# Event Driven Dialogs

## Introduction

One of the challenges when building bots is how to create a bot that can deliver on the promise of natural language to be more natural and expressive. Most bots today take a guided or flow dialog approach where the bot asks a question until it gets an answer or gives up and then moves to the next step. This has the advantage of being easy to create and easy to use since it is clear at each step what the bot is asking. The disadvantage is that the interactions are not particularly natural, and it struggles to deal with all the different conversational paths. Composer (and the underlying Adaptive Dialogs) makes it very easy to create guided (or flow) dialogs, but it also supports building a more robust and flexible dialog using *event-driven dialogs*.

An event-driven dialog is one where dialog is driven by events like “this input is ambiguous” or “what prompt do you want to do next?” together with the state of memory rather than “this is the next step”. Just like events in UI where interactions can be with anything in the UI conversational events allow interactions that with anything the dialog understands. Flow dialogs in Composer have some first steps towards event driven dialogs in that they support triggering on an intent. The issue is that every intent and its interaction with the rest of the guided dialog needs to be handled explicitly. If you are in a choice dialog you can detect an intent to switch to a different task, but it would take a lot of work to handle an intent to correct a previous entry or answer to an upcoming question—and even worse you might want to do both on the same utterance. To handle these cases, you must explicitly manage keeping track of what you have done so far and need to go back to in every part of your bot. The result of this complexity is that most bots do not allow this kind of behavior.

Event driven dialogs are driven by triggers with actions that can be run when an event is fired with memory in a certain state. They ask the bot developer to switch from thinking of conversation as a flow chart to conversation emerging from the dynamic combination of small triggers like “if I don’t have a value for the Bread property then here is how to ask for it”, or “if you don’t know whether a bread entity is ‘whole wheat’ or ‘multigrain wheat’ ask the user to pick”. The advantage of using triggers is that you cover many different conversational paths without having to be explicit about each path. Those triggers apply whether someone started off with “I want a sandwich” or “I want a ham and swiss sandwich”, or “I want a ham and swiss on wheat sandwich” or “wheat”. In each case either bread has not been specified or the entity is ambiguous, and the triggers potentially apply. If at every step the user could answer any question, correct previous answers and you needed to deal with ambiguous answers modeling it with a flow chart would have thousands and thousands of nodes, but you can get the same behavior from a handful of triggers.

As you will see, event driven dialogs are built using modular triggers that are capable of handling at every step in a dialog:

* Explicit utterances like: “order a roast beef and rye sandwich”
* Implicit context like: “2” in response to “How many sandwiches do you want?”
* Ambiguity: “Is ‘wheat’ your name or the bread you wanted?”
* Asking for multiple responses together with status: “What kinds of toppings and cheese do you want on your ham sandwich?”
* Conversion and error checking. “You can only order between 1 and 5 sandwiches”

The subsequent sections describe triggers, the underlying event and memory capabilities and how those are used in form generation.

## Triggers & Selectors

Adaptive dialogs support an explicit list of actions for modeling flows and a list of triggers for handling events. An event driven adaptive dialog typically only has triggers. All triggers have four parts: condition, priority, runOnce and a set of actions. While Composer or the SDK have many kinds of triggers, they are just wrappers around that core to make it easier to author i.e., the “OnIntentTrigger” adds an intent property that automatically gets added to the condition.

Triggers are selected by a *selector* which is responsible for choosing the next trigger to execute. The default selector is the *MostSpecificSelector* with a *PrioritySelector* as a secondary selector and that is the behavior outlined below. There are other selectors that you can configure like the *RandomSelector* which will randomly pick one of the matching triggers which will change the behavior described below, but that is a topic beyond this doc.

### Condition & Priority

A trigger condition is an adaptive expression that is evaluated against memory. If the condition is true and the priority is >= 0 then the trigger is a candidate to be selected. The *MostSpecificSelector* will pick the most specific condition that is true. For example, given condition1: *!$Bread* (the *Bread* property has no value) and condition2: *!$Bread && !$Meat* (both the *Bread* property and the *Meat* property have no value) only condition2 will be considered a possibility. This is because all situations with both bread and meat missing will always have bread missing, but not all situations with bread. missing will have meat missing as well. Condition2 is more specific than condition1.

This is a very powerful capability. You could start with the trigger for missing bread and later you can add a trigger to handle the more specific situation of missing both *Bread* and *Meat* without changing the missing *Bread* trigger. This is exactly the behavior you want, i.e. a general handler and in more specific situations you want to override it.

This capability is great for when one condition is more specific than another, but you often end up with many possible triggers. For example if you have *!$Bread->AskForBread* and *!$Cheese->AskForCheese* and you don’t have either *Bread* or *Cheese* then both triggers are possible. The good news is that since each trigger was written from the viewpoint of “in this situation this is a reasonable thing to do” asking either question is OK. (And if you change the secondary selectorto *RandomSelector,* one of them will be picked randomly.) With the default *FirstSelector*, the priority field is used to decide which one of the most specific triggers should be selected. The priority field is an adaptive expression to compute the relative priority of a trigger. Anything less than 0 will be sn an adaptive expression and if the value is less than zero the trigger is filtered out and otherwise the minimum value will be selected. If you do not specify the priority on a trigger than a default priority will be defined based on the order that triggers are found in the list of triggers.

### RunOnce

The *RunOnce* flag indicates that the actions on a trigger can only be run once for each unique set of memory references in the condition. Imagine that when you wanted to call a weather REST API once you had a city and a time i.e., the condition was something like *$City && $Time*. This condition would still be true after you called the REST API so the trigger would run forever. To fix this yourself you would have to add to the condition a flag like *$WeatherRan* and in your actions set the flag which would make the condition false after you executed the trigger actions. The problem is that now the rule would not ever run again unless you reset *$WeatherRan* whenever you changed the value of *$City* or *$Time.* This would work but would be painful to create and get right. Luckily the *RunOnce* flag does this for you. When you set *RunOnce* on a trigger the runtime adds an invisible test to the condition to see if any of the values in the condition have changed since the trigger was last executed. With this mechanism the trigger will run at most once for each unique set of values. (At most because other triggers might have been more specific or of higher priority and they might have changed memory.)

### Execution

Once a trigger has been selected its actions are executed. The actions can be anything, but usually they are relatively simple like ask a question, validate an entity, and assign to a property or call an API.

## Schemas

For adaptive dialogs to be able to generate events it needs to know what properties are being collected and how they relate to recognized entities. To understand this, it is important to first understand the concepts being used.

* **Property:** The goal of a dialog is to fill in the properties defined by your schema. These are the normal Composer properties with their type defined by JSON schema.
* **Entity:** NLU systems like LUIS can recognize entities in an NLU utterance like datetime, bread types or names. These are the “values” found in an utterance that you ultimately want to use in properties. In simple models, entities and properties are often the same but this does not scale. For example, if I have a flight it is reasonable to have a city entity which can map to an origin or destination property. Sometimes this distinction is explicit in an utterance like “fly from Seattle to Dallas” and sometimes it is implicit like when somebody responds “Seattle” to the question “Where you want to fly from?” We handle that at the dialog management level. It is also useful to be able to introduce validation and mapping between an entity and a property. For example, if you have a property that should be a number in feet, then we can handle both dimension and number entities. For a dimension entity, the generated DM will verify the units are in feet, convert the units to feet if possible or indicate the value is not legal. If your prompt is something like “How long is your bed in feet?” and someone says “6” the LUIS number entity will be mapped to your property assuming that it is in feet.
* **Operation:** An operation describes what you want to do with some combination of entity and property. For example, “help on toppings” would provide help on the “toppings” property and “add mustard on the top” would add mustard to the “TopSauce” property. Traditionally operations have been modeled as intents in LUIS, but for operations we want to support multiple operations in a single utterance so you can do things like “add lettuce and remove pickles” so we model them as entities.

The schema is described using JSON schema with a few additional keywords. The schema describes how to tie the entities in your NLU model to the properties you want your dialog to create. Currently, only simple properties or ones where the schema comes from LUIS entities (like dimension) can be defined. The simplest way to create these schemas is to use form generation as described later. (You can also read this [doc](https://microsoft.sharepoint.com/:w:/t/ConversationalAI785/EZMPNp5PJSBFq_G9Z_QhXO4BYvsNdTJd2HFx1MFRSfZp7Q) to understand the NLU model that form generation creates.) Here for example is a sample schema that has numeric *MinBid, MaxBid* and numeration for *Suit* and a *Name.*

{

    "$schema": "http://json-schema.org/draft-07/schema",

    "type": "object",

    "properties": {

        "MinBid": {

            "type": "number",

            "minimum": 0,

            "maximum": 10,

            "$entities": [

                "number"

            ]

        },

        "MaxBid": {

            "type": "number",

            "minimum": 0,

            "maximum": 10,

            "$entities": [

                "number"

            ]

        },

        "Suit": {

            "type": "string",

            "enum": [

                "clubs",

                "hearts",

                "diamonds",

                "spades",

                "NoTrump"

            ],

            "$entities": [

                "SuitEntity"

            ]

        },

        "Name": {

            "type": "string",

            "$entities": [

                "personName",

                "utterance"

            ]

        }

    },

    "required": [

        "Name",

        "MaxBid",

        "Suit"

    ],

    "optional": [

        "MinBid"

    ],

    "$expectedOnly": [

        "utterance",

        "personName"

    ],

    "$operations": [

        "Add()",

        "Remove()",

        "Clear()",

        "Show()",

        "Help()",

        "Change()",

        "Skip()",

        "Cancel()"

    ],

    "$defaultOperation": {

        "": {

            "": "Add()"

        }

    },

    "$requiresValue": [

        "Add()",

        "Remove()"

    ],

    "$public": [

        "MinBid",

        "MaxBid",

        "Suit",

        "Name"

    ]

}

This file is largely a JSON schema file with a few extra keywords including:

* **$entities:** For each property, this lists the entities that can be mapped into that property. For example, *MinBid/MaxBid* can map a number entity into their properties. *Name* can either handle a *personname* entity or the special *utterance* entity which corresponds to the whole user response.
* **$expectedOnly:** Some entities are recognized so often that it is not worth asking the user to choose between possible choices unless they are expected by the *Ask* action. Here *utterance* and *personName* are in that category.
* **$operations:** This defines the entities that should be interpreted as operations rather than values.
* **$requiresValue:** Are operations that must have a value to operate on. For example without a specific value *Add()* or *Remove()* do not make sense.
* **$public:** Document that properties that are exported from this dialog. Here that is all the properties, but when you have a full dialog you may have other properties for confirmation, etc. that need to map from entities but that are part of the dialog management itself.
* **$defaultOperation:** Defines the default operation for each property. In this case if someone does not specify an operation it is assumed the operation should be *Add().*

## Triggers and Actions

Now that we understand triggers, selectors, and schemas we need to understand the events and actions related to them. There is a new action *Ask* and new triggers including *OnEndOfActions, OnAssignEntity, OnChooseEntity* and *OnChooseProperty.* Each is described in the next sections.

### *Ask*

The current input actions all expect a specific response like a number or a choice. This does not support more complex expressions that are a mixture of multiple values, corrections, etc. They also do not provide enough context to help disambiguate response. What is needed is a new prompt that provides the context and then allows any response i.e., the action does not wait for a response. To this end a new action *Ask* has been added. *Ask* has three properties:

* **activity:** The message to send to the user. For example, “What is the maximum bid you would like to make?”
* **expectedProperties:** A list of the properties that you expect the user to answer. This information is used to decide which property a potentially ambiguous entity belongs to. For example, if some has “3” from the schema above it could be either *MinBid* or *MaxBid*, but if we asked “What is your max bid?” then we would map the number to *MaxBid.* This still allows a response like “5 and a min bid of 3” where we know from *expectedProperties* that we should interpret 5 as a *MaxBid* while still allowing the recognizer to tell us that 3 is a *MinBid.*
* ***defaultOperation:*** By default the operation is picked up from the *$defaultOperation* in the schema, but if an *Ask* prompt was something like “What do you want to remove from toppings?” then we want the adaptive dialog to know that the default operation is *Remove().*

*Ask* also has one property it adds to the dialog memory, **$retries**. This is a counter that is incremented each time you execute an *Ask* action with the same *expectedProperties* until a *ChooseEntity, ChooseProperty* or *AssignEntity* event resets it to zero. You can use this counter to provide help or change the prompts you ask.

### *OnEndOfActions*

This event is raised when the last action is executed. Remember actions are flows that are executed in order. This event tells the event driven dialog to use triggers to figure out the next step to do. For example, you might have a trigger for each property that does *Ask* if the property is missing, one that looks at the number of *$retries* and one that calls an API when all properties have values. Each time *OnEndOfActions* is raised one of the triggers runs. Initially that would be to ask for missing properties and if you kept not answering the question it might trigger on *$retries* to print help. Finally, it would call the API when all values were collected. for a missing property would run if you had triggers that do an *Ask* for each missing

### *OnChooseEntity*

Sometimes and entity is ambiguous i.e., “wheat” might mean the value “whole wheat” or “multigrain wheat”. This is identified when multiple possible resolutions come back from the recognizer. When this is detected the *ChooseEvent* event is raised to handle the ambiguity. An *OnChooseEvent* trigger will fire on this event and allows you to optionally specify:

* **operation:** The operation that would be applied to the property and entity.
* **property:** The property that would be assigned.
* **entity:** The type of the entity that is ambiguous.

An action can access information about the event through *turn.dialogEvent.value* which includes:

* **property:** The name of the property that would be assigned once this ambiguity is resolved.
* **operation:** The operation that will be applied to the entity and property.
* **entity:**The ambiguous resolution which will be a list of possible resolutions.

Typically, this event is handled by asking the user which value they meant. If the response matches one of the choices in entity or assigns the same property a different value, then the event is resolved.

### *OnChooseProperty*

This event is raised when there are multiple possible combinations of operation, property and entity that can be combined. For example, if you ask “What is your name?” and the response is “3” then a response might be “By 3 did you mean a min bid, max bid or name?”. An action can access information about the event through *turn.dialogEvent.value* which is a list of alternatives each of which has information on each choice.

* **operation:** The operation that will be applied to any entity or property.
* **property:** The name of the property to use.
* **entity:**An entity value.

The event is resolved if a user response matches either the operation or property of one of the alternatives. Picking one alternative will cause other compatible alternatives to be selected as well. For example, if the response to “Did you mean ham meat, swiss cheese or name is ‘ham and swiss’” was “meat” you would get an assignment of “ham” to the *Meat* property and of “swiss” to the *Cheese* property. If you answered, “name” you would only get an assignment to the *Name* property.

### *OnAssignEntity*

There are many possible paths to ending up with an operation, property and entity. It could be specified all at once like “ham and swiss on rye” or in response to a specific property or as the result of resolving ambiguity. Whatever the path, we want a single place where we can validate the arguments to an operation and transform values if needed before applying the operation. Some validation examples are checking min/max for numbers or checking the units are what is expected. A transformation can also do things like add default units to a number to assign to a property or convert dimensions in one unit to another.

### Entity Analysis

When an the *RecognizedIntent* is fired, an adaptive dialog that has an associated schema will analyze the entities to create possible event, operation, property, entity assignments and then choose between them. These potential assignments are merged with any outstanding potential assignments into a queue found in *$events.Assignments*. Each assignment in the queue will be raised as an event in turn possibly resulting in more potential assignments that are merged into the queue. Usually, potential assignments are removed by analyzing entities in response to clarification questions, but it is also possible to manipulate the queue directly. For example, you might want to support not skipping a clarification answer or you might be able to resolve the ambiguity without user involvement. The *EntityAssignment* class in the SDK describes the properties that make up each assignment. The diagram below shows the typical flow with some of the steps outlined after. It is only typical because at each event you can insert your own processing. Ovals are events, square boxes actions, rounded boxes processing and diagonal boxes input.

Ask

Entity Analysis

Assignments

User Input

Recognizer

Steps in a typical event drive flow.

1. *EndOfActions* fires which causes the most specific *OnEndOfActions,* highest priority trigger to fire.
2. An *Ask* action is executed by the trigger and sets up the *ExpectedProperties* and *ExpectedOperation* as context.
3. At some point the user responds which generates an *ActivityReceived* event.
4. The activity is processed by the dialog’s recognizer which raises a *RecognizedIntent* event.
5. The adaptive dialog analyzes the entities to generate possible assignments. This analysis works by taking each hierarchical entity recognized and creating candidate assignments where missing property or operation information is filled in using information in the schema and the *ExpectedProperties or ExpectedOperation* context. The candidates are sorted with expected possibilities first. A candidate is selected and then any overlapping candidates are removed or become alternatives for a *ChooseProperty.* The resulting candidates are merged into *$events.assignments* and ordered by event: *AssignEntity, ChooseProperty, ChooseEntity* and the by the order of schema *$operations,* expected before unexpected, by turn and finally by position in the utterance.
6. If there are any assignments, then the corresponding event is raised. *AssignEntity* are immediately removed from *$events.assignments.* The other assignments are usually removed only in response to another user input through an *Ask,* but it is also possible to manipulate the assignments directly in trigger actions.
7. Once all assignments are complete another *EndOfActions* event is raised and the cycle repeats until a trigger ends the dialog.

## Form Generation

The mechanisms described in this doc are complex, but they do support rich conversation where at every point the user can respond in a natural way. Luckily, most of the time you do not have to pay attention to these details. Form generation is new preview feature in Composer that will automatically generate an event driven dialog for you. It generates language understanding (LU), language generation (LG) and dialog management (DM) assets from a JSON schema. The [generated NLU model](https://microsoft.sharepoint.com/:w:/t/ConversationalAI785/EZMPNp5PJSBFq_G9Z_QhXO4BYvsNdTJd2HFx1MFRSfZp7Q) takes full advantage of LUIS features and can be refined over time to incrementally improve it. This is much simpler than having to master LUIS as well as figure out how to create and use a robust model. The resulting form dialog has these features:

* Support for all LUIS entities.
* Multiple entity input: “order a ham and swiss sandwich”
* Disambiguation: “By ‘wheat’ did you mean [whole wheat, multigrain wheat]?’, “Was ‘rye’ your name or the bread you wanted?”
* Multiple operations including revision: “Add lettuce and remove pickles”
* Built-in intelligence for min/max, units and more coming.
* Ability to create richer prompts asking for multiple entities and providing status: “What kind of toppings and sauces do you want on your ham and swiss sandwich?”
* Standard support for Help, Welcome, Navigation, Confirmation

Form generation can build the scaffolding to bootstrap these capabilities, but to make a fully functional bot you need to use Composer to extend and refine the initial assets.

### Customizing LG

#### Value

#### Entity

#### PerProperty

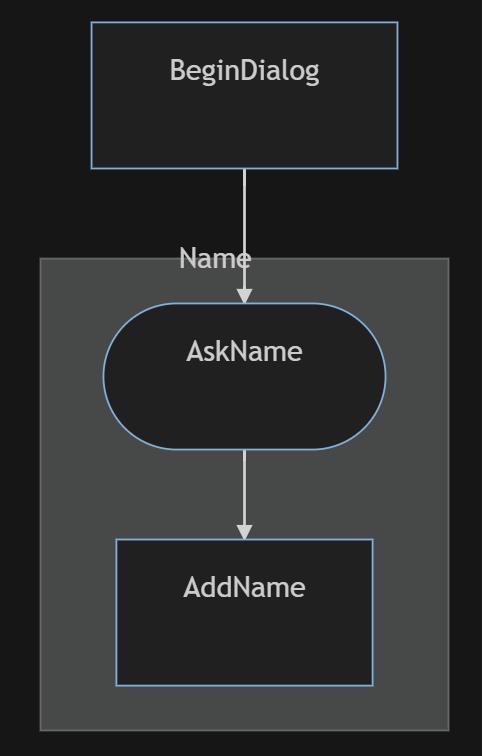
#### Form level stuff

### Customizing LU

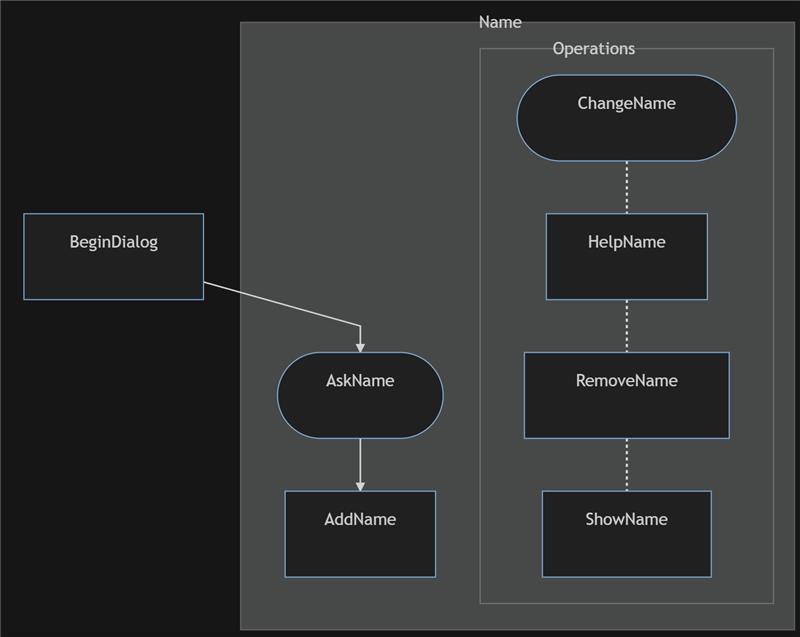
### Customizing Dialog Management

### Visualization of “flows”

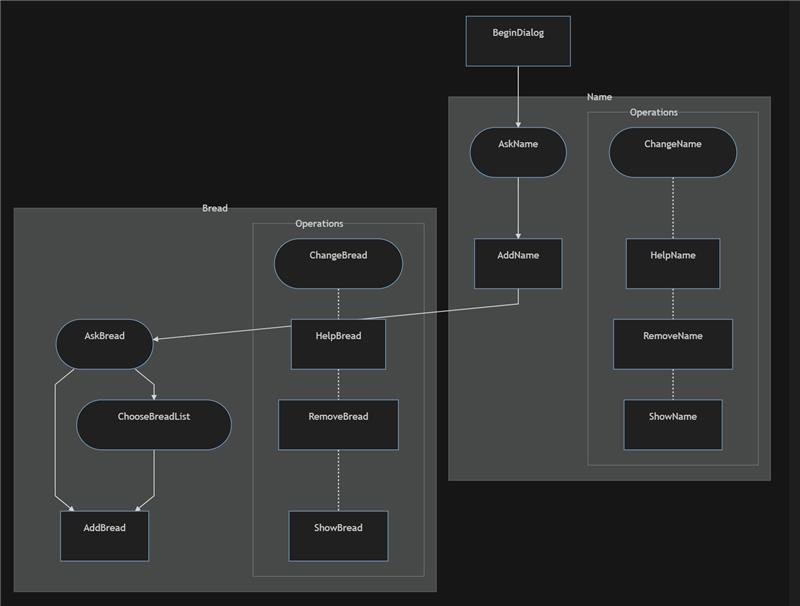
This is some experiments with a visual representation of the golden path flow using mermaid. Each node in the graph is a trigger handler and the lines are connecting the "expected" flow. Imagine a schema for ordering a sandwich that asks for name, bread, meat, and cheese. Let us start with BeginDialog and the Name property. This represents the mainline where we do an Ask and expect to get back Name as an answer.



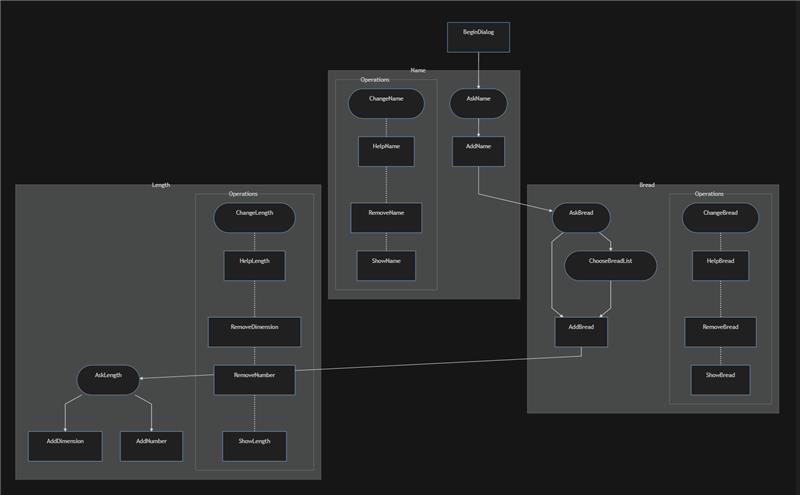
We might want to capture the related operations that can be done, but that are note the mainline flow. (The dashed lines don't mean anything it is just for layout purposes.)



Adding Bread would look like this because it also adds in the possibility to choose between ambiguous entity values like “wheat” could mean “whole wheat” or “multigrain wheat”.



If you added Length it has two entities so it has an add for two different entities.



After adding meat and cheese and a prompt that asks for both properties at once.

