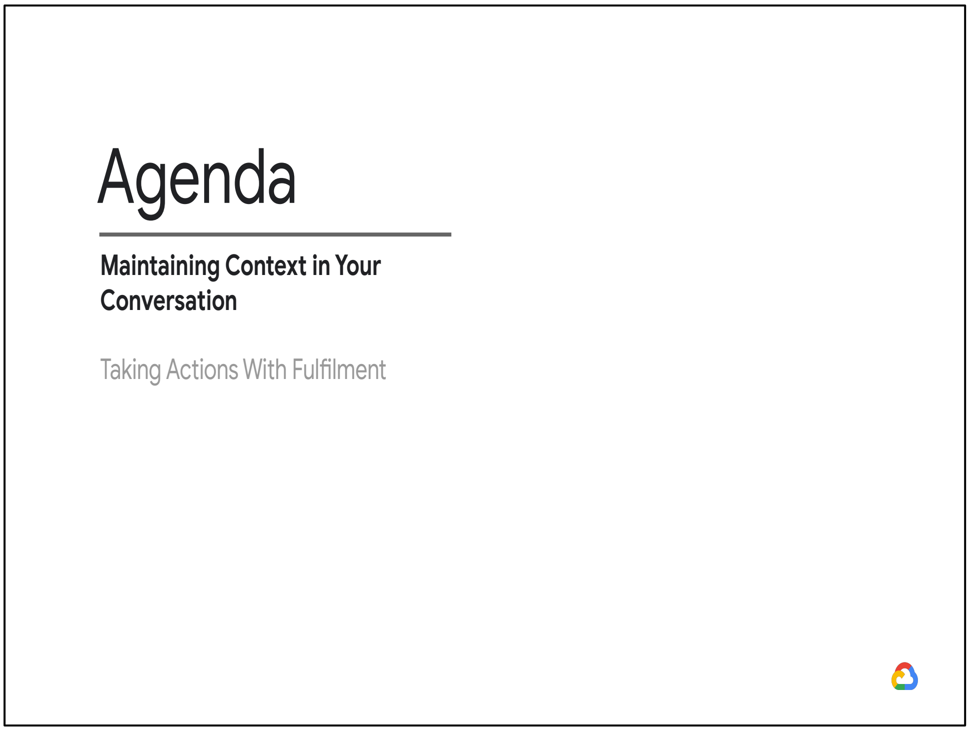
Google Cloud

Maintaining Context and Taking Actions

In this module, we continue to build our conversational agent for the pizza ordering example. We will start by adding context so the agent can keep the conversation going.
Next, we will talk about how to write code to perform actions. In the case of our pizza ordering example, this could mean storing pizza orders into a database.
So, let’s start with context.

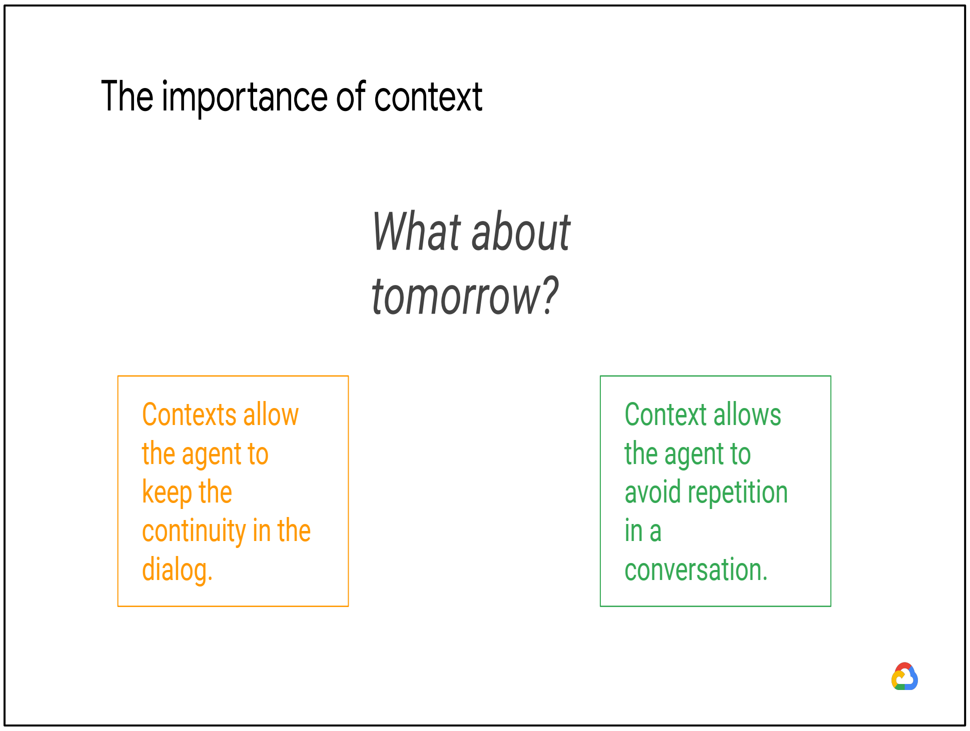


Agenda

Maintaining Context in Your Conversation

Taking Actions With Fulfilment

We will start this module by discussing the importance of context in a conversation.



The importance of context

What about tomorrow?

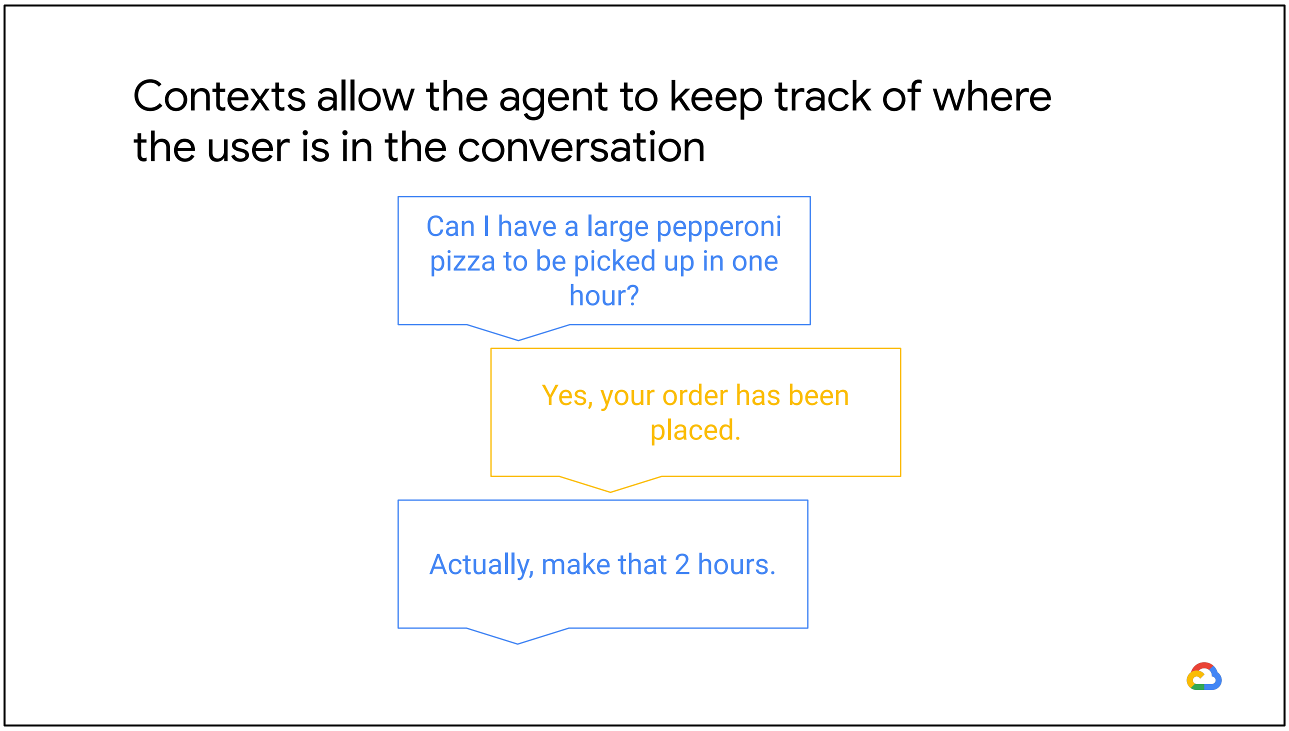
Contexts allow the agent to keep the continuity in the dialog.

Context allows the agent to avoid repetition in a conversation.

Have you ever experienced a situation where you walk up to a group at the moment for a strange comment and you catch yourself trying to make sense of what they might be talking about?

If a friend comes to you and says: "what about tomorrow?" how would you react? You would probably ask them: "what are you referring to?" and that reaction comes from your lack of context.

The same happens to agents. Knowing in which context the user is, allows for less repetition. One example is if I ask: what should we have for lunch today? The other party responds, say, sandwich. Then if I ask: "what about dinner?", I would expect that the other person knows I am referring to **what** we are going to eat, and not what time we should be heading out for dinner, for example. This awareness can be provided to the agent through contexts.



Contexts allow the agent to keep track of where the user is in the conversation

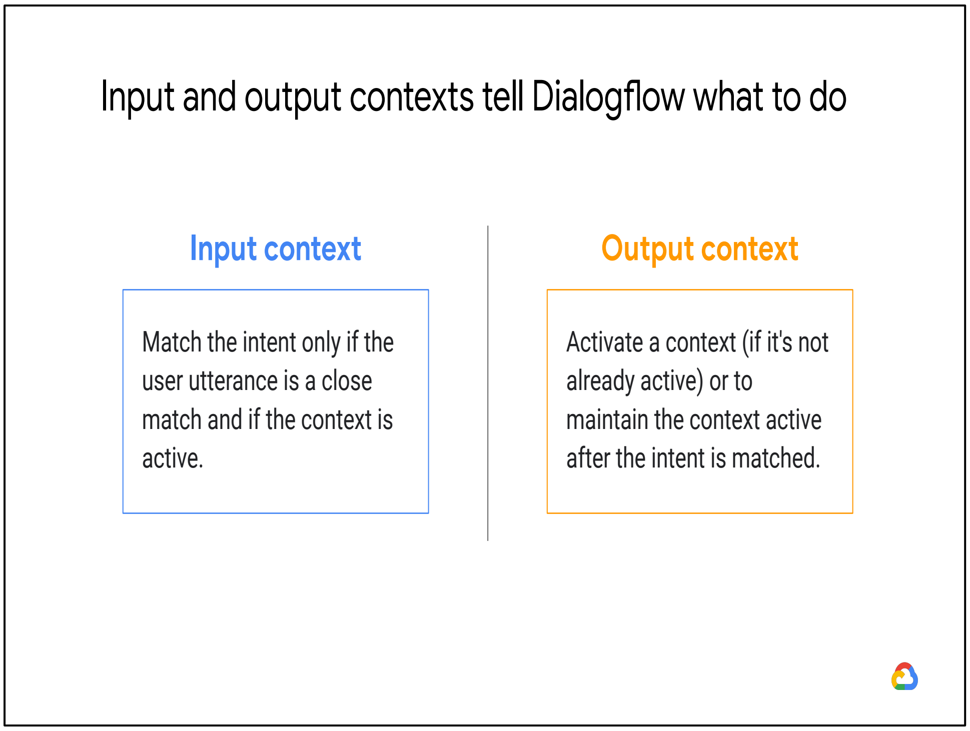
Can I have a large pepperoni pizza to be picked up in one hour?

Yes, your order has been placed.

Actually, make that 2 hours.

Contexts allow the agent to keep track of where the user is at in the conversation. In the context of Dialogflow, they are also a means for an application to store and access variables mentioned in the dialog.

In this example, if the sentence: "Actually, make that 2 hours" is said out of context, there is no way the agent can know that the user is referring to the pick up time for the pizza order.



Input and output contexts tell Dialogflow what to do

Input context

Match the intent only if the user utterance is a close match and if the context is active.

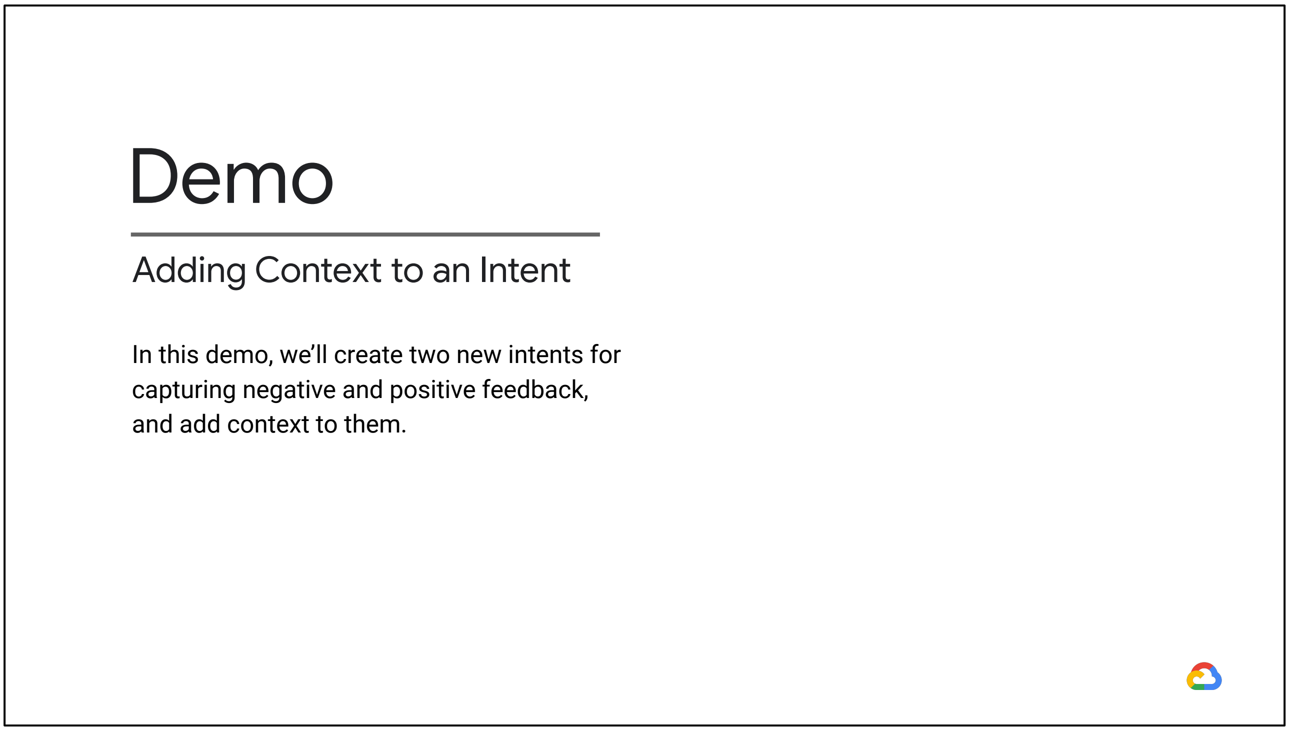
Output context

Activate a context (if it's not already active) or to maintain the context active after the intent is matched.

Context allows the agent to control conversation flows. This can be done by defining specific states that the dialog must be in for an intent to match. For example, Dialogflow matches an intent if what the user says resembles the provided training phrases for that intent. However, when you apply contexts to an intent, Dialogflow will only consider that intent for matching if the context is active. This is one of the purposes of the input context.
If **an input context** is applied to an intent, it tells Dialogflow to match the intent only if the user utterance is a close match and if the context is active.

Another type of context is **output contexts**. When applied to an intent, an output context tells Dialogflow to activate a context (if it's not already active) or to maintain the context active after the intent is matched. This allows the agent to control the conversation flow.

Let's see an example of how to add context to an intent.



Demo

Adding Context to an Intent

In this demo, we'll create two new intents for capturing negative and positive feedback, and add context to them.

<u>Switch to demo. (Agent: agent-6)</u>

Here we are going to create two new intents for capturing negative and positive feedback, and add context to them. Let's suppose the agent suggests a drink to go with the pizza order and it wants to capture if the answer from the user is positive or negative. This will allow the agent to match the response to the question: *would you like to have a drink with your pizza?* And associate it with the right intent. So, first, let's add the upselling question to the *order.pizza* response. Click Save. This step allows the agent to ask the question, but it doesn't recognize the response. Let's test it.

To create a context, let's add an output context to our order.pizza intent. Let's call it *pizza-upsell*. Notice that a number is added to the context name. This number represents the lifespan of the context, in other words, it shows for how many interactions the context will be active. You can modify the lifespan, if you wish. Click save.

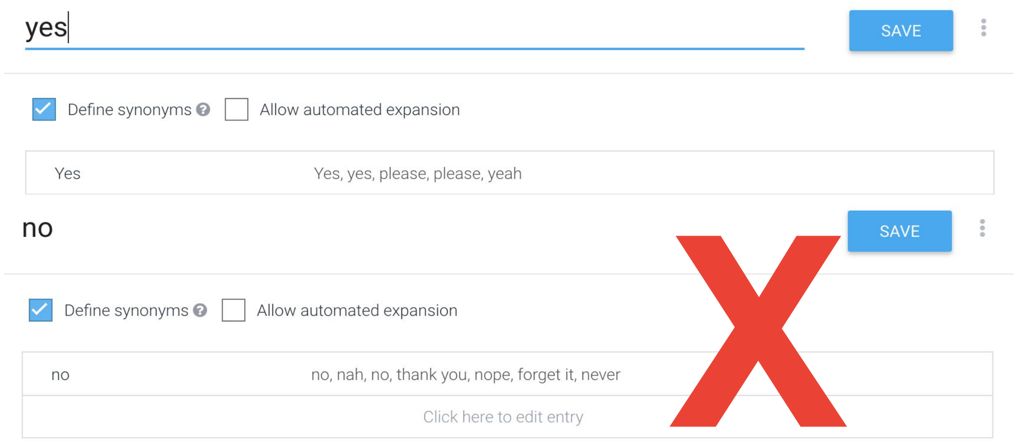
Now let's create an intent for capturing a positive response. Let's call it *order.pizza.upsell.drink - yes*. Before we add training phrases, let's set up the context. Add the *pizza-upsell* context to the input section. If you want the context to still be active leaving this intent, then you need to add *pizza-upsell* to the output context as well. The last step is to add training phrases to recognize a positive response. Some examples are yes, of course, sure, I like that. Can you think of a couple more? Now you can pause this video and finish adding the examples you can think of. Once you're done, click save and take the same set of steps for a negative response (to cover the cases where the user won't want to order a drink with their pizza).

<u>Complete agent from demo is agent-7</u>

Let's test our context and see if it is working. Starting from the beginning, let's order a pizza. Then the agent offers a drink. Let's say why not? There, it detects it as a positive response and replies that a combo will be ordered.

But wait, is the order actually going anywhere? Not yet. We haven't implemented the fulfilment of the action which we’ll talk about later.

Best practices: When and how to use context



yes

☑ Define synonyms ☐ Allow automated expansion

Yes	Yes, yes, please, please, yeah

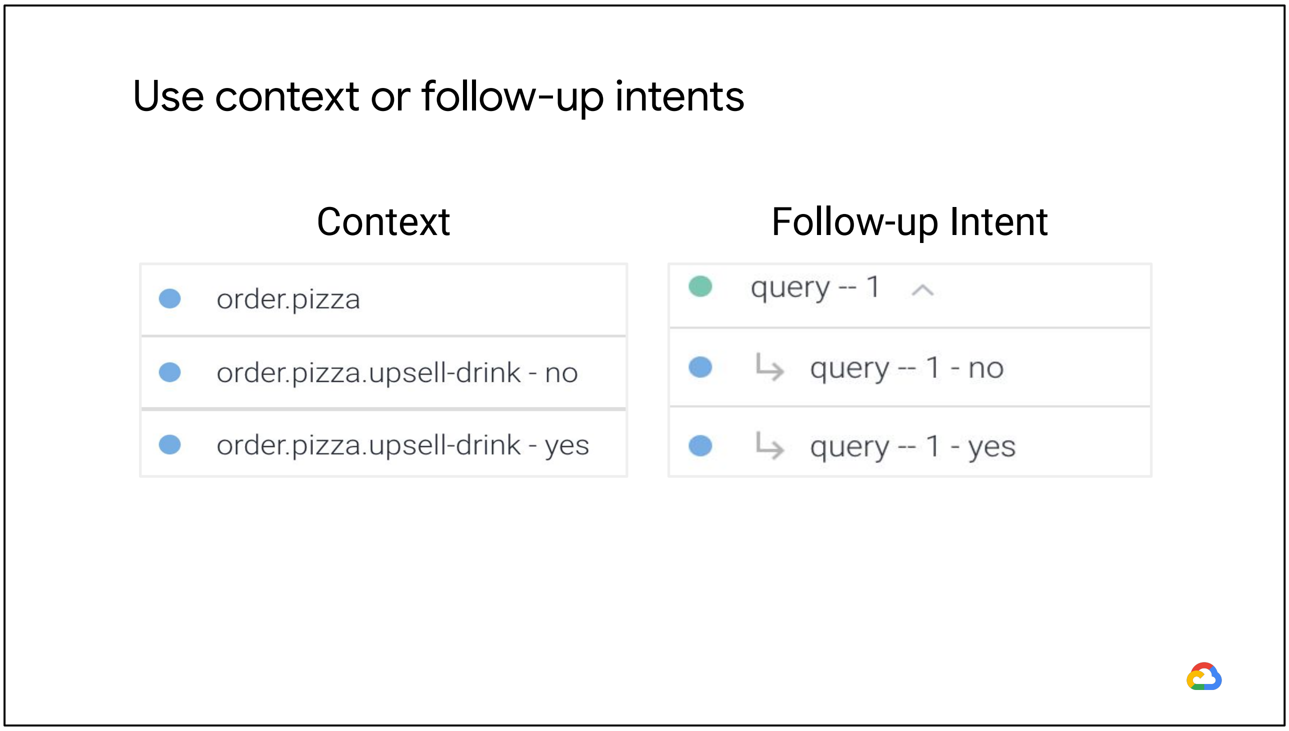
no

☑ Define synonyms ☐ Allow automated expansion

no	no, nah, no, thank you, nope, forget it, never
	Click here to edit entry

Now we will go over some of the best practices that make developing agents easier and more scalable. We will cover some of them and would highly encourage that you take a look at the reference page for this module where we have additional links to the Dialogflow documentation.

As we saw in the previous video, sometimes we might want to collect positive or negative feedback from the user. When gathering boolean feedback, there is a better alternative to creating entities that match yes and no.



Use context or follow-up intents

Context

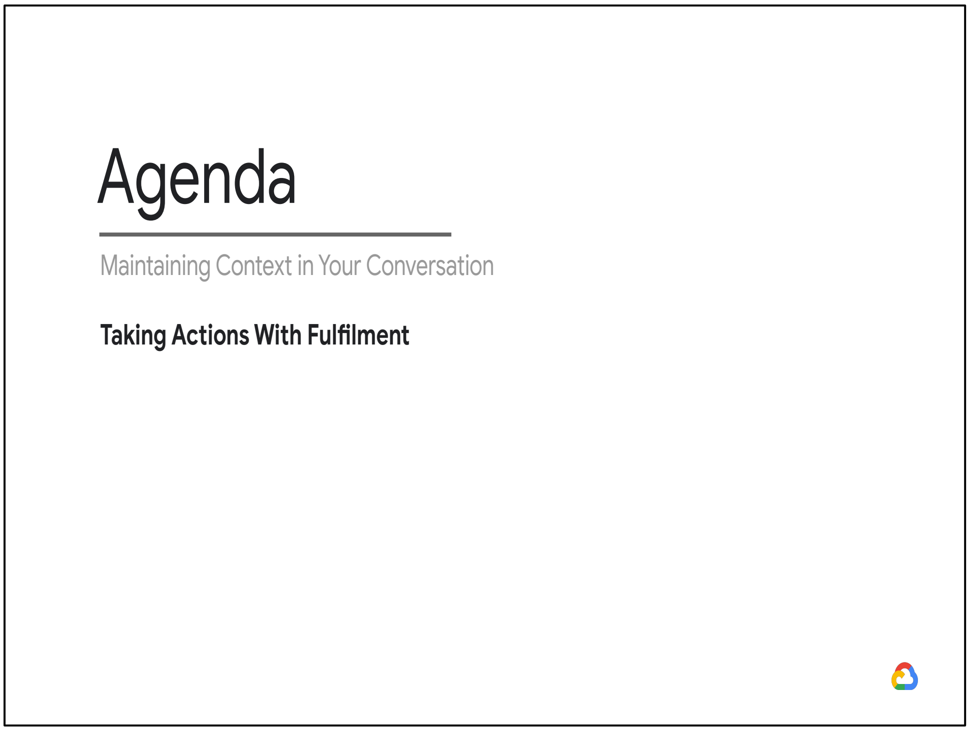
- ● order.pizza
- ● order.pizza.upsell-drink - no
- ● order.pizza.upsell-drink - yes

Follow-up Intent

- ● query -- 1
  - ● ↳ query -- 1 - no
  - ● ↳ query -- 1 - yes

Instead, use context or follow-up intents instead. These intents can, in turn, have the training phrases reflect the negative or positive expressions, accordingly.

Follow-up intents will carry the context automatically when created.

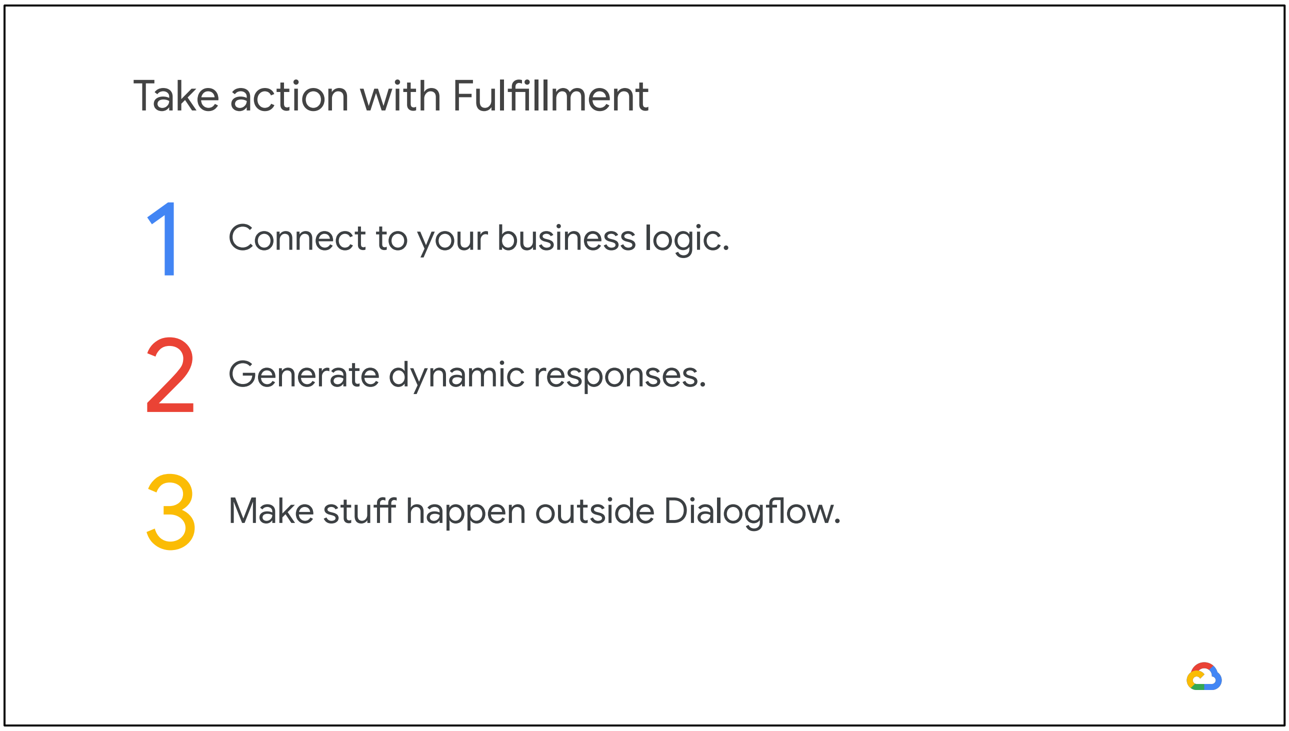


Agenda

Maintaining Context in Your Conversation

Taking Actions With Fulfilment

Up to this point you learned how to create flows for a conversation between the agent and the user. You've learned how to create intents, annotate them with relevant entities and how to keep the context of the conversation. Now, what if you want to allow the agent to do more than that?



Take action with Fulfillment

1 Connect to your business logic.

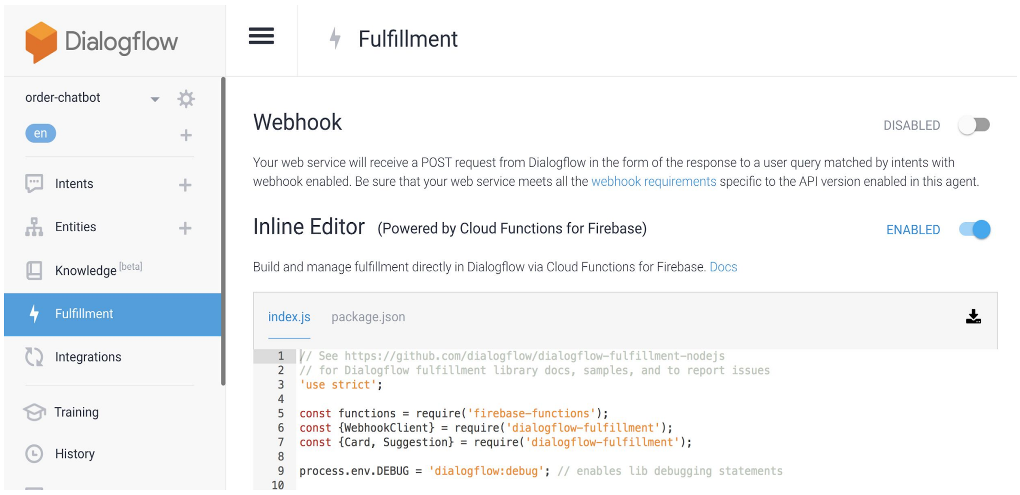
2 Generate dynamic responses.

3 Make stuff happen outside Dialogflow.

What if you decide to persist the pizza order to a database? Or to allow the agent to provide prices of items stored in the database? You can achieve that with Fulfillment.

Fulfillment is the mapping of an action using code that's deployed outside of Dialogflow. This allows your agent to execute external business logic on an intent-by-intent basis.

Creating Fulfillment



Webhook DISABLED

Your web service will receive a POST request from Dialogflow in the form of the response to a user query matched by intents with webhook enabled. Be sure that your web service meets all the webhook requirements specific to the API version enabled in this agent.

Inline Editor (Powered by Cloud Functions for Firebase) ENABLED

Build and manage fulfillment directly in Dialogflow via Cloud Functions for Firebase. Docs

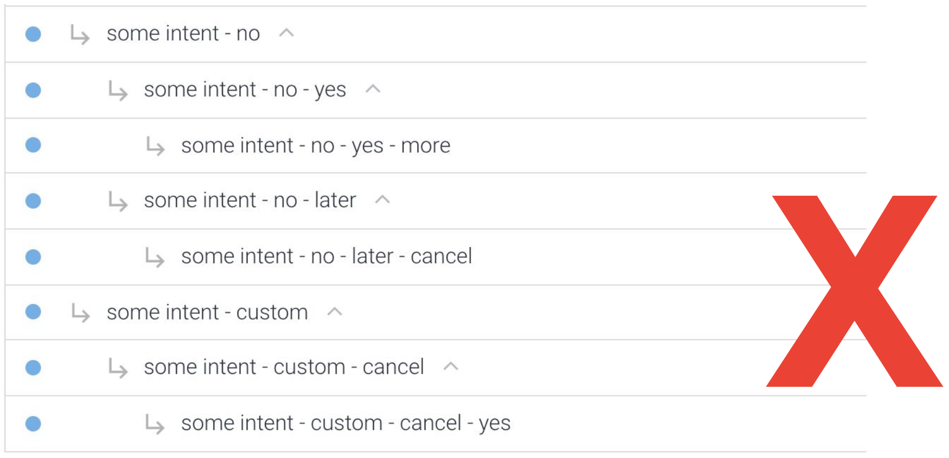
index.js package.json

```
// See https://github.com/dialogflow/dialogflow-fulfillment-nodejs
// for Dialogflow fulfillment library docs, samples, and to report issues
'use strict';

const functions = require('firebase-functions');
const {WebhookClient} = require('dialogflow-fulfillment');
const {Card, Suggestion} = require('dialogflow-fulfillment');

process.env.DEBUG = 'dialogflow:debug'; // enables lib debugging statements

```

Upon the detection of an intent which corresponds to an action that needs to be taken, the agent should be enabled to "talk" to external systems in order for the actions to be fulfilled.

We can write code for this communication within the console or using an IDE of choice. For this example, we will use the inline editor for Cloud Functions under the Fulfillment menu option. In a later module, we will cover how to use a webhook to achieve fulfilment.

Don't use follow-up intents to capture missing entities


some intent - no
some intent - no - yes
some intent - no - yes - more
some intent - no - later
some intent - no - later - cancel
some intent - custom
some intent - custom - cancel
some intent - custom - cancel - yes

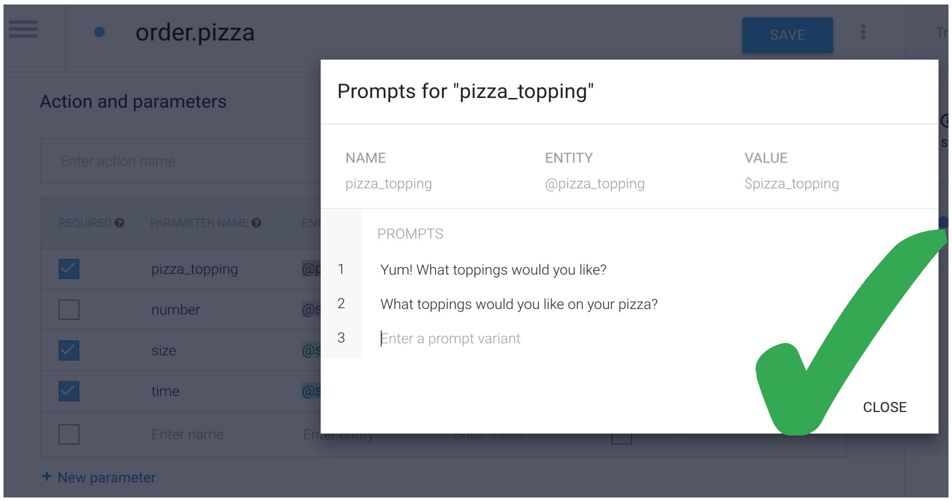
Before I show you a demo of how to set up fulfilment, lets address one more thing. Make sure all of the needed pieces of information are collected.

For placing an order for a pizza, for example, let's say that the backend system needs to know at least three pieces of information: the size of the pizza, the toppings and the pick up time. These would be three different entities that we need to identify and extract from the customer request.

If the customer says: can I have a pizza? We need to make the agent ask for the additional information needed before the order can be placed (or sent to the backend system responsible for placing orders).

How can we collect these missing pieces of information? You might feel inclined to use follow up intents, or context, to do so. However, there is a more efficient way to do it. It is through the slot filling functionality.

Use slot filling instead


order.pizza

SAVE

Action and parameters

Enter action name

REQUIRED	PARAMETER NAME	ENTITY
✓	pizza_topping	@p
☐	number	@s
✓	size	@s
✓	time	@s
☐	Enter name	Enter entity

+ New parameter

Prompts for "pizza_topping"

NAME	ENTITY	VALUE
pizza_topping	@pizza_topping	$pizza_topping

PROMPTS

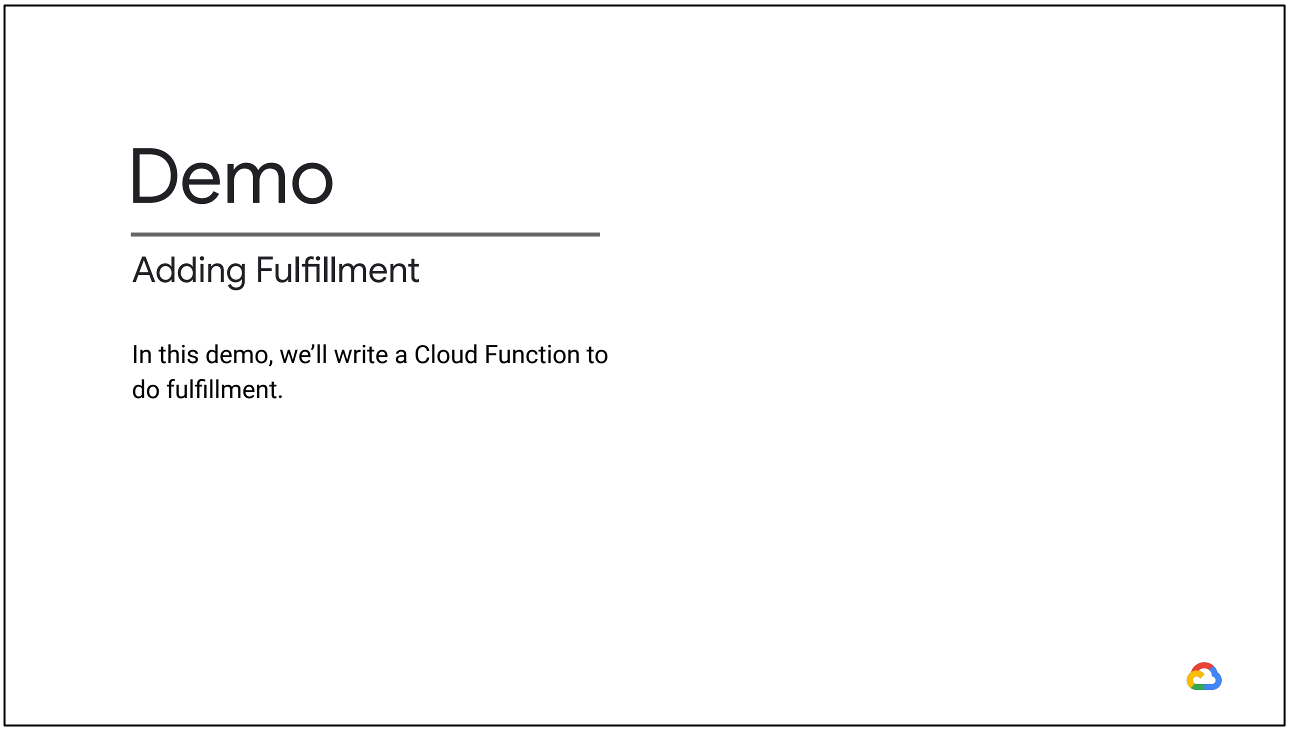
1 Yum! What toppings would you like?

2 What toppings would you like on your pizza?

3 Enter a prompt variant

CLOSE

To use slot-filling, you set the needed parameter values corresponding to the entities in the request to be required. If users omit one or more of the parameters in their response, your agent prompts them to provide values for each missing parameter. Let's see an example of how to use slot filling and fulfill the pizza order request.



Demo

Adding Fulfillment

In this demo, we'll write a Cloud Function to do fulfillment.

On GCP:
1 - Create an entity with a [default] namespace and description: ordem_item
2 - Add the properties:
Item_name - String
Order_time - String
Pickup_time - String
Size - String
Topping - array

On Dialogflow Console:
Do slot filling on size and toppings:
1 - toppings: Yum! What toppings would you like?
2 - Size: Would you like a slice or the whole pie?

Fulfilment (Inline editor)
1 - Enable Cloud functions - mention that webhooks are another way to communicate to backend systems.
2 - Add "@google-cloud/datastore": "^0.8.0" to the list of dependencies. We are adding this package because our fulfilment will write the order to Cloud Datastore, which is a NoSQL storage service in GCP.
3 - Click deploy
4 - Go to Cloud Function and see the deployed function (named: dialogflowFirebaseFulfillment)

5 - Go back to the Dialogflow console and click on the code section
6 - Below the code

```
process.env.DEBUG = 'dialogflow:debug';
```

Add the code:

```
const Datastore = require('@google-cloud/datastore');
const datastore = new Datastore({
  projectId: '[YOUR CHATBOT ID]'
});
```

7 - Get the project id
8 - Write (copy and paste) and explain what the order.pizza function is doing
9 - Add the intent mapping
10 - Deploy it

<u>Complete agent from demo is agent-9</u>

Let's test our fulfillment code. Let's order a pizza.
Let's see if our newest order was added to Cloud Data Store.

Lab

Lab 1,Part 2

cloud.google.com