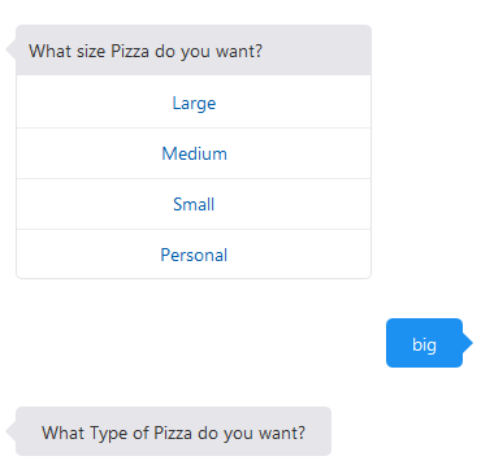
```
main: true
name: "PizzaBot"

...

context:
  variables:
    size: "PizzaSize"
    crust: "PizzaCrust"
    iResult: "nlpresult"

...

states:

...

crust:
  component: "System.List"
  properties:
    options: "Thick,Thin,Stuffed,Pan"
      prompt: "What crust do you want for your pizza?"
      variable: "crust"
```

```
main: true
name: "PizzaBot"

...

context:
  variables:
    size: "PizzaSize"
    crust: "PizzaCrust"
    iResult: "nlpresult"
...

states:

...

crust:
   component: "System.List"
   properties:
     options: "Thick,Thin,Stuffed,Pan"
     prompt: "What crust do you want for your pizza?"
     variable: "crust"
     nlpResultVariable: "iresult"
   transitions: {}
size:
   component: "System.List"
   properties:
     options: "${size.type.enumValues}"
       prompt: "What size Pizza do you want?"
       variable: "size"
       nlpResultVariable: "iresult"
    transitions: {}
```



Note:

Users aren't limited to the options displayed in the list. They can resolve the entity by entering a word that the entity recognizes, like a synonym. Instead of choosing from among the pizza size options in the list, for example, users can instead enter *big*, a synonym defined for the `PizzaSize` entity's *Large* option. See Custom Entities.



The options Property

You can set the `options` property using any of the following:

- A list of maps—While you can set the options property as a text string or value expression, you can also configure the `options` property as list of maps. Each one has a `label` property, a `value` property, and an optional`keyword` property. You can localize your list options when you follow this approach because, as noted by the following example, you can reference a resource bundle. See Resource Bundles to find out more about using the dot notation. When users enter a value that matches one the values specified in the `keyword` property, the bot reacts in the same way that it would if the user tapped the list option itself.

```
askPizzaSize:
  component: "System.List"
  properties:
    prompt: What size do you want?"
    options:
    - value: "small"
      label: "${rb.pizza_size_small}"
      keyword: "1"
    - value: "medium"
      label: "${rb.pizza_size_medium}"
      keyword: "2"
    - value: "large"
      label: "${rb.pizza_size_large}"
      keyword: "3"
   variable: "pizzaSize"
```

- A text string of comma-separated options, like "`small, medium, large`" in the following snippet. You can't add `label` and `value` properties when you define `options` as a string.

```
askPizzaSize:
  component: "System.List"
  properties:
    prompt: "What size do you want?"
    options: "small, medium, large"
    variable: "pizzaSize"
```

- An Apache FreeMarker value expression that loops over either a list of strings, or a list of maps, where each map must contain both the `label` and `value` properties and optionally, a `keyword` property.

```
askPizzaSize:
  component: "System.List"
  properties:
    prompt: "What size do you want?"
    options: "${pizzaSize.value.enumValues}"
    variable: "pizzaSize"
```

Refer to the Apache FreeMarker Manual to find out more about the syntax.

Action Lists

You don't need to define the `variable` property for a `System.List` option when you're configuring a list of actions. In this case, the component sets a transition action based on the option selected by the user. For example:

```
showMenu:
  component: "System.List"
  properties:
    prompt: "Hello, this is our menu today"
    options:

    - value: "pasta"
      label: "Pasta"
    - value: "pizza"
      label: "Pizza"

  transitions:
    actions:
      pasta: "orderPasta"
      pizza: "orderPizza"
```

> **Tip:**
>
> Not only can you use this approach to configure conditional navigation, you can use an action list in place of a `System.Switch` component.

System.Output

Use the `System.Output` component to output a message that doesn't require a user response, or doesn't require your bot to process the user's response. If you need to process the user's message, use either the `System.Text` or the `Systen.CommonResponse` component.

Your `System.Output` component definition requires the text property. As illustrated in the following example of a confirmation message, you can add value expressions to this string.

```
done:
    component: "System.Output"
    properties:
      text: "Your ${size.value}, ${type.value} pizza with ${crust.value} crust is on
its way. Thank you for your order."
```

By default, the Dialog Engine waits for user input after it outputs a statement from your bot. If you override this behavior, add the optional property called `keepTurn` to the `System.Output` component definition and set it to `true` to direct the Dialog Engine to the next state as defined by the `transitions` property. When no transition as been defined, the Dialog Engine moves to the next state in the sequence.

```
  wait:
    component: "System.Output"
    properties:
      text: "Please wait, we're reviewing your order"
      keepTurn: true
    transitions:
      next: "ready"
 waitmore:
    component: "System.Output"
    properties:
      text: "Almost done..."
      keepTurn: true
    transitions:
      next: "done"
  done:
    component: "System.Output"
    properties:
      text: "Your ${size.value}, ${type.value} pizza with ${crust.value} crust is on
its way. Thank you for your order."
    transitions:
      return: "done"
```

Use the `keepTurn` option when you want output multiple statements in quick succession and without user interruptions.

Autotranslation

You can suppress or enable the `System.Output` component's autotranslated text on a per-component basis using the `translate` property. By setting it to `false`, as in the following snippet, the components outputs the text as is, with no translation. By setting this property to `true` , you can enable autotranslation when the `autoTranslate` variable is either set to `false` or not defined. See Autotranslation.

> **Note:**
>
> Typically, you would not set the `autoTranslate` variable to `true` if you're translating text with resource bundles. We do not recommend this approach.

```
setAutoTranslate:
    component: "System.SetVariable"
    properties:
```

```
      variable: "autoTranslate"
      value: "true"
    transitions: {}
...
pizzaType:
   component: "System.Output"
   properties:
     text: "What type of pizza do you want?"
     translate: false
   transitions: {}
```

Defining Value Expressions for the System.Output Component

You can define one or more value expressions for the `text` property, as in the following snippet that uses different expressions for outputting the text for an order confirmation (pizza size and type).

```
confirmation:
    component: "System.Output"
    properties:
      text: "Your ${size.value} ${type.value} pizza is on its way."
    transitions:
      return: "done"
```

Your bot outputs raw text when these expressions return a null value for the variable. If you're defining the `text` property with multiple expressions, each one must return a value. Otherwise, your bot users will see output text like:

```
Your ${size.value} ${type.value} is on its way.
```

It's all or nothing. To make sure that your bot always outputs text that your users can understand, substitute a default value for a null value using the Apache Freemarker default value operator: `${size.value!\"piping\"} ${type.value!\"hot\"}`. The double quotes indicate that the default value is a not a variable reference, but is instead the constant value that the operator expects. For example:

```
text: "Your ${size.value!\"piping\"} ${type.value!\"hot\"} pizza is on its way."
```

Important:

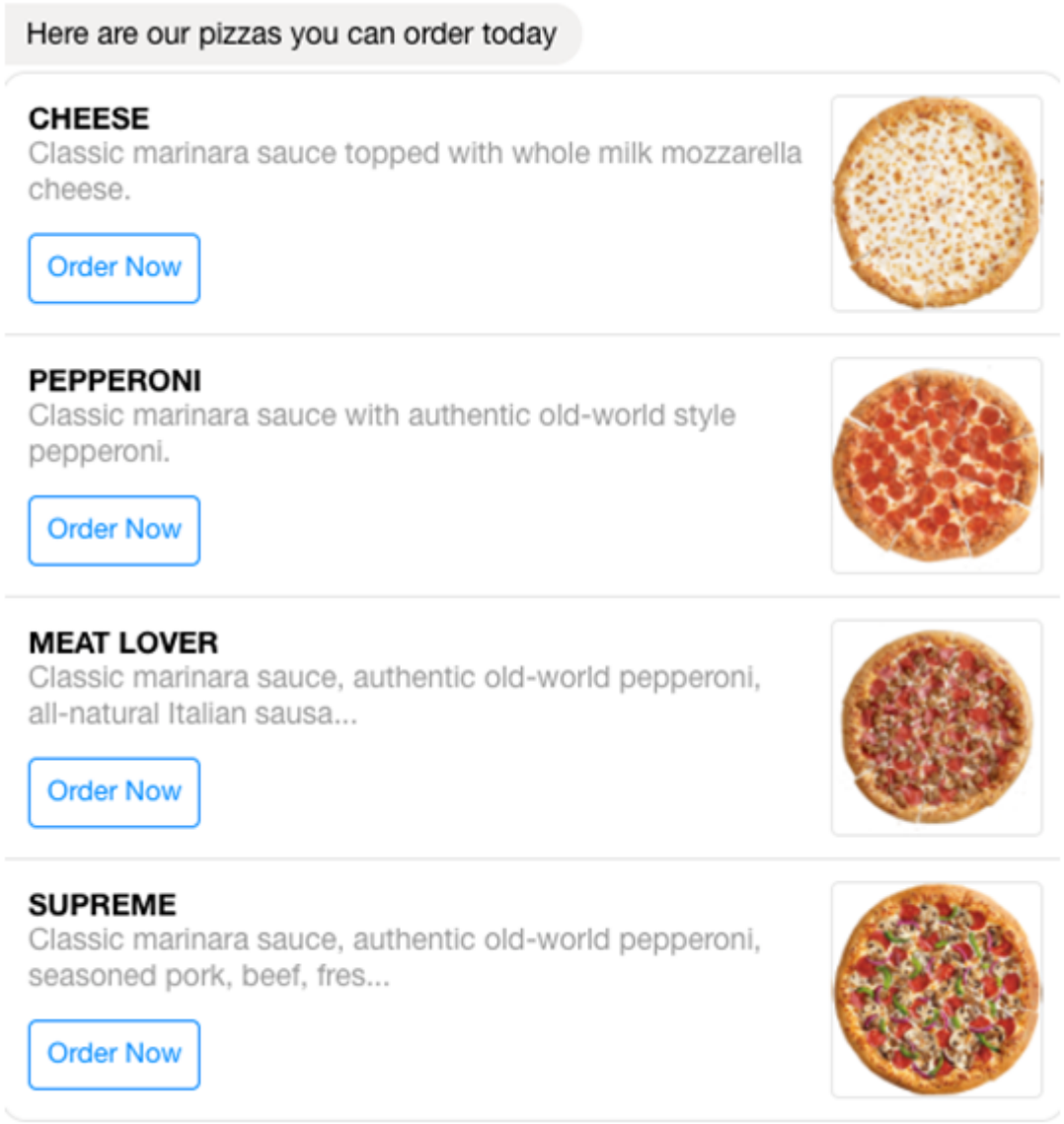
Always escape the quotation marks (`\"...\"`) that enclose the default value when you use the Freemarker operator. Your dialog flow's OBotML syntax won't be valid unless you use this escape sequence whenever you define a default value operation, or set off output text with double quotes. For example, the following `System.Output` component definition lets bot users see *You said, "Cancel this order."*

```
confirmCancel:
    component: "System.Output"
    properties:
      text: "You said, \"Cancel this order.\""
    transitions:
      return: "cancelOrder"
```

System.CommonResponse

The `System.CommonResponse` component enables you to build a specialized user interface that can include text, action buttons, images, and cards without having to write custom code. Instead, you define the component's properties and metadata.

You can see an example of using the `System.CommonResponse` component in the CrcPizzaBot, one of the sample bots. In this spin on the PizzaBot, you can display an image-rich menu with quick action “Order Now” buttons.

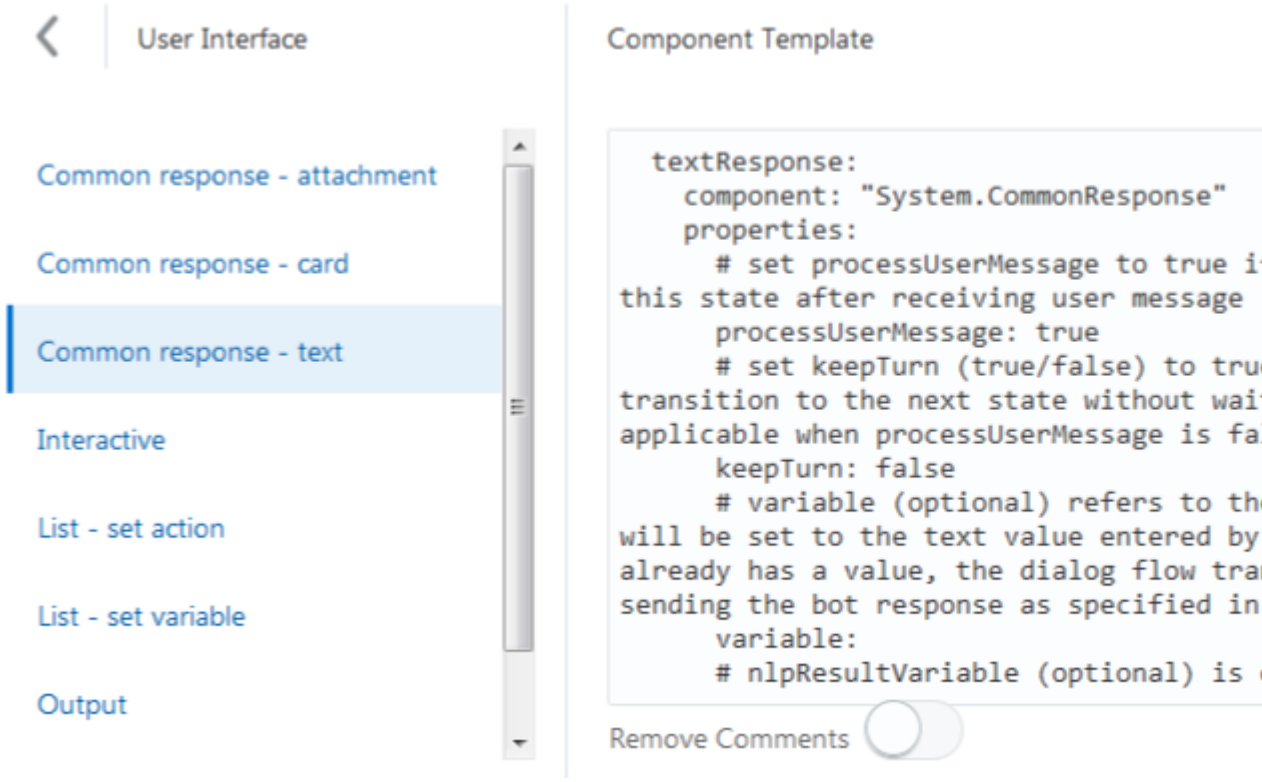


Within the context of the `System.CommonResponse` component, the different types of messages are known as “response types” and the CrcPizzaBot shows you how, among other things, they allow the bot users to respond to prompts using action buttons and view the pizza menu as a cascade of card items.

Adding a System.CommonResponse Component to Your Dialog Flow

Use the **Add Components** menu to add template `System.CommonResponse` states for the text, card, and attachment responses to your OBotML definition. These templates include the properties that are common to all of these response types as well as the ones that particular to each one. While the **Add Components** menu adds separate states for each response type, you can combine one or more response types into a single state. The CrcPizzaBot shows you examples of both in its `ShowMenu` (text

response) and `OrderPizza` (text and card responses) states.



The Component Properties

As shown in the following OBotML snippet from the CrcPizzaBot, configuring the `System.CommonResponse` component includes setting properties that direct the Dialog Engine along with metadata properties that describe not only how the component delivers messages (as text prompts, cards, or attachments), but also sets the content and behavior for the messages themselves.

```
  AskPizzaSize:
    component: "System.CommonResponse"
    properties:
      variable: "pizzaSize"
      nlpResultVariable: "iresult"
      maxPrompts: 2
      metadata:
        responseItems:
        - type: "text"
          text: "<#if system.invalidUserInput == 'true'>Invalid size, please try
again.\
            \ </#if>What size do you want?"
          name: "What size"
          separateBubbles: true
          actions:
          - label: "${enumValue}"
            type: "postback"
            payload:
              action: ""
              variables:
                pizzaSize: "${enumValue}"
            name: "size"
```

```
        iteratorVariable: "pizzaSize.type.enumValues"
  processUserMessage: true
transitions:
  actions:
    cancel: "Intent"
  next: "AskLocation"
```

> **Tip:**
>
> The `text` property in this snippet is defined using Apache FreeMarker Template Language (FTL). To find out how to add FTL expressions and use FreeMarker built-in operations to transform variable values, see Accessing Variable Values with Apache FreeMarker FTL.

Name	Description	Required?
`metadata`	The chat response created by this component is driven by the contents of the `metadata` property. See The Metadata Property.	Yes
`processUserMessage`	Set this property to `true` to direct the Dialog Engine to return to the state after the user enters text or taps a button. Set this property to `false` if no user input is required (or expected). When you set this property to `false`, the `System.CommonResponse` component behaves like the `System.Output` component.	Yes
`variable`	This variable holds the name of the context or user variable that gets populated when a user responds by entering free text instead of tapping a button. This property is ignored when a user taps a button, because the button's payload determines which variables values get set. If the variable property has already been set when the Dialog Engine enters this state, then the state is skipped.	No

Name	Description	Required?
`nlpResultVariable`	This property only applies when you set the `variable` as an entity-type variable. If the variable is null and the `nlpResultVariable` has an entity match that's the same type as the `variable` property, then the variable will be set with this entity value and the dialog flow will then transition to the next state. You can get this entity match by simply adding the `nlpResultVariable` property; you don't need to create a separate `System.SetVariable` state to set the entity value.	No
`maxPrompts`	Before the `System.CommonResponse` component can populate the variable value that you've specified for the `variable` property from the text entered by the user, it validates the value against the variable type. This can be entity-type validation, or in the case of a primitive type, it's a value that can be coerced to the primitive type. When the component can't validate the value, the Dialog Engine sends the message text and options again. (You can modify this message to reflect the validation failure.) To avoid an endless loop resulting from the user's inability to enter a valid value, set a limit on the number of attempts given to the user with the `maxPrompts` property. When the user exceeds this allotment, the `System.CommonResponse` component transitions to the `cancel` action. See Limiting the Number of User Prompts.	No
`keepTurn`	The `keepTurn` property only applies when you set the `processUserMessage` property to `false`. See System.Output to find out how to set this property.	No

Name	Description	Required?
`translate`	Use this property to override the boolean value that you've set for the `autotranslate` context variable. If you haven't set this variable, or if you set it to `false`, then you can set this property to `true` to enable autotranslation for this component only. If you set the `autotranslation` variable is set to `true`, you can set this property to `false` to exclude this component from autotranslation. See Autotranslation.	No

The Metadata Property

You define the metadata at two levels for the `System.ComponentResponse` component: at the root level, where you define the output and actions specific to the component itself, and at the response item level, where you define the display and behavior particular to the text, list, card, or attachment messages that are displayed by this component.

The component-level metadata describes the component's overall output in terms of the type of items, or messages, that it sends to the user along with any actions that particular to the component itself (and are independent of the message processing actions configured for the list items).

```
AskLocation:
  component: "System.CommonResponse"
  properties:
    variable: "location"
    metadata:
      responseItems:
      - text: "To which location do you want the pizza to be delivered?"
        type: "text"
        name: "What location"
        separateBubbles: true
      globalActions:
      - label: "Send Location"
        type: "location"
        name: "SendLocation"
```

Property	Description	Required?
`responseItems`	A list of response items, each of which results in a new message sent to the chat client (or multiple messages when you set iteration for the response item using the `iteratorVariable` property). Define these response items using these values: • `text`—Text bubbles (the text property) that can include a list of buttons that typically display as buttons • `cards`—A series of cards that scroll horizontally or vertically. • `attachment`—An image, audio, video, or file attachment.	Yes
`globalActions`	A list of actions that are not related to the specific response item. These actions are typically displayed at the bottom of the chat window. In Facebook Messenger, for example. these options are called quick replies.	No

You also configure the metadata for the various response items, such the text, card, or attachment messages.

Property	Description	Required?
`type`	The type of response item that determines the message format. You can set a message as `text`, `attachment`, or `cards`.	Yes
`name`	A name for the response item that's used for identification within the Bots platform. It's not used at runtime.	No
`visible`	Display properties.	No
	Property **Description**	

Property	Description		Required?
	`expression`	A boolean FreeMarker expression for conditionally showing or hiding text, a card, or attachment. For example, the CrcPizzaBot's `OrderPizza` state defines this property as follows: `expression: "<#if cardsRangeStart? number+4 < pizzas.value? size>true<#else>false</#if>"`	

	`channels:`
	`include:`
	`exclude:`
		For `include` and `exclude`, enter a comma-separated list of channel types for which the text, card, or attachment should be shown (`include`) or hidden (`exclude`). The valid channel values are:
		• `facebook`
		• `webhook`
		• `web`
		• `android`
		• `ios`
		• `twilio`
		• `kakaotalk`
		• `test`	

	`onInvalidUserInput`	A boolean flag that shows the text item or attachment either when the user enters valid input (`value=false`) or when the user enters input that's not valid (`value=true`).	

Property	Description		Required?
	`iteratorVariable`	Dynamically adds multiple text items to the response by iterating over the items stored in the variable that you specify for this property. Although you define the variable as a string, it holds JSON array when it's used as an iterator variable. You can reference properties in an object of the array with an expression like `${iteratorVarName.propertyName}`. For example, with an iterator variable named `pizzas`, the name property of a pizza can be referenced using the expression: `${pizzas.name}`.	No
`rangeStart`	If you've specified an `iteratorVariable`, you can stamp out a subset of response items by specifying the `rangeStart` property in combination with the `rangeSize` property. You can enter a hardcoded value or use a FreeMarker expression that references a context variable that holds the range start. By using a `rangeStart` variable, you can then page to the next set of data by setting the `rangeStart` variable in the payload of the browse option. You can see an example of the `rangeStart` and `rangeSize` properties in the CrcPizzaBot's `OrderPizza` state.		No
`rangeSize`	The number of response items that will be displayed as specified by the `iteratorVariable` and `rangeStart` properties.		No

`channelCustomProperties`	A list of properties that trigger functions that are particular to a channel. Because these functions are platform-specific, they're outside of the `System.CommonResponse` component and as such, can't be controlled by either the component's root-level or response item-level properties. You can find an example of this property in the CrcPizzaBot's `OrderPizza` state.
	
	`channelCustomProperties:`
	`- channel: "facebook"`
	`properties:`
	`top_element_style: "large"`
			No

The Action Metadata Properties

You can assign various actions to the response items.

Property	Description	Required?
`type`	The action type: • `postback`—Sends the payload of the action back to the Dialog Engine. • `share`—Opens a share dialog in the messenger client, enabling users to share message bubbles with their friends. • `call`—Calls the phone number that's specified in the payload. • `url`—Opens the URL that's specified in the payload in the browser. For Facebook Messenger, you can specify the`channelCustomProperties` property with `webview_height_ratio`, `messenger_extensions` and `fallback_url`. • `location`—Sends the current location. On Facebook Messenger, current location is not supported for text or card responses. It's only supported using a Quick Reply. For more information, see the Facebook Messenger Platform documentation.	Yes
`label`	A label for the action. To localize this label, you can use a FreeMarker expression to reference an entry in your bot's resource bundle.	Yes
`iteratorVariable`	Use this option to stamp out multiple actions by iterating over the items stored in the variable that you specify for this property. You can't use this property with the share and location actions.	No
`imageUrl`	The URL of image used for an icon that identifies and action. You can use this property to display an icon for the Facebook quick reply button (which is a global action).	No
`channelCustomProperties`	A list of properties that some trigger channel-specific functionality that isn't controlled by the standard action properties. You can find an example in the CrcPizzaBot's `OrderPizza` state.	No
`payload`	A nested payload object that has the following properties.	
	Payload Property / **Description**	
	`action` — The transition action set by the Dialog Engine when the user taps this action. You can only use this property for the postback action type.	No

Property	Description		Required?
	`variables`	When you set the action type to `postback`, the payload may have additional properties named after context variable or user variable. When the user taps this action, the variables are set to the values specified in this property. For the `unexpectedAction` transition, you can store the value for the unexpected action in the `user.botsUnexpectedAction` variable so that it's included in the postback payload. See Transitions.	No
	`url`	The URL of the website that opens when users tap this action.	Yes (only for the `url` action type)
	`phoneNumber`	The phone number called when a user taps this action.	Yes (only for the `call` action type)
`name`	A name that identifies the action on the Bots platform. This name is used internally and doesn't display in the message.		No
`visible`	Display properties.		No
	Property	**Description**	
	`expression`	A boolean FreeMarker expression for conditionally showing or hiding an action.	

Property	Description		Required?

	`channels:`
	`include:`
	`exclude:`
		For `include` and `exclude`, enter a comma-separated list of channel types for which the action can be shown (`include`) or hidden (`exclude`). The valid channel values are:
		• `facebook`
		• `webhook`
		• `web`
		• `android`
		• `ios`
		• `twilio`
		• `kakaotalk`
		• `test`	

	`onInvalidUserInput`	A boolean flag that shows the action either when the user enters valid input (`value=false`) or when the user enters input that's not valid (`value=true`).	

The Text Response Item

After you add a `textResponse` state to your dialog flow, you can rename it and then either replace the placeholder properties with your own definitions, or delete the ones that you don't need. The template state includes properties the following text-specific properties.

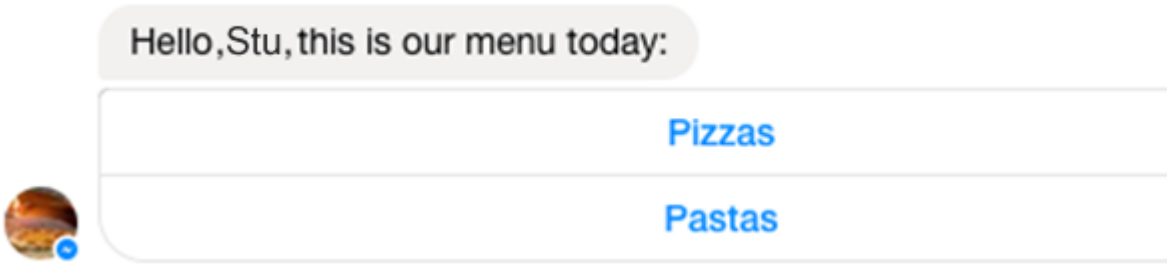
Property	Description	Required?
`text`	The text that prompts the user.	Yes
`iteratorVariable`	Dynamically adds multiple text items to the response by iterating over the items stored in the variable that you specify for this property. Although you define the variable as a string, it holds JSON array when it's used as an iterator variable. You can reference properties in an object of the array with an expression like `${iteratorVarName.propertyName}`. For example, with an iterator variable named `pizzas`, the name property of a pizza can be referenced using the expression: `${pizzas.name}`.	No

<table>
<tr><th>Property</th><th>Description</th><th>Required?</th></tr>
<tr><td>separateBubbles</td><td>You can define this property if you also define the iteratorVariable property. When you set this property to true, each text item is sent as separate message, like Pizzas and Pastas in the CrcPizzaBot's ShowMenu and OrderPizza states. If you set it to false, then a single text message is sent, one in which each text item starts on a new line.</td><td>No</td></tr>
<tr><td>visible</td><td>Text display properties
Property — Description
expression — A boolean FreeMarker expression for conditionally showing or hiding tex. For example, the CrcPizzaBot's OrderPizza state defines this property as follows:
<pre>expression: "<#if cardsRangeStart? number+4 < pizzas.value? size>true<#else>false</#if>"</pre>
channels:
include:
exclude: — For include and exclude, enter a comma-separated list of channel types for which the text should be shown (include) or hidden (exclude). The valid channel values are:
facebook
webhook
web
android
ios
twilio
kakaotalk
test
onInvalidUserInput — A boolean flag that shows the text item when the user enters valid input (value=false) or when the user enters input that's not valid (value=true).</td><td>No</td></tr>
</table>

In addition to the metadata properties, you can assign the following actions for a text response item.

- Postback
- Share
- Call
- URL
- Location

If you want to see an example of text response item, take a look at the CrcPizzaBot's `showMenu` state.



Because it names `postback` as an action, it enables the bot to handle unexpected user behavior, like selecting an item from an older message instead of selecting one from the most recent message.

```
ShowMenu:
  component: "System.CommonResponse"
  properties:
    metadata:
      responseItems:
      - type: "text"
        text: "Hello ${profile.firstName}, this is our menu today:"
        name: "hello"
        separateBubbles: true
        actions:
        - label: "Pizzas"
          keyword: "1"
          type: "postback"
          payload:
            action: "pizza"
          name: "Pizzas"
        - label: "Pastas"
          keyword: "2"
          type: "postback"
          payload:
            action: "pasta"
          name: "Pastas"
    processUserMessage: true
```

The Card Response Item

Like the `textResponse` state, you can rename the `cardResponse` state that's added to your dialog flow and then update the properties with your own definitions. Specifically, you can configure a card response item by defining the following properties. You can delete the properties that you don't need.

Property	Description	Required?
`cardLayout`	The card layout: `horizontal` (the default) and `vertical`.	Yes
`title`	The card title	Yes
`description`	The card description, which displays as a subtitle.	No
`imageUrl`	The URL of the image that displays beneath the subtitle.	No
`cardUrl`	The URL of a website. It displays as a hyperlink on the card that user open by tapping on it.	No
`iteratorVariable`	Dynamically adds multiple cards to the response by iterating over the items stored in the variable that you specify for this property. Although you define the variable as a string, it holds a JSON array when it's used as an iterator variable. You can reference properties in an object of the array with an expression like `${iteratorVarName.propertyName}`. For example, with an iterator variable named `pizzas`, the name property of a pizza can be referenced using the expression: `${pizzas.name}`.	No
`visible`	Card display properties	No
	`expression` — A boolean FreeMarker expression for conditionally showing or hiding tex. For example, the CrcPizzaBot's `OrderPizza` state defines this property as follows: `expression: "<#if cardsRangeStart? number+4 < pizzas.value? size>true<#else>false</#if>"`	

Property	Description		Required?

	`channels:`
	`include:`
	`exclude:`
		For `include` and `exclude`, enter a comma-separated list of channel types for which the card should be shown (`include`) or hidden (`exclude`). The valid channel values are:
		• `facebook`
		• `webhook`
		• `web`
		• `android`
		• `ios`
		• `twilio`
		• `kakaotalk`
		• `test`	

	`onInvalidUserInput`	A boolean flag that shows the text item when the user enters valid input (`value=false`) or when the user enters input that's not valid (`value=true`).	
`rangeStart`	If you've specified an `iteratorVariable`, you can stamp out a subset of cards by specifying the `rangeStart` property in combination with the `rangeSize` property. You can enter a hardcoded value or use a FreeMarker expression that references a context variable that holds the range start. Using a `rangeStart` variable, you can then page to the next set of data by setting the `rangeStart` variable in the payload of a browse option.		No
`rangeSize`	The number of cards that will be displayed as specified by the `iteratorVariable` and `rangeStart` properties.		No

You can assign a set of actions that are specific to a particular card, or a list of actions that are that are attached to the end of the card list.

The CrcPizzaBot's `OrderPizza` state includes a card response item definition, as shown in the following snippet:

```
cards:
          - title: "${pizzas.name}"
            description: "${pizzas.description}"
            imageUrl: "${pizzas.image}"
            name: "PizzaCard"
            iteratorVariable: "pizzas"
            rangeStart: "${cardsRangeStart}"
            rangeSize: "4"
            actions:
            - label: "Order Now"
```

```
    type: "postback"
    payload:
      action: "order"
      variables:
        orderedPizza: "${pizzas.name}"
        orderedPizzaImage: "${pizzas.image}"
    name: "Order"
```

The Attachment Response Item

The `attachmentResponse` state includes the following properties.

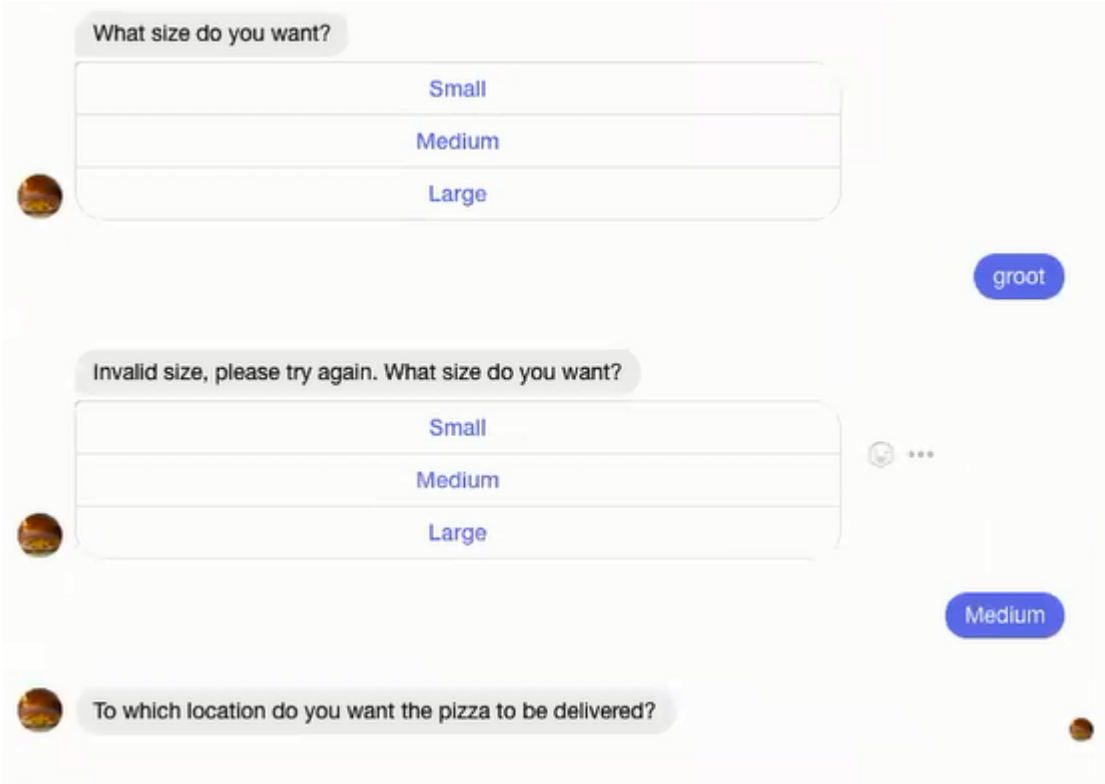
Property	Description	Required?
`attachmentType`	The type of attachment: `image`, `audio`, `video`, and `file`.	Yes
`attachmentURL`	The attachment's download URL or source.	Yes

The CrcPizzaBot's `Confirmation` state uses an attachment response item to display picture of the order, one that's different from the item pictured in the menu.

```
  Confirmation:
    component: "System.CommonResponse"
    properties:
      metadata:
        responseItems:
        - text: "Thank you for your order, your ${pizzaSize} ${orderedPizza} pizza\
            \ will be delivered in 30 minutes at GPS position $
{location.value.latitude},${location.value.longitude}!"
          type: "text"
          name: "conf"
          separateBubbles: true
        - type: "attachment"
          attachmentType: "image"
          name: "image"
          attachmentUrl: "${orderedPizzaImage}"
      processUserMessage: false
```

User Message Validation

The `System.CommonResponse`, `System.Text`, and `System.List` components validate the user-supplied free-text value that gets set for the `variable` property. For example, when the `variable` property is defined as a primitive type (string, boolean, float, double), these components try to reconcile the value to one of the primitive types. When the variable property is defined for an entity-type variable, these components call the NLP Engine to resolve the value to one of the entities. But when these components can't validate a value, your bot can display an error message.



By referencing the `system.invalidUserInput` variable, you can add a conditional error message to your bot's replies. This variable is a boolean, so you can use it as a condition with the FreeMarker `if` directive to display the message only when a user enters an invalid value. Otherwise, the message is hidden. The CrcPizzaBot's `AskPizzaSize` state in the following snippet demonstrates this by adding this variable as condition within a FreeMarker template that's evaluated by the `if` directive. Because it's set to `true`, the bot adds an error message to the standard message (*What size do you want?*) when the user enters an invalid value.

```
  AskPizzaSize:
    component: "System.CommonResponse"
    properties:
      variable: "pizzaSize"
      nlpResultVariable: "iresult"
      maxPrompts: 2
      metadata:
        responseItems:
        - type: "text"
          text: "<#if system.invalidUserInput == 'true'>Invalid size, please try
again.\
            \ </#if>What size do you want?"
          name: "What size"
          separateBubbles: true
```

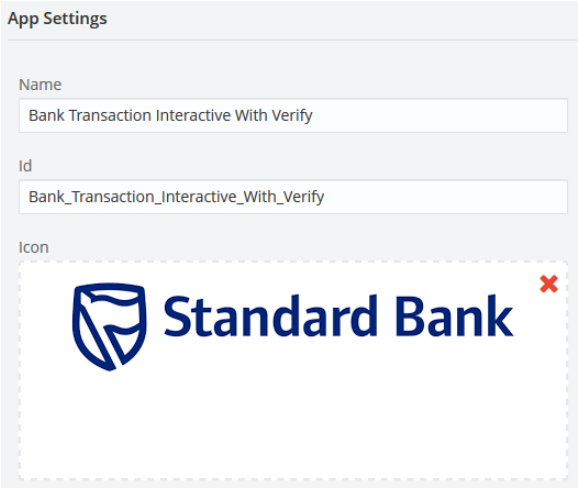
System.Interactive

Instant apps are rich, interactive widgets that you can embed as web links in your dialog. Your bot can transition to an instant app when it needs to data using structured forms.

After you create one with the Instant App Builder, you can integrate it into your OBotML definition using the `System.Interactive` component. See Instant Apps.

When you add this component using the Add Components menu, it generates a state called `interactive`, which has placeholders for the following properties.

Property	Description	Required?
`sourceVariableList`	A comma-separated list of context or user variable names. These variable names are the parameters that are that's sent to the instant app. You can set each variable by adding a series of `System.SetVariable` states before the `System.Interactive` state. To get an idea, take a look at the `setDate`, `setAmount`, `setMerchant`, and `SetDescription` states in the FinancialBot.	Yes
`variable`	The name of the variable (a string value) that identifies the instant app's callback payload. When the bot user completes the instant app, it sends a callback with a payload which is stored by this variable. At a later point in your OBotML definition, you can use this payload in a `System.Output` component.	Yes
`id`	The id of the instant app schema that gets instantiated. Enter the ID from the App Settings page of the Instant App Builder. See App Settings.	Yes



Property	Description	Required?
`prompt`	A text string. By default, this component outputs “Please tap on the link to proceed.”	No
`linkLabel`	The label for the button that invokes the instant app. The default string is `Link`.	No
`cancelLabel`	The label for the Cancel button that lets users leave the state without invoking the instant app. By default, the string is `Cancel`.	No
`translate`	Use this property to override the boolean value that you’ve set for the `autotranslate` context variable. If you haven’t set this variable, or if you set it to `false`, then you can set this property to `true` to enable autotranslation for this component only. If you set the `autotranslation` variable is set to `true`, you can set this property to `false` to exclude this component from autotranslation. See Autotranslation.	No

How Do I Use This?

The FinancialBotWithQnA uses an intent called `startDispute` to trigger an instant app. At runtime, when the Dialog Engine moves to the `System.Interactive` state called `startDispute`, the bot returns a link to the user. (This is the URL that’s configured for the Invite Message in the Instant App Builder. See App Settings.) The component identifies the instant app (`Bank_Transaction_Interactive_With_Verify`). The `sourceVariableList` names the variables whose values get passed to the instant app, namely date, merchant, amount, and description.

> **Note:**
>
> For the purposes of this reference bot, the values for these variables are populated with sample data through the `System.SetVariable` component.

The instant app defines parameter counterparts for each of these variables called `inputDate`, `inputDescription`, `inputMerchant`, and `inputDescription`. See Parameters and Using Brace Notation in Element and Parameter Values.

The snippet also shows how the bot returns the reason and dispute ID from the instant app’s return action. See Exit to Bot.

```
context:
  variables:
...
    dispute: "string"
```

```
    amount: "string"
    merchant: "string"
    date: "string"
    description: "string"
states:
  intent:
    component: "System.Intent"
    properties:
      variable: "iResult"
      confidenceThreshold: 0.4
    transitions:
      actions:
        Balances: "startBalances"
        Transactions: "startTxns"
        Send Money: "startPayments"
        Track Spending: "startTrackSpending"
        Dispute: "setDate"
        unresolvedIntent: "unresolved"
...

# Populate the required variables
  setDate:
    component: "System.SetVariable"
    properties:
      variable: "date"
      value: "2017-10-25T11:34:31Z"
    transitions: {}
  setAmount:
    component: "System.SetVariable"
    properties:
      variable: "amount"
      value: "$60"
    transitions: {}
  setMerchant:
    component: "System.SetVariable"
    properties:
      variable: "merchant"
      value: "PizzaUGotcha"
    transitions: {}
  setDescription:
    component: "System.SetVariable"
    properties:
      variable: "description"
      value: "restaurants"
    transitions: {}
# Call instant app
  startDispute:
    component: "System.Interactive"
    properties:
      sourceVariableList: "date, merchant, amount, description"
      variable: "dispute"
      id: "Bank_Transaction_Interactive_With_Verify"
    transitions: {}
# Use the callback payload data
  instantAppOutput:
    component: "System.Output"
    properties:
      text: "Successfully filed dispute, your reference number is '$
{dispute.value.disputeID}'\
       \ and reason is '${dispute.value.reason}'"
```

```
transitions:
  return: "instantAppOutput"
```

Transitions

The `System.CommonResponse` , `System.List`, and `System.Text` component use these transitions. See Message Handling for Output Components to find out how these transitions get triggered.

Transition	Description
`cancel`	Set this transition when a user exceeds the allotted attempts set by the `maxAttempts` property.

`textReceived`	Set this when users send text or emojis. For example:
	`ShowMenu:`
	`component: "System.CommonResponse"`
	`...`
	`processUserMessage: true`
	`transitions:`
	`actions:`
	`pizza: "OrderPizza"`
	`pasta: "OrderPasta"`
	`unexpectedAction: "HandleUnexpectedAction"`
	`textReceived: "Intent"`

`attachmentReceived`	Set this when a user sends an image, audio, video, or file attachment.
`locationReceived`	Set this when the user sends a location.
`unexpectedAction`	Set this to circumvent unexpected user behavior. Specifically, when a user doesn't tap an action item in the current message, but instead taps an action belonging to an older message in the chat session. You can access the unexpected action by referencing the `system.botsUnexpectedAction` variable.

Message Handling for Output Components

Typically, a user might respond to a message in the following ways:

- By entering free text.
- By sending their location.
- Using a multi-media option to send an image, audio file, video, or file attachment.
- Tapping one of the postback buttons displayed in the most recent message output by the bot.
- By scrolling to a previous message in the conversation and tapping one of its buttons.

Handling Free Text

When a user enters free text, the `System.CommonResponse`, `System.List` and `System.Text` components first validate the value. For valid values, the components trigger the `textReceived` transition. You don't have to set this transition in the OBotML definition; if you don't define this transition, then the Dialog Engine transitions to the next state, or performs the default transition.

> **Tip:**
>
> Use `textReceived` to handle unexpected user messages when you expect the user to tap a button, send an attachment, or a location.

Handling Multimedia Messages

When a users sends a file, image, video, or audio file, the `System.CommonResponse`, `System.List`, and `System.Text` component stores the attachment information as a JSON object in the variable property that's specified for the component. This object has the following structure:

```
{
  "type": "video",
  "url": "https://www.youtube.com/watch?v=CMNry4PE93Y"
}
```

For example, if a video attachment is stored in a variable called `myVideo`, you can access the video using the FreeMarker expression, `${myVideo.value.url}`. It also sets the transition action to `attachmentReceived`. You don't have to specify this transition in your OBotML definition. If you don't define this transition, then the Dialog Engine transitions to the next state, or performs the default transition.

> **Tip:**
>
> Use `attachmentReceived` to handle situations where users send an attachment unexpectedly.

Handling Location Messages

When a user sends his or her current location, the `System.CommonResponse`, `System.List`, and `System.Text` components store the location information as a JSON object in the variable property specified for the component. This object has the following structure:

```
{
  "title": "Oracle Headquarters",
  "url": "https://www.google.com.au/maps/place/…",
  "longitude": -122.265987,
  "latitude": 37.529818
}
```

For example, if the location is stored in a variable called location, you can access the latitude using the FreeMarker expression, `${location.value.latitude}`. It also triggers the `locationReceived` action, so you don't have to specify this transition in your OBotML

definition. If you don't define this transition, then the Dialog Engine transitions to the next state, or performs the default transition.

> **Tip:**
>
> Include `locationReceived` transition to handle situations where users send a location unexpectedly, or when you want to ensure that a user sends a location at the point where it's expected.

Handling Button Postback Actions

When a user taps a button in the most recent message, that button's payload gets processed. This payload is a JSON object which can hold an action property and one or more payload variables (all of which are optional). The payload's action is set as a transition action. Each user or context variable that's included in the payload is set to the value that's included in the payload. For example, when a user taps the Order Now button for a pepperoni pizza, the `System.List` or `System.CommonResponse` components receive the payload as a stringified JSON object:

```
{
    "action": "order",
    "state": "OrderPizza",
    "variables": {
        "orderedPizza": "PEPPERONI",
        "orderPizzaImage": "http://pizzasteven/pepperoni.png"
    }
```

In this example, the component parses the payload object with the order value, sets the transition to `order`, and sets the `orderedPizza` and `orderedPizzaImage` variables to the value specified in the payload.

Handling Button Postback Actions for an Older Message

A user might ignore the most recent message and instead scroll up and tap a button that's part of an earlier message. For example, user might tap the Order Now button for a pepperoni pizza, but is now asked which size. At this point, he might change his mind and click the Order Now button for another type of pizza, or he might decide on pasta rather than pizza, so he scrolls further up to a previous message and clicks Order Now for a pasta dish. When this happens, the component processes the button payload and sets any user or context variables as a postback action. But in this case, if the payload action has been specified, the action is not used to set the conversation transition. Instead, the `unexpectedAction` transition is triggered and the action value is stored in the `system.unexpectedAction` variable. This allows you to have one generic state to handle all of the unexpected user actions and messages.

```
...
defaultTransitions:
   unexpectedAction: "HandleUnexpectedAction"
states:
 OrderPizza:
    component: "System.CommonResponse"
    properties:
      metadata:
        responseItems:
        - type: "text"
          text: "Here are our pizzas you can order today"
```

```
...

      processUserMessage: true
    transitions:
      actions:
        order: "AskPizzaSize"
        more: "OrderPizza"
        textReceived: "Intent"

...

  HandleUnexpectedAction:
    component: "System.Switch"
    properties:
      variable: "system.unexpectedAction"
      values:
      - "pizza"
      - "pasta"
      - "order"
    transitions:
      actions:
        NONE: "ActionNoLongerAvailable"
        pizza: "OrderPizza"
        pasta: "OrderPasta"
        order: "AskPizzaSize"
```

Detecting Unexpected Actions

Because the built-in components that send bot messages with postback actions store the name of the state in the `state` property of the postback payload, they allow your bot to detect when a user does the unexpected by tapping a button from a previous message, essentially traversing backwards through the dialog flow. When a user taps this button, the name of the state is set for the postback's state property. The bot compares the payload's state name against the current state. When the two no longer match, the bot fires the `unexpectedAction` transition.

> **Note:**
>
> Only components that set the state property in the payload can enable the bot to respond when the user skips back in the flow. The `SystemOAuthAccountLink` doesn't set this property, so tapping the button on an older message can't trigger the `unexpectedAction` transition.

Limiting the Number of User Prompts

The `maxPrompts` property limits the number of times that the output components can prompt the user when they can't match the input value to any of the values defined for the entity or input type that's referenced by the `variable` property. While this is an optional property, adding it can prevent your dialog from going in circles when users repeatedly enter invalid values. You can set the maximum number of prompts using an integer (like 2 in the following snippet). The dialog moves onto the next state if the user enters a valid value before reaching this limit. Otherwise, the dialog transitions to the state defined by the `cancel` action. In the following sample, the dialog moves to the `setDefaultSize` state when users run out of chances. At this point, the bot makes their

choice for them, because the `System.SetVariable` component sets the pizza size to large.

```
size:
  component: "System.List"
  properties:
    prompt: "What size Pizza do you want?"
    options: "${size.type.enumValues}"
    variable: "size"
    maxPrompts: 2
  transitions:
    actions:
      cancel: "setDefaultSize"
setDefaultSize:
  component: "System.SetVariable"
  properties:
    variable: "size"
    value: "Large"
  transitions: {}
```

> **Note:**
>
> Setting the `maxPrompts` property to a negative number is same as not entering a value, or not including the property at all: the bot will continue to prompt the user until it receives a valid value.

Variable Components

System.SetVariable

The `System.SetVariable` component sets the value of a pre-defined variable. For example, you can set the value for an entity variable because this component can extract the entity match that's held by the `iResult` variable that's set for the `System.Intent` component.

Property	Description	Required?
`variable`	The name of the variable that's defined as one of the `context` properties. This can be a variable defined for an entity or a predetermined value, like a string.	Yes
`value`	The target value, which you can define as a literal or as a expression that references another variable.	Yes

The `startTxns` state in the following code snippet shows how you can define the target value using an expression that references another entity variable. In this case, `"${iResult.value.entityMatches['AccountType'][0]"` references the `iResult` variable that's resolved earlier in the flow by the `System.Intent` component. This variable sets the `accountType` variable if the `AccountType` entity is associated with the intent that's resolved by the `System.Intent` component. For example, if a user enters, "I want to

transfer money to checking," then this expression sets the `accountType` variable value to "checking." If the `System.SetVariable` component can't find matches, then the Dialog Engine moves on to the next state in the dialog flow (declared by `transitions: {}`).

```
main: true
name: "FinancialBotMainFlow"
context:
  variables:
    accountType: "AccountType"
    txnType: "TransactionType"
    txnSelector: "TransactionSelector"
    toAccount: "ToAccount"
    spendingCategory: "TrackSpendingCategory"
    paymentAmount: "string"
    iResult: "nlpresult"
states:

...

  startTxns:
    component: "System.SetVariable"
    properties:
      variable: "accountType"
      value: "${iResult.value.entityMatches['AccountType'][0]}"
    transitions: {}
...
```

Besides entity variables, you can set a predetermined value for a variable using an Apache FreeMarker expression or, as shown in the following snippet, a literal. You can find out more about FreeMarker here.

```
setOAuthRedirectURL:
    component: "System.SetVariable"
    properties:
      variable: "redirectURL"
      value: "https://thatcompany.io/connectors/v1/tenants/5c824-45fd-b6a2-8ca/
listeners/facebook/channels/78B5-BD58-8AF6-F54B141/redirect"
    transitions: {}
```

See System.OAuthAccountLink

System.ResetVariables

This component resets the values of the variables to null. This component doesn't require any transitions (use `transitions :{}`).

Property	Description	Required?
`variableList`	A comma-separated list of variable names.	Yes

System.CopyVariables

Copies the variable values.

Define this component using `from` and `to` properties as in the following snippet, where the value is copied to a user context:

```
setupUserContext:
    component: "System.CopyVariables"
    properties:
      from: "lastQuestion,lastResponse"
      to: "user.lastQuestion,user.lastResponse"
```

This component needs both of these properties, but their definitions don't have to mirror one-another. While you can define both `from` and `to` as lists of variables, you can also define `from` with a single variable and `to` as a list. If you set an additional `to` property, it inherits the variable value of the proceeding `from` property.

Apache FreeMarker Reference

- Built-In String FreeMarker Operations
- Built-In FreeMarker Number Operations
- Built-In FreeMarker Array Operations
- Built-In FreeMarker Date Operations

Built-In String FreeMarker Operations

The following table shows you how to use some of the built-in string operations using a string variable called `tester` as an example. As shown in the following snippet, its value is set to `"hello world "` (with three trailing blank spaces):

```
context:
  variables:
    tester: "string"
…
states:
  setVariable:
    component: "System.SetVariable"
    properties:
      variable: "tester"
      value: "hello world   "
```

> **Note:**
>
> The following `text` property definition allows the bot to output either the `tester` value, or, `no string found` if no value has been set for the variable.
>
> ```
> printVariable:
>   component: "System.Output"
>   properties:
>     text: "${tester.value!'no string found'}"
>   transitions: {}
> ```

Built-In Operation	Usage	Output
`capitalize`	`${tester.value?capitalize}`	`Hello World`
`last_index_of`	`${tester.value? last_index_of('orld')}`	`7`

Built-In Operation	Usage	Output
left_pad	${tester.value?left_pad(3,'_')}	___hello world
length	${tester.value?length}	14
lower_case	${tester.value?lower_case}	hello world
upper_case	${tester.value?upper_case}	HELLO WORLD
replace	${tester.value?replace('world', 'friends')}	hello friends
remove_beginning	${tester.value?remove_beginning('hello')}	world
trim	${tester.value?trim}	hello world (the trailing three spaces are removed)
ensure_starts_with	${tester.value?ensure_starts_with('brave new ')}	brave new hello world
ensure_ends_with	${tester.value?ensure_ends_with(' my friend')}$	hello world my friend
contains	${tester.value?contains('world')?string('You said world', 'You did not say world')}	You said world The contains('world') expressions returns either true or false. These boolean values are replaced with a string using the string('string1','string2') function.
ends_with	${tester.value?ends_with('world')?string('Ends with world', 'Doesn't end with world')}	Ends with world
starts_with	${tester.value?starts_with('world')?string('Starts with world', 'Doesn't start with world')}	Doesn't start with world
matches (regular expression returns true or false)	${tester.value?matches('^([^0-9]*)$')}	The regular expression returns true or false depending on whether the value contains a number (in which case the boolean value is returned as false). The tester value returns true.

Built-In Operation	Usage	Output
`matches` (regular expression returns a string)	`${tester.value?matches('^([^0-9]*)$')?}`	Same as above, but this time, `true` is returned as a string. The `matches('regular expression')` function returns `true` or `false` as boolean types. To print `true` or `false` in a `System.Output` component, use `?string` to perform a to-string conversion. **Note**: regular expressions can't be used in expressions that return groups. Use them in expressions that returns a single match or no match.

Example: Improving the Confidence Level with Casing

While The casing of the user input can impact the confidence level of the intent resolution. For example, *May* might refer to the month or the verb and user input can be erratic (Pizza, piZza, PIZZA). Instead of catching all of the possible case variations as synonyms in the entity definition, you can make the casing uniform using the an FTL operator like `lower_case` in the following snippet.

```
getIntent:
    component: "System.Text"
      properties:
        prompt: "Hi, I am a the Pizza Palace bot. How can I help?"
        variable: "userstring"
      transitions: {}
toLowercase:
    component: "System.SetVariable"
      properties:
      variable: "userstring"
      value: "${userstring.value?lower_case}"
    transitions: {}
intent:
     component: "System.Intent"
      properties:
      variable: "iResult"
      confidenceThreshold: 0.8
      sourceVariable: "userstring"
    transitions:
      actions:
      orderPizza: "orderPizza"
      cancelOrder: "cancelOrder"
      unresolvedIntent: "handleUnresolved"
```

To implement this, you first ask the for the user input using the `System.Text` component. In this example, the `System.Text` component saves the user input in the `userstring` variable. The `Sytem.SetVariable` uses FTL to change the case of the user input string to lower case and saves the modified string to the same `userstring` variable. Finally, the `userstring` variable is referenced by the `System.Intent` component using the `sourceVariable` property to run the modified user string against the intent engine.

Example: Transforming Case with the System.Switch Component

Another component that can be simplified with FTL is System.Switch.

In the following snippet shows different states that get called depending on the user input (wine or beer), which is stored in the `choice` variable.

```
switch:
    component: "System.Switch"
    properties:
      variable: "choice"
      values:
      - "wine"
      - "beer"
    transitions:
      actions:
        wine: "serverWine"
        beer: "serveBeer"
        NONE: "serveWater"
```

The casing of the input collected using the `System.Text` component may inconsistent, even within a word (WiNE). Instead of adding all possible variations to the `System.Switch` definition, use an FTL operation like `upper_case` to make the casing uniform:

```
switch:
  component: "System.Switch"
  properties:
    source: "${choice.value?upper_case}"
    values:
    - "WINE"
    - "BEER"
transitions:
 actions:
  WINE: "serveWine"
  BEER: "serverBeer"
  NONE: "serveWater"
```

Example: Concatenating FTL Expressions

The following snippet shows how concatenating FTL expressions transforms user input UA1234 and UA 1234, to simply 1234.

```
normalizeFlightNumber:
  component: "System.SetVariable"
  properties:
    variable: "flight"
    value: "${flight.value?trim?lower_case?remove_beginning('ua ')
            ?remove_beginning('ua')}"
```

Built-In FreeMarker Number Operations

The following table lists the built-in number operations and shows how they output the value set for the `negativeValue` (-2.5) and `positiveValue` (0.5175) context variables in the following snippet.

```
context:
  variables:
```

```
    negativeValue: "float"
    positiveValue: "float"
states:
  setNegativeValue:
    component: "System.SetVariable"
    properties:
       variable: "negativeValue"
       value: -2.5
setPositiveValue:
  component: "System.SetVariable"
  properties:
    variable: "positiveValue"
    value: 0.5175
```

Operation	Example	Output
`abs`	`${negativeValue.value?abs}`	2.5 The operator turns the negative numeric value into a positive value.
`string` (used with a numerical value)	`${negativeValue.value?abs?string.percent}`	250% The operator first changes the negative value to a positive. Then it converts it into percent, implicitly multiplying the value by 100.
`string` (with the decimal format value and various currencies) **Tip**: Check out Charbase for other currency symbols.	`${positiveValue.value?string['###.##']}`	0.51
	`${positiveValue.value?string['###.##%']}`	51% The operator adds adding a percentage character after multiplying the value by 100.
	`${positiveValue.value?string['##.###\u00A4']}`	0.51 $
	`${positiveValue.value?string['##.###\u20AC']}`	0.51 €
	`${positiveValue.value?string['##.###\u00A3']}`	0.51 £
`round`	`${negativeValue.value?round}`	-2 The operator rounds to the nearest whole number. If the number ends with .5, then it rounds upwards.
	`${positiveValue.value?round}`	1 The operator rounds to the nearest whole number. If the number ends with .5, then it rounds upwards.
`floor`	`${positiveValue.value?floor}`	0 The operator rounds downwards.
`ceiling`	`${positiveValue.value?ceiling}`	1 The operator rounds upwards.

Operation	Example	Output
`lower_abc`	`${negativeValue.value?abs?round?lower_abc}`	c The operator turns the negative value into a positive, then rounds it to 3. It returns c, the third letter of the alphabet.
`upper_abc`	`${negativeValue.value?abs?round?upper_abc}`	C The operator turns the negative value into a positive, then rounds it to 3. It returns C, the third letter of the alphabet.
`is_infinite`	`${positiveValue.value?is_infinite?string}`	false The operator returns false, because a float value is not infinite according to IEEE 754 (Standard for Floating-Point Arithmetic). **Note**: The returned value would be a boolean without ? `string`.

Built-In FreeMarker Array Operations

Array (or sequence) operations enable your bot to, among other things, determine the size of an array, sort arrays, or find content within an array.

Arrays return the results from the intent and entity processing. For example:

- `${iResult.value.entityMatches['name of entity']}` returns an array of entities found in a user string that's passed to the `System.Intent` component and stored in the `iResult: nlpresult` variable.
- `${iResult.value.intentMatches.summary}` returns an array of intents and the confidence level for the given user input.

You can save an array in a custom component, in a user-scoped variable, or as shown in the following snippet, a context variable. In it, there are arrays set for the `person` and `colors` variables.

```
context:
  variables:
    person: "string"
    colors: "string"
...

setPerson:
    component: "System.SetVariable"
    properties:
      variable: "person"
      value:
        - firstName: "Frank"
          lastName: "Normal"
        - firstName: "Grant"
          lastName: "Right"
        - firstName: "Geoff"
          lastName: "Power"
```

```
        - firstName: "Marcelo"
          lastName: "Jump"

...

setColors:
    component: "System.SetVariable"
    properties:
      variable: "colors"
      value:
        - "yellow"
        - "blue"
        - "red"
        - "black"
        - "white"
- "green"
```

These `colors` and `person` arrays are used to illustrate the array operations and in Example: Iterating Arrays.

You can use arrays with different components like the System.Output and System.SetVariable to accomplish different things:

- To create mock data for testing.
- To define data structures that persist beyond user sessions.

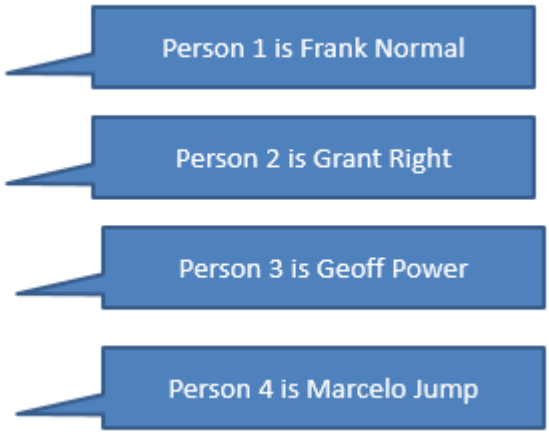
You can define array properties for different components, like System.Output or System.SetVariable (illustrated in the following snippet).

Operator	Example	Output
`size`	`${person.value?size?number}`	`4`—The size (four members) of the `person` array
array index	`${person.value[1].firstName}`	`Grant`—It's the value of the second `firstName` property in the `person` array.
	`${person.value[1].firstName!'unknown'}`	Same as the above, but in this case, the bot outputs unknown if the second `firstName` property has no value.
`first`	`${person.value?first.firstName}`	`Frank`—The first entry of the person array. This operation doesn't use the array index.
`last`	`${person.value?last.firstName}`	`Marcelo`—The final `lastName` value in the person array.

Operator	Example	Output
sort_by	${person.value? sort_by('lastName') [0].firstName}	Marcelo This operator sorts the person array by the lastName property in ascending order. It then prints the value of the corresponding firstName property for final entry in the person array: • Jump, **Marcelo** • Normal, Frank • Power, Geoff • Right, Grant **Note**: Unless you save the sorted array in a variable using System.SetVariable, the data remains sorted for a single request only.
	${person.value? sort_by('lastName')? reverse[0].firstName}	Grant—the values are sorted in descending order: • Right, **Grant** • Power, Geoff • Normal, Frank • Jump, Marcelo
seq_index_of	${colors.value? seq_index_of('red')}	2—The index value for red in the colors array.
seq_last_index_of	${colors.value? seq_last_index_of('red')}	2—The last index value for red in the
join	${colors.value?join(',')}	Returns the colors array as a comma-separated string: yellow, blue, red, black, white, green
seq_contains	${colors.value? seq_contains('red')?	Returns Yes because the array contains red. **Note**: ?seq_contains returns a boolean value. This value is then replaced by a string using the ?string('...'.'...') expression.
sort	${colors.value?sort? join(',')}	Returns the colors array as a comma-separated string in ascending order: black, blue, green, red, white, yellow
reverse	${colors.value?sort? reverse?join(',')}	Returns the colors array as a comma-separated string in descending order: yellow, blue, red, black, white, green

Example: Iterating Arrays

Arrays determine the number of entities in the user input. The following snippet shows how to determine the size of the array held in the `person` variable and then iterate over its elements so that the bot outputs something like:




```
component: "System.CommonResponse"
properties:
  metadata:
    responseItems:
    - type: "text"
      text: "${person?index+1}. ${person.firstName} ${person.lastName}"
      name: "Sorry"
      separateBubbles: true
      iteratorVariable: "person"
  processUserMessage: false
```

> **Note:**
>
> The output described in this code is not sorted (that is, no `sort_by` operation is used).

Built-In FreeMarker Date Operations

The following snippet derives the current date using the FreeMarker special variable reference, `.now` and the built-in `date` operator.

```
PrintToday:
  component: "System.Output"
  properties:
   text: "${.now?date}"
   keepTurn: false
```

The following table lists some of the built-in date operations that you can use to define properties and manipulate entity values.

Operation(s)	Example	Output
`date`	`${.now?date}`	The current date
`time`	`${.now?time}`	The time of day, like 5:46:09 PM
`datetime`	`${.now?datetime}`	Prints current date and time, like Jan 17, 2018 5:36:13 PM.
`long` and `number_to_date`	`${(.now?long + 86400000)?number_to_date }`	Adds 24 hours to the current date. If the call is made on January 17, 2018, FreeMarker outputs January 18, 2018.
`string` (with formatting styles)	`${.now?string.full}`	Converts the current date to string formatted as Wednesday, January 17, 2018 6:35:12 PM UTC.
	`${.now?string.long}`	Converts date to string with the following formatted output: January 17, 20186:36:47 PM UTC.
	`${.now?string.short}`	Converts date to string with the following formatted output: 1/17/18 6:37 PM
	`${.now?string.medium}`	Converts date to string with the following formatted output: Jan 17, 2018 6:38:35.
	`${.now?string.iso}`	Prints the date in the ISO 8601 standard like 2018-01-17T18:54:01.129Z.
`string` (with specified output formats)	`${.now?string['dd.MM.yyyy, HH:mm']}`	Prints the current date in a custom format, like 17.01.2018, 18:58.
	`${.now?string['yyyy']}`	2018
`datetime` (with `string` and formatting style)	`${date_variable?datetime?string.short}`	Converts the date to a string formatted as 1/17/18 6:37 PM. The `datetime` operator enables FreeMarker to tell if the variable holds a date that contains both date and time information. Similarly, you can use the `date` or `time` operators to indicate if the date value contains only the date or only the time, but using `datetime?string` avoids errors.
Converting the entity value to a string using • `date` • `long` • `number_to_date` • formatting styles • custom date formats	`${dateVar.value.date?long?number_to_date?date?string.short}`	Converts the date from the entity extraction to a string formatted as 11/17/18. The date operator tells FreeMarker that the variable only holds a date, not time information. Using this format avoids errors.

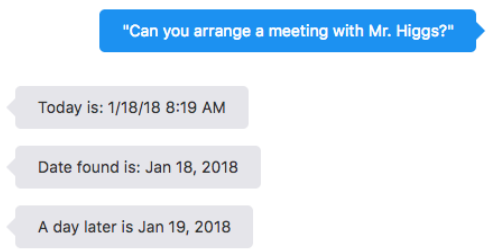
Operation(s)	Example	Output
	`${dateVar.value.date?long?number_to_date?string.medium}`	Converts the date that's derived from entity extraction to a string formatted as Jan 17, 2018. **Note**: All other formats like `full`, `short`, `long` and `iso` work the same with dates derived from entity extraction.
	`${dateVar.value.date?long?number_to_date?string['dd.MM.yyyy']}`	Prints the date in custom format. For example: 17.01.2018, 18:58.
	`${dateVar.value.date?long?number_to_date?string['yyyy']}`	Prints the date derived from entity in a custom format.

Example: Extracting Dates from User Input

The following snippet is from a bot that manages appointments. When a user asks it, *Can you arrange a meeting with Mr. Higgs a day later than tomorrow?*, the bot uses a complex entity, `DATE`, to extract *tomorrow* from the request. It outputs the requested date using `${(theDate.value.date?long + 86400000)?number_to_date}` to add 24 hours (or 86,400,000 milliseconds) to the current date.

OBotML Code	Output

```
context:
  variables:
    iResult: "nlpresult"
    theDate : "DATE"
states:
   intent:
    component: "System.Intent"
    properties:
      variable: "iResult"
      confidenceThreshold: 0.4
    transitions:
      actions:
        unresolvedIntent: "dunno"
        Appointment: "printToday"
printToday:
  component: "System.Output"
  properties:
    text: "Today is: ${.now}"
    keepTurn: true
startAppointement:
  component: "System.SetVariable"
  properties:
    variable: "theDate"
    value: "$
{iResult.value.entityMatches['DATE'][0]}"
printDateFound:
  component: "System.Output"
  properties:
    text: "Date found is: $
{theDate.value.date}"
    keepTurn: true
printDayAfter:
  component: "System.Output"
  properties:
    text: "A day later is $
{(theDate.value.date?long + 86400000)?
number_to_date}"
  transistions:
    return: "done"
```



Example: Setting a Default Date (When No Date Value Is Set)

If the user message doesn't include any date information, your bot can prompt users for the date, or provide a default date, as shown by the following snippet (which augments the dialog flow in the previous example). To perform the latter, your bot needs to check if date variable has been set after the NLP engine extracts entities from the user input.

```
  conditionEquals:
    component: "System.ConditionEquals"
    properties:
      variable: "theDate"
      value: null
    transitions:
      actions:
```

```
      equal: "setDefaultDate"
      notequal: "printDateFound"
```

If no date value has been set, the `System.SetVariable` component defines a default value in a variable and transform it into a string.

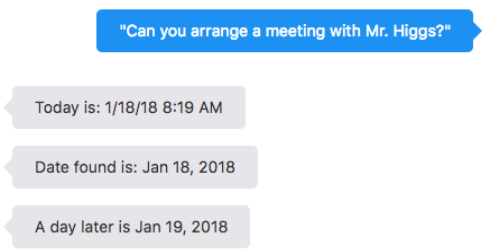
```
setDefaultDate:
  component: "System.SetVariable"
  properties:
   variable: "defaultDateInput"
   value: "${.now?datetime?string.long}"
```

The `System.MatchEntity` component verifies that this value is a date and then sets the`theDATE` variable:

```
matchEntity:
  component: "System.MatchEntity"
  properties:
    sourceVariable: "defaultDateInput"
    variable: "theDate"
  transitions:
    actions:
      match: "printDateFound"
      nomatch: "exit"
```

OBotML	Output

```
context:
  variables:
    iResult: "nlpresult"
    theDate : "DATE"
    #need extra variable for default
date input
    defaultDateInput: "string"
states:
 ...

#try to extract date information from
user sentence
  startAppointement:
    component: "System.SetVariable"
    properties:
      variable: "theDate"
      value: "$
{iResult.value.entityMatches['DATE'][0]}"
#set default date if none found
  conditionEquals:
    component: "System.ConditionEquals"
    properties:
      variable: "theDate"
      value: null
    transitions:
      actions:
        equal: "setDefaultDate"
        notequal: "printDateFound"
  setDefaultDate:
    component: "System.SetVariable"
    properties:
     variable: "defaultDateInput"
     value: "${.now?datetime?
string.long}"
  matchEntity:
    component: "System.MatchEntity"
    properties:
  sourceVariable: "defaultDateInput"
    variable: "theDate"
    transitions:
      actions:
        match: "printDateFound"
        nomatch: "exit"
  printDateFound:
    component: "System.Output"
    properties:
      text: "Date found is:
            ${theDate.value.date?long?
number_to_date?date?string.medium}"
      keepTurn: true
  printDayAfter:
    component: "System.Output"
    properties:
    text: "A day later is $
{(theDate.value.date?long + 86400000)?
number_to_date}"
```



OBotML	Output
`transistions:` `  return: "done"`	

The SDK Helper Methods

Function	Usage
`conversation.payload()`	Retrieves the payload of the current user message. The payload contains the message text and other information, like the user ID.
`conversation.text()`	Accesses the text string.
`conversation.attachment()`	Accesses an attachment message.
`conversation.location()`	Accesses a location message.
`conversation.postback()`	Accesses a postback message.

Function	Usage

`conversation.transition("action")` and `conversation.transition()`	Directs the Dialog Engine to the next state in the dialog flow. The custom component can influence the navigation by returning an action string that you've mapped to state in the dialog flow.
	• Component-controlled navigation (`conversation.transition("action")`)—To set the target state, pass a string argument that matches one of the `supportedActions` strings in the component module's `metatdata` function, like `nameFound` and `nameNotFound` in the following snippet:
	`metadata: () => ({ "name": "helloWorld", "properties": { "properties": { "name": { "type": "string", "required": false" } }, "supportedActions": ["nameFound", "nameNotFound"] }),`
	• Dialog flow-controlled navigation (`conversation.transition()`)—You can call this function with no arguments when the component module has no `supportedActions` definition (and therefore, no arguments to pass). In this case, the dialog flow definition sets the transition, not the component. For example, depending on the dialog flow definition, the Dialog Engine might move to the next state in the flow after `transition` method executes (`transitions: {}`) or on to a specific state:
	`transitions: next: "newState"`
	The dialog flow will also determine the transition when the component module has a `supportedActions` definition, but the function itself has no arguments.

`conversation.channelType()`	Allows you to determine the messaging channel.

Function	Usage
`conversation.keepTurn(boolean)`	Enables your bot to retain control of the conversation. `keepTurn` essentially decides who provides input or a response: the bot or its user. So before you call `done`, you can indicate who goes next by calling either `conversation.keepTurn(true)` or `convesationkeepTurn(false)`. • `conversation.keepTurn(true)`—Set to `true` to allow the bot to control the conversation. This is essentially the bot (through the component) asserting "It's still my turn to speak." Use this setting when the component doesn't require user input or when it needs to send multiple replies in quick succession while suppressing user input. • `conversation.keepTurn(false)`—Set to `false` (the default) to enable the user to reply. This setting essentially hands control back to the user until the next reply from the component. It enables a typical back-and-forth conversation.
`conversation.reply({text: "..."})`	Returns the response from the messaging client. This response can be a simple text message, or a a more complex response with a rich UI that uses the functions of the `MessageModel` class in the Custom Component SDK. This function enables you to build more a complex response, such as a scrolling carousel on Facebook. For this type of response, you need to structure the JSON payload appropriately. **Important:** You must call `done()` to send the response, regardless of the number of calls made to `conversation.reply`.
`conversation.properties()`	Provides access to the component input properties (`conversation.properties().accountType`).
`conversation.error`	Indicates that there was an error in the processing.
`conversation.botId()`	Returns the ID of the bot that sent the request.
`conversation.platformVersion()`	Returns the version of the message platform (such as `Facebook 1.0`).

Function	Usage
`conversation.text()`	Provides access to the NLP text message that triggered the invocation of the intent and the component.
`conversation.variable("name",value)`	Provides read or write access to variables defined in the current flow. This function takes the following arguments: • `variable(name)`—Reads the `name` variable and returns its value. • `variable(name, value)`—Writes the value of the `value` variable to the `name` variable. Only enclose the value in quotes when it's a string. This function also creates a variable at runtime, one that can be used to track the state of component. You can use this when component needs to track its internal state because it doesn't transition to the next state in the dialog flow.
`conversation.nlpResult()`	Returns an `NLPResult` helper object for `nlpresult` variables. For example, you can find the value of an entity that was extracted from the user input by calling `conversation.nlpResult.entityMatches(entity name)`. You can use this value to update an entity type variable.
`conversation.request()`	Accesses the JSON object body that's sent by the bot. Use this function to parse the payload for any information that's not directly exposed by one of the SDK functions.
`conversation.response()`	Grants access to the HTTP response payload (a JSON object) that's sent back to the bot when you call `done()`.

Navigation with keepTurn and transition

Use different combinations of the `keepTurn` and `transition` functions to define how the conversation continues once the component has finished processing.

```
invoke: (conversation, done) ==> {
   ...
   conversation.keepTurn(true);
   conversation.transition ("success");
   done();
    }
```

Use Case	Values Set for keepTurn and transition
A custom component's reply that doesn't require any user interaction.	• Set `keepTurn` to `true`: `conversation.keepTurn(true)`. • Set `transition` with a `supportedActions` string(`conversation.transition("success")`) or with no arguments (`conversation.transition()`). For example, a custom component updates a context variable with a list of values that is then displayed by a `System.List` component that's defined for the next state in the dialog flow definition. `invoke: (conversation, done) => { const listVariableName = conversation.properties().variableName; ... // write list of options to a context variable conversation.variable(listVariableName, list); //navigate to next state. No user interaction. conversation.keepTurn(true); conversation.transition(); done(); }`

Note:

When component doesn't transition to the next state, it needs to track its own state by creating a runtime variable using the `conversation.variable ("name", variable)` method.

Use Case	Values Set for keepTurn and transition

A sequential user conversation in which the user provides input, the bot replies, and so on.	• Set `keepTurn` to `false`.
	• Set `transition` with a `supportActions` string
	For example:
	`conversation.keepTurn(false);`
	`conversation.transition("success");`

The bot to passes control back to the user without navigating to the next dialog state. This allows the component to process the user input. Here are a couple of examples:
• A component passes the user input to query a backend search engine. If the chatbot can only accommodate a single result, but the query instead returns multiple hits, the component can then prompt the user more input to filter the results. In this case, the custom component continues to handle the user input; it holds the conversation until the search engine returns a single hit. When the backend system is satisfied, the component calls `conversation.transition()` to move on to the next state as defined in the dialog flow definition.
• A questionnaire, wherein a custom component handles all of the questions and only transitions to the next state when each of them gets answered.
	• Do not call `transition`.
	• Set `keepTurn` to `false`.
	For example:
	`conversation.reply("text");`
	`conversation.keepTurn(false);`
	`done();`

The custom component goes into a loop, which can't be stopped by user input. For example, a component pings a remote service for the status of an order until the status is returned as `accepted` or when the component times out. If the `accepted` status is not returned after the fifth ping, then the component transitions to a `failedOrder` state, which is defined in the dialog flow.	• Do not call `transition`.
	• Set `keepTurn` to `true`: `conversation.keepTurn(true)`.
	For example:
	`conversation.reply("text");`
	`conversation.keepTurn(true);`
	`done();`

Note:

Always call `keepTurn` after `reply` and not before, because `reply` implicitly sets `keepTurn` to `false`.

The Custom Component Payload

Taking a Look at the Metadata Retrieval

The response payload for the GET endpoint is made up of properties that are required to call the component along the name of the component itself. You can shape this payload by defining the state machine transitions (that is, the possible actions returned by this component). As illustrated by the array in this example, you can add as many components as you need.

```
{
    "version": "1.0",
    "components": [{
            "name": "AgeChecker",
            "properties": {
                "minAge": {
                    "type": "integer",
                    "required": true
                }
            },
            "supportedActions": [
                "success",
                "fail"
            ]
        }, {
            "name": "PizzaBaker",
            "properties": {
                "size": {
                    "type": "string",
                    "required": true
                },
                "crust": {
                    "type": "string",
                    "required": true
                },
                "type": {
                    "type": "string",
                    "required": true
                }
            },
            "supportedActions": [
                "pizzaReady",
                "fail"
            ]
        }
    ]
}{
```

Taking a Look at the Invocation Request and Response Payloads

For the POST, the request contains the bot's GUID, the platform version of Bots, and a `context` definition for the dialog flow variables, the state's properties, the original message, the channel that delivered it, and the tenant.

```
{
  "botId" : "963B57F7-CFE6-439D-BB39-2E342AD4EC92",
  "platformVersion": "1.0",
  "context" : {
    "variables" : {
      "location" : {
        "type" : "string",
        "value" : null,
        "entity" : false
      },
      "system.errorAction" : {
        "type" : "string",
        "value" : null,
        "entity" : false
      },
      "system.errorState" : {
        "type" : "string",
        "value" : null,
        "entity" : false
      }
    }
  },
  "properties" : {
    "location" : "San Francisco"
  },
  "message" : {
    "payload" : {
      "text" : "What stores are near me?"
    },
    "retryCount" : 0,
    "channel" : {
      "botId" : "963B57F7-CFE6-439D-BB39-2E342AD4EC92",
      "sessionId" : "1769637",
      "type" : "test",
      "userId" : "1769637",
      "channelId" : "18DE5AEF-65A4-461B-B14D-0A4BCF1D6F71"
    },
    "tenantId" : "DefaultOliveTenant",
    "createdOn": "2017-01-20T00:23:53.593Z",
    "id" : "8c9556b2-9930-4228-985e-f972bfcfe26d"
  }
}
```

The POST response payload also includes the context definition for all of the variable values, including those that have been mutated by the component. The response payload can also control the routing through properties like `exit`, `done`, and `error`. You don’t have to parse the JSON if you use our SDK. Instead, you just need to set the variables. See The SDK Helper Methods.

```
{
  "platformVersion": "1.0",
  "context" : {
    "variables" : {
      "location" : {
        "type" : "string",
        "value" : "San Francisco",
        "entity" : false
      },
      "system.errorAction" : {
        "type" : "string",
        "value" : null,
        "entity" : false
      },
      "system.errorState" : {
        "type" : "string",
        "value" : null,
        "entity" : false
      }
    }
  },
  "action" : null,
  "exit" : false,
  "done" : false,
  "error" : false,
  "modifyContext" : false
}
```