Form Generation NLU Model

# 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 dialog approach where the bot asks a question until it gets an answer or gives up. 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. Form generation is new preview feature in Composer that will automatically generate language understanding (LU), language generation (LG) and dialog management (DM) assets from a JSON schema for more robustly handling natural language. The generated NLU model 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 supplying 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. This document describes the underlying NLU model, why it is built the way it is and how to extend it.

# NLU Model Concepts

To understand the generated NLU model it is important to first understand the concepts that make up the form generation model. These include:

* **Property:** The goal of generated form dialog is to fill in the form properties defined by a JSON schema. These are the normal Composer properties with their type defined by JSON schema. The type information is used to generate the assets needed to recognize, validate, and show the property. Currently we only support flat properties like string, number, email, and statically known objects like a LUIS dimension. Eventually we want to support custom objects but that requires more development.
* **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 say 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 supply 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.
* **LUIS Entity:** LUIS has no direct knowledge of properties or operations—from its perspective they are all entities that can be recognized in an utterance. As will be seen later, generated models use a suffix to identify what kind of role LUIS entities play, *\*Entity* for entities*, \*Property* for properties and *\*()* for operations. When we need to refer to all the entities LUIS can recognize including ones interpreted as properties or operations, this is the term we will use.
* **Feature:** A feature in LUIS allows you to create a special kind of “label” that is automatically computed for you. This allows learning more robust machine learning models from fewer examples. LUIS can automatically create features for you from a list of phrases or via using other models like the presence of an entity, making an intent more likely.
* **Hierarchical Entity:** A hierarchical entity allows you to define how to combine underlying entities into a more complex structure. This can be used for recognizing something like an address that is made up of lower-level entities like a street, city, and zip. In the form dialogs we use this to model an operation and the property or entity it is applied to.

# NLU Model

To understand the generated NLU model, I will show a schema and show bits of the generated LU file and talk about what it means and how you can change it. Imagine you had a card game and wanted to collect a name, maximum bid, and suit together with an optional minimum bid. From an application perspective this would be a reasonable schema:

{

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

    "type": "object",

    "properties": {

        "MinBid": {

            "type": "number"

        },

        "MaxBid": {

            "type": "number"

        },

        "Suit": {

            "type": "string",

            "enum": ["clubs", "hearts", "diamonds", "spades", "NoTrump"]

        },

        "Name": {

            "$ref": "template:personname.schema"

        }

    },

    "required": [

        "Name",

        "MaxBid",

        "Suit"

    ],

    "optional": [

        "MinBid"

    ]

}

If you generate the form you will end up with a collection of LU, LG, and DIALOG files. To understand the NLU model you need to look at files inside *language-understanding/en-us*. Files are found in a subdirectory for each property and in *form* for parts that are used in the specific properties. Together they define the NLU model that is used by the generated form dialog. The generated form dialog works by defining triggers for *operation(property(entity)),* but it needs to be prepared for any of *operation, property,* and *entity* to be missing. The generated form dialog needs to be prepared to deal with:

* A plain entity like “3” where we need to figure out the property and operation. The SDK runtime resolves this property by seeing if the type of entity corresponds to one of the expected properties for an Ask action. If it does not, then it looks at the schema and identifies the non-expected properties that can be defined by an entity of that type. (In this schema *MinProperty* and *MaxProperty*.) To figure out the operation the SDK runtime will fill in the default operation based on the property that could use the entity. It is quite possible for there to be multiple possible properties in which case the user will be asked to choose which property.
* A plain property like “max bid”. In this case we do not have an entity or operation. There is no default entity or operation and by default the generated form dialog ignores it. (Alternatives would be to show the current value or change the value which you could define with your own trigger.)
* A plain property and entity like “max bid of 3”. Here we know that the number is associated with the *MaxBidProperty*, but we do not know the operation, but we know the default operation for the Ask prompt and for the specific property so we can fill that in.
* Operation and property like “show max bid”.
* Operation, property, and entity like “add max bid of 3”.

To support recognizing all those cases we need a LUIS application that models those concepts and has enough labeled data to work. To understand how form generation does this we will start with the *MaxBid/Tutorial-MaxBid-number.en-us.lu* file. The filename itself gives you information. *Tutorial* is the name of your schema, *MaxBid* is the name of the property, *number* tells you the entity and *en-us* the language.

## Definitions

The beginning of the file defines the LUIS entities that make up the model for the *MaxBid* property. The first line in the file tells us that we are using the LUIS prebuilt number entity.

@ prebuilt number

This is all we need to recognize plain entities.

The next few lines define a phrase list *MaxBidPhrases* for referring to this specific property in utterances.

> Define phrase list for MaxBid property.

@ phraselist MaxBidPhrases(interchangeable)=

**-**max

**-**bid

**-**max bid

Any of the phrases in the list above would be considered to have the *MaxBidPhrases* feature. The effect of a feature is to allow learning with fewer examples i.e., a single utterance like “a max bid of 3 clubs” will be enough to also recognize “a bid of 3 clubs” or “a max of 3 clubs”. If you later added another phrase “maximum bid” it would also recognize “a maximum bid of 3 clubs” without having to add it as a training utterance.

The next few lines define our first hierarchical entity, in this case one for recognizing the combination of the MaxBid property and a number entity. This is to cover the case where we know the property and might have an associated entity value.

> Define property entity and optional value for MaxBid property.

@ ml MaxBidProperty usesFeature MaxBidPhrases

**-**@ number number

The *ml* tells LUIS that the hierarchy is learned from examples. The LUIS entity name ends with *Property* to tell you that this LUIS entity is used as a property in the form dialog. *usesFeature MaxBidPhrases* tells LUIS that the phrase list we setup to recognize the ways people might say “maximum bid” are useful for recognizing this hierarchical entity.

The indented line tells us that *MaxBidProperty* can have an optional child *MaxBidProperty.number* which must be of the prebuilt *number* entity type. This can be a bit confusing—*MaxBidProperty.number* and *number* are not the same. *MaxBidProperty.number* is learned and needs labeled examplesand *number* is a LUIS prebuilt and requires no learning. We do not need a lot of examples because *MaxBidProperty.number* must be a *number* which means that the presence of the prebuilt *number* feature makes it easier to learn.

The hierarchical entity covers a specific span in the user response. For example, in the fragment “3 max bid” we want that whole phrase to be interpreted as being about the *MaxBid* property and have an entity value of 3. For LUIS to learn this we need to label utterances. Here is the label for utterance “3 max bid”.

**-**{@MaxBidProperty={@number=3} max bid}

The *MaxBidProperty* spans the whole utterance. The child *number* only covers the number 3 and is used for learning *MaxBidProperty.number*. The phrase “max bid” is what tells you that the *MaxBidProperty* entity is involved. (And since this is in the *MaxBidPhrases* feature we get other phrases like “max” recognized for free.)

The next lines are all about defining the hierarchical entities used to model operations which work over properties and entities. The LUIS entity name ends with () to show the entity is interpreted as an operation by the dialog management. Default operations include *Add(), Remove(), Clear(), Change(), Show()* and *Help().* They all work the same way so we will look just at the *Add()* operation.

@ ml Add() usesFeature AddPhrases

    -@ ml MaxBidProperty usesFeature MaxBidProperty

**-**@ number number

This defines the *Add()* operation over an optional *MaxBidProperty* property which has an optional entity *number.* It would be nice if we could just reuse the hierarchical entity for *MaxBidProperty*, but LUIS does not let us do that directly. It would have been ideal to just say - @ MaxBidProperty i.e., a child named *MaxBidProperty* that has a type corresponding to the *MaxBidProperty* hierarchical entity. Instead, the definition of the *MaxBidProperty* hierarchical entity is defined with the same names as the *MaxBidProperty* hierarchical entity and *usesFeature* allows the operation hierarchical entity to benefit from the labels we did for the standalone *MaxBidProperty.*

The whole operation uses a set of *AddPhrases* to help learn from fewer examples which are defined in *tutorial-form-phrases.en-us.lu* like this:

@ phraselist AddPhrases(interchangeable)=

**-**add, set, with, is, are, of

One thing to be aware of is that operations span multiple properties, and the operation hierarchies are distributed across per-property files. For example, if you look in *MinBid/tutorial-MinBid-number.en-us.lu* you will see this definition.

@ ml Add() usesFeature AddPhrases

    -@ ml MinBidProperty usesFeature MinBidProperty

**-**@ number number

LUIS ends up putting all the definitions together into a single definition, i.e.

@ ml Add() usesFeature AddPhrases

    -@ ml MaxBidProperty usesFeature MaxBidProperty

**-**@ number number

    -@ ml MinBidProperty usesFeature MinBidProperty

**-**@ number number

This allows an operation to be over multiple properties like “add a min bid of 3 and a max bid of 5”.

Depending on the underlying entity you might see other definitions. For enum properties like suit you will see the entity defined like this:

> List entity for Suit enum values

@ list SuitValue =

**-**clubs:

**-**clubs

**-**hearts:

**-**hearts

**-**diamonds:

**-**diamonds

**-**spades:

**-**spades

**-**NoTrump:

**-**no

**-**trump

**-**no trump

This defines a list entity *SuitValue* which ends with the Value suffix to tell you the role it plays in form dialogs. The list resolutions all come directly from the schema and the phrases are automatically generated by breaking up the resolution. In this case you probably want to remove “no” and “trump” since they do not really mean the *NoTrump* concept. If you add or remove enum elements in your schema and regenerate, this will get updated, but it will not overwrite any changes you make.

## Labeling

The rest of the file consists of labeled utterances. They are generated from templated utterances and depending on your domain the examples may or may not be good examples. The details of your file may also change as we improve the underlying templated utterances. The label section starts with a single intent corresponding to your form.

> Add examples of operations on the 'MaxBid' property to the intent 'tutorial'.

**# tutorial**

The generated model only has two intents—one for your whole form i.e., *tutorial* in this case and the *none* intent. The reason is that intents are modeled using operation hierarchical entities as described above.

The first section has labelled examples for entities without any property or operator information.

>> entity: Standalone entities that map to multiple properties

In this case the section is empty because *number* is a prebuilt entity that requires no training examples. This section will also be empty if the underlying entity is only used in one property—in that case the property is implicit in the kind of entity value.

The next section has examples labeled with property and entity.

>> entity: Standalone MaxBidProperty with number

**-**{@MaxBidProperty={@number=1} max}

**-**{@MaxBidProperty={@number=1} max bid}

**-**{@MaxBidProperty={@number=32.4} max}

You can see the phrases that mean the *MaxBid* property i.e., “max” and “max bid” being labeled as part of a *MaxBidProperty* hierarchical entity. You can also see some generic numbers like “1” and “32.4”. For bids, 32.4 is clearly not a meaningful example, but it is OK in this case because *MaxBidProperty.number* can lean on the presence of the prebuilt *number* to learn that the actual number does not matter.

The next section has utterances that just show phrases that mean a particular property even without an entity.

>> property: Standalone MaxBidProperty name

**-**{@MaxBidProperty=max}

**-**{@MaxBidProperty=max bid}

The next section has examples of operations, properties, and entities together.

> Add() examples for MaxBidProperty and number

**-**{@Add()=add {@MaxBidProperty={@number=215} max bid}}

**-**{@Add()=add {@MaxBidProperty={@number=1000} max}}

**-**{@Add()={@MaxBidProperty=max bid is {@number=99}}}

**-**{@Add()={@MaxBidProperty=max bid is {@number=1000}}}

If you have an array property you will also see a section that shows multiple values like this.

> Add() examples with multiple number

**-**{@Add()={@MaxBidProperty=max bid are {@number=99}}}

**-**{@Add()={@MaxBidProperty=max are {@number=1}}}

**-**{@Add()={@MaxBidProperty=max bid are {@number=1}}}

**-**{@Add()=add {@MaxBidProperty={@number=215} and {@number=32.4} max}

The rest of the file has other labelled examples of all the other operations. Including some examples of just operation and property.

> Show() examples for MaxBidProperty

**-**{@Show()=show {@MaxBidProperty=max bid}}

**-**{@Show()=show {@MaxBidProperty=max}}

**-**{@Show()=value of {@MaxBidProperty=max}}

**-**{@Show()=value of {@MaxBidProperty=max bid}}

## Common LU files

In addition to the per-property LU files there are also some files to define common building blocks as well as cross-property examples. In particular:

* **Tutorial.en-us.lu:** This is the master LU file for a particular language. It includes all the other LU files and defines settings for LUIS.
* **form/tutorial-form-phrases.en-us.lu:** Define phrase lists for operations that are used within each property LU file.
* **form/tutorial-form-NoneIntent.en-us.lu:** Define the “None” intent. The examples here are nonsense and you can improve your model by adding things that should not be a part of your form.
* **form/tutorial-form-global.en-us.lu:** This has examples that cross multiple properties. Depending on your schema they may not be particularly good examples.

> Add() examples across properties

**-**{@Add()=add {@NameProperty={@personName=Mike Smith}} and {@MinBidProperty={@number=1} min}}

**-**{@Add()=add {@MaxBidProperty={@number=1} max} and {@MaxBidProperty={@number=1} max}}

**-**{@Add()=add 1 and {@MaxBidProperty={@number=1} max bid}}

# Best Practices

The generated NLU model provides a scaffold that interacts with dialog management and language generation to supply a complete dialog. It is not possible to generate a perfect model for every application, so it is important to be able to improve the model using Composer. Here are a few of the best practices:

* Whenever possible change the schema and regenerate. For example, adding or removing a property or enumeration could be done manually, but much easier to regenerate since it will update LU, LG, and dialog assets. We make every effort to not change any file where you have changed labels, which means in some cases you might have to manually label examples, but you should not lose the work you have done.
* To add new definitions or labels add them to the top-level LU file or create a new file and include it. This will ensure that regeneration does not touch anything there.
* Removing/changing existing labels. This is a little trickier since once you change labels, we will not generally touch any labels in that file which means if you regenerate you will need to manually add or update utterances. Usually, you do not need to remove or relabel utterances—the utterances may not be syntactically correct for your schema, but they will not hurt.
* Search and replace is your friend. In Composer you can always show the code of the LU file and Ctrl+F will bring up a search/replace function. If you need to do multiple file replace it might be worth opening the files in Visual Studio Code and doing it there.