Generating Dialogs   
from Schema, APIs and Databases

Contact: Chrimc (Chris McConnell)  
Date: 1/21/2020

# Overview

The Bot Framework has a rich collection of building blocks, for developers to create conversational experiences. To facilitate creating a bot with the most common customer scenarios, we created the popular open sourced Bot Framework Virtual Assistant (VA) solution accelerator. This helps a developer quickly bootstraps key components on Azure (orchestration, language understanding, language generation, QnA, dialog management, speech, storage, analytics) and also provides a set of reusable, customizable skills.

Today many customers would like to make it even easier to create skills, and make the bot’s conversational capabilities more expressive and natural, while still reusing their custom schema, apis and databases. Generated Dialogs focus on making it easy to combine assets like the existing VA skills, leverage custom existing assets (schema, APIs and databases) and facilitates authoring event driven and ambiguity handling, which helps make conversations more natural.

This capability is under development and will release in the Bot Framework SDK and CLI in public preview in March 2020 (R8), and the BF Composer in Q2 2020. It is also being tested by Power Virtual Agent and Speech teams to support moving more fully to the Bot Framework runtime.

### Goals

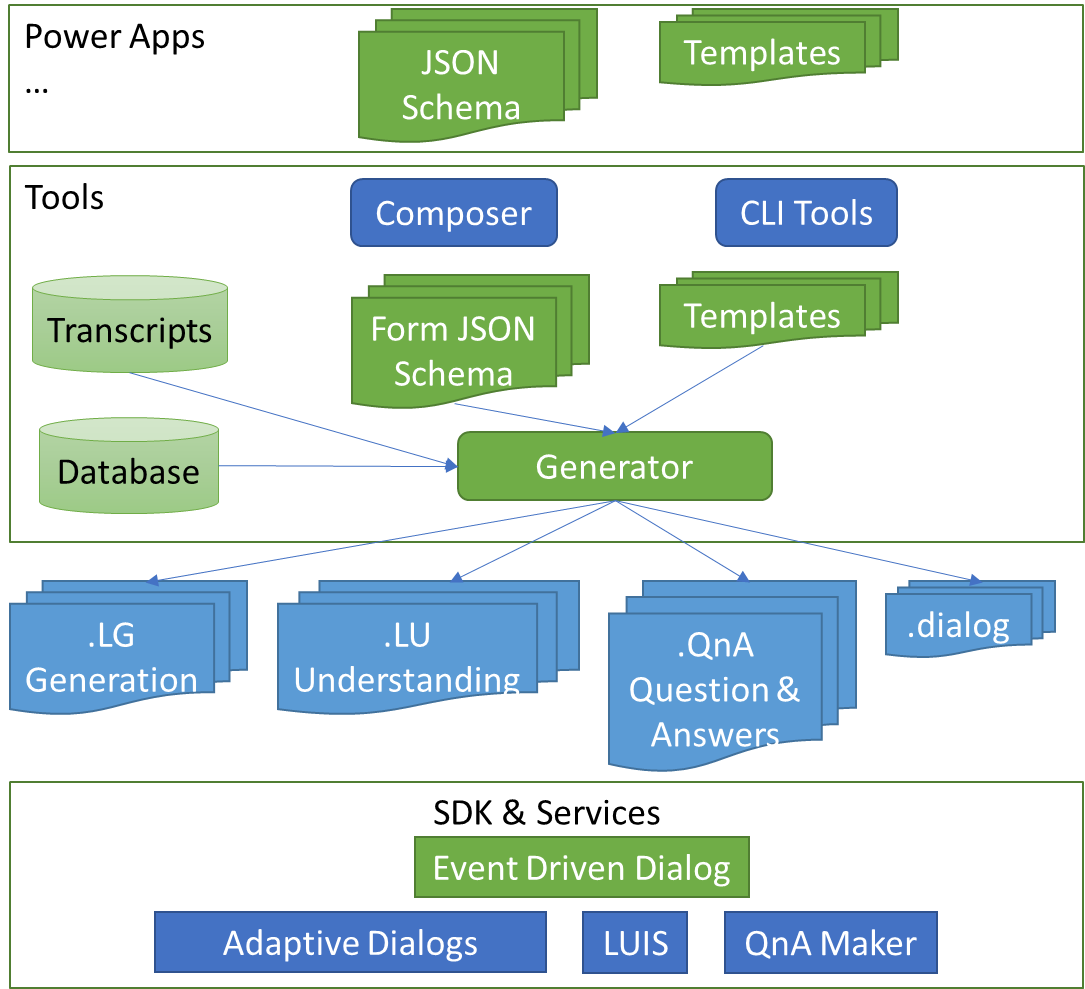
* Leverage existing customer assets like schema, APIs or databases to automatically generate dialog assets.
* Easy to customize and understand generated assets in the Bot Framework Composer.
* Functional bot that handles ambiguity from the start with quality only limited by effort.
* Integration of generation into the Bot Framework Composer in a way that builds on existing concepts.
* Utilize generation to help teams move from custom runtimes to standard Bot Framework runtime utilizing Adaptive Dialogs.

# Architecture

The overall approach breaks down into three parts:

1. SDK support for event driven conversations and ambiguity.
2. A generator which given a schema or database can generate bot assets from reusable and customizable templates.
3. Common patterns captured as templates that connect all the assets required for a bot.

In the picture below, dark blue are our existing tools, services, and SDK. Light blue are the generated assets. Dark green are the new parts being added to support generation. The next sections describe the additions in more detail. The overall effort is at the point where pieces will be integrated, and broader feedback is needed.



# SDK Support for Event Driven Dialogs

In order to be robust, a bot needs to be able to handle any user response at any point in a conversation. This requires an event driven architecture with conversational events like the arrival of a user utterance, the identification of an intent and where the mapping of an entity to a property is ambiguous. Adaptive Dialogs provide the core capability to integrate flow and event driven dialogs. Language generation and trigger evaluation provide a rich foundation for responding to conversational events in a natural way. It is not easy to go from an “***entity***” recognized in an utterance to a “***property***” which is stored in memory. For example, a single yes/no confirmation entity should be defined, but there are many different properties that might use it like “Are you stuck?”, “Is this correct?”, etc. In an event driven dialog someone might respond to any of those questions with something other than a confirmation, i.e. “change my destination to Dallas” so response handling needs to be driven by the user—not just expectations.

In order to support event driven conversations and to better handle ambiguity, Adaptive Dialogs include:

* A schema for describing properties and optionally how entities map to them. This includes scope and lifetime information.
* An open-ended prompt “Ask” which includes the expected properties. Expected properties are used to resolve entity to property ambiguities and could also be used to do things like prime speech.
* Entity and intent processing to identify and track ambiguities which can be surfaced as events. Handling ambiguity makes it much easier to write event driven dialogs.
* System properties for schema and expected properties which can be used both in triggers and to make language generation more sophisticated.

## Schema

We need schema in Adaptive Dialogs for several reasons.

* It helps drive the composer UI so that we know about properties and their types for intellisense, debugging, and error checking across LU, LG and Dialog.
* Schema defines the contract between skills.
* The definition can be used to automatically generate OBI dialog assets including LG, LU, QnA and Dialog. You start by defining the properties you want, and the dialog assets are generated from that definition. This provides a natural place to define the connections between properties, entities, and business logic.

JSON Schema is our core schema language. We are already using it for describing OBI parts and it is a standard. The schema has some optional extensions for helping to drive the extension process. Using standard JSON Schema we can define properties in terms of reusable templates like a list of states or the definition of an address. These templates can be a mixture of fixed and dynamically generated assets that connect language understanding, language generation and dialog management into a functional bot. Reuse can also include standard templates for things like chit-chat, help, cancel, confirmation, etc.

# Generator (bf dialog:generate)

The generator is written in typescript as part of bf cli. Given a JSON Schema definition and templates it will generate dialog assets to define a functional bot. Schemas can reuse existing schema pieces which have their own associated templates for generating dialog assets. You can also include standard mechanisms for things like help/cancel/correction, etc. The templates are overridable and can use our language generation system to dynamically generate the .lg, .lu, .qna or .dialog assets. The templates can range from full skills like VA down to generating assets from an enum in a property definition. Because they are overridable, it is also possible to support many different styles of dialogs. Some example styles:

* A guided dialog to get specific properties.
* An open-ended dialog where the only prompt is “What can I do for you?”
* A deferred dialog like PVA does where if you do something unexpected, handling it is deferred until they reach that point in a static flow.

You can see more at the current [documentation](https://github.com/microsoft/botframework-cli/blob/master/packages/dialog/src/commands/dialog/readme.md) or by looking at the [code](https://github.com/microsoft/botframework-cli/tree/master/packages/dialog/src/library). The generator is stable except for how to update existing generated assets which will require some more work.

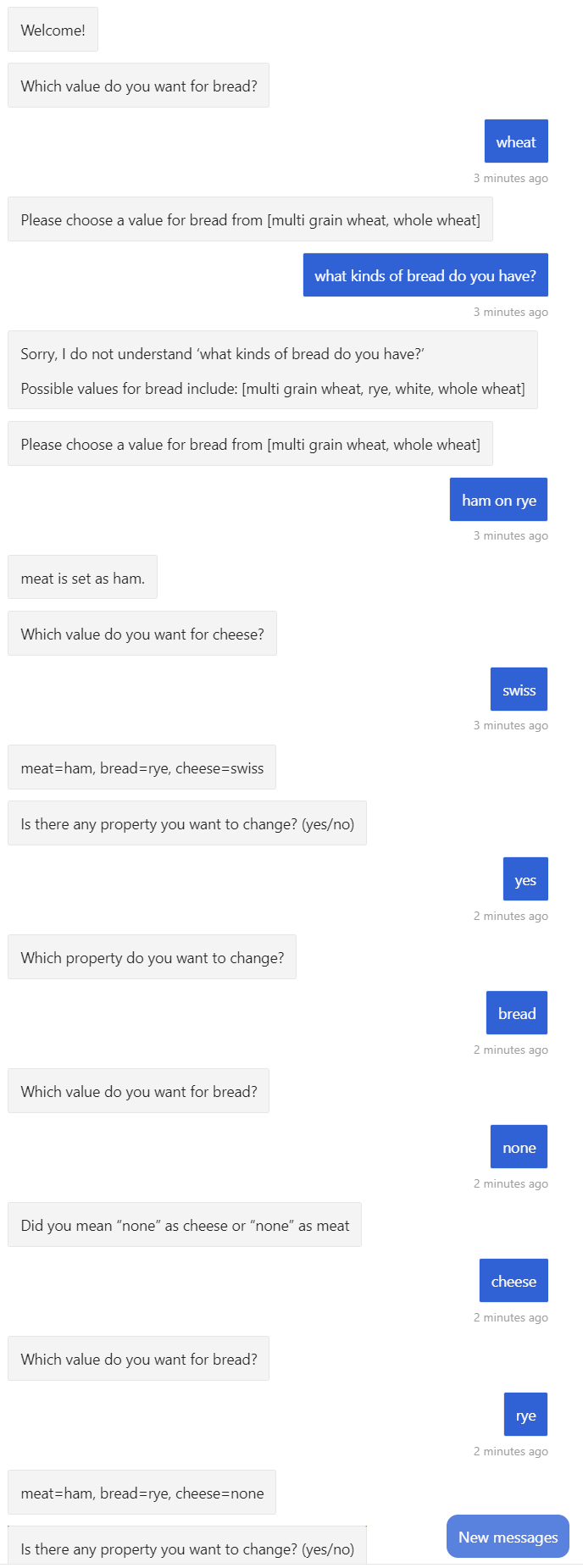
# Templates

By design, the runtime and generator should be stable and unopinionated. Most of the actual functionality is driven by the [templates](https://github.com/microsoft/botframework-cli/tree/master/packages/dialog/templates) that define how to generate the assets. We have been rapidly iterating on the templates and today we support automatically generating .lu, .lg, .qna, and .dialog files for the mapping from prebuilt or enum generated entities to properties. The resulting files are enough for a functional bot and are easily customized. We also handle unknown input, unexpected changes, clarifying entities, choosing slots, help and cancel. The assets are only defined for English so far, but they are structured to support multiple natural languages. You can see an example of assets generated from a schema and templates [here](https://github.com/microsoft/botbuilder-dotnet/tree/master/tests/Microsoft.Bot.Builder.TestBot.Json/Samples/GeneratedForm).

# Appendix

A few more details on how the generation process works and an example dialog.

## Example Conversation

This is a conversation generated from a schema with three required properties and some optional ones—no manual effort was applied. It shows contextual help, entity and slot clarification, unexpected response confirmation and completion confirmation.

## Schema

Below is a snippet of JSON schema defining a “Bread” property and the possible values together with the required properties and a reference to the standard schema which defines help, cancel and confirmation properties.

{ ...

"Bread": {

"type": "string",

"enum": [

"multiGrainWheat",

"rye",

"white",

"wholeWheat"

]

},

"required": [

"Meat",

"Bread",

"Cheese"

],

"$requires": [

"standard.schema.dialog"

]

}

## Generated assets

Below are some fragments of assets generated through bf dialog:generate using the schema information. Ellipsis are used to show where there is more information. To give an idea of scale there are usually 6 files generated per property plus additional ones for help, cancel and confirmation.

From the schema we can automatically define a “BreadEntity” list entity via .lu files and also associated .lg files like below:

sandwich-BreadEntity.en-us.lu:

$BreadEntity:multiGrainWheat=

- multi

- grain

- wheat

- multi grain

- grain wheat

- multi grain wheat

sandwich-BreadEntity.en-us.lg:

# BreadEntity(value)

- SWITCH: @{value}

- CASE: @{'multiGrainWheat'}

- multi grain wheat

- CASE: @{'rye'}

- rye

…

# clarifyBreadEntity

- @{clarifyEnumEntity('Bread')}

We can also define an .lg file for the “Bread” property that makes use of some underlying common templates. The library templates can provide sophisticated built-in behaviors based on things like retries or schema information like min/mas for number. In composer this provides a place where you can change or localize what how language is generated. You can also see how the generated file makes use of the “BreadEntity” .lg template for displaying information stored in the property.

sandwich-Bread.en-us.lg:

# AskBread

- @{askEnum('Bread')}

# BreadName

- bread

# Bread(val)

- @{BreadEntity(val)}

In addition to the language-oriented files we also need .dialog files for putting everything together. For example, here is the trigger handler to ask for “Bread” if it is not defined and it is required. It makes use of the underlying .lg definitions for bread and utilizes the open-ended “Ask” action which shows the what properties are expected in response.

{

"$schema": "https://raw.githubusercontent.com/microsoft/botbuilder-dotnet/chrimc/map/schemas/sdk.schema",

"$type": "Microsoft.OnAsk",

"condition":"and(!$Bread, contains(dialog.requiredProperties, 'Bread'))",

"actions": [

{

"$type": "Microsoft.Ask",

"activity": "@{AskBread()}",

"expectedProperties": [

"Bread"

]

}

]

}

There are also other dialogs generated for handling things like entity clarification, confirmation and so on.

## Templates

The underlying templates make use of the language generation system to generate the required assets. It can be a little confusing, but most customers will use, not write these and it is very powerful. Here for example is how we generate an .lu file from the schema.

# filename

- @{formName}-@{property}Entity.@{locale}.lu

# template

- @{join(foreach(schema.properties[property].enum, enum, concat('$', property, 'Entity:', enum, '=\n', synonyms(enum))), '\n\n')}

# synonyms(value)

- @{join(foreach(phrases(value), phrase, concat('- ', phrase)), '\n')}