

# Contents

[Bot Framework Composer Documentation](#)

[Overview](#)

[Introduction to Bot Framework Composer](#)

[What's new?](#)

[Composer releases](#)

[Installation](#)

[Install Composer](#)

[Build Composer from source](#)

[Supported operating systems](#)

[Quickstart](#)

[Application settings](#)

[Create your first bot with Composer](#)

[Create your first bot with Azure](#)

[Tutorials](#)

[0. Tutorials introduction](#)

[1. Create a bot](#)

[2. Add a dialog](#)

[3. Add actions to a dialog](#)

[4. Add interruptions to a conversation](#)

[5. Add language generation](#)

[6. Add cards](#)

[7. Add LUIS](#)

[Concepts](#)

[Templates](#)

[Dialog](#)

[Trigger](#)

[Conversation flow and memory](#)

[Skills](#)

[Natural language processing in Composer](#)

- [Language generation](#)
- [Language understanding](#)
- [Composer best practices](#)
- [Conversational user experience](#)
- [Composer extensions](#)
- [Connections](#)
- [Speech](#)
- [Orchestrator](#)
- [Packages](#)
- [Hand off to human](#)
- [Extending your bot with code](#)
- [Templates & Samples](#)
  - [Get started with templates](#)
    - [Enterprise Assistant Bot Template](#)
      - [Overview](#)
      - [Tutorial](#)
    - [Enterprise Calendar Bot Template](#)
      - [Overview](#)
      - [Tutorial](#)
    - [Enterprise People Bot Template](#)
      - [Overview](#)
      - [Tutorial](#)
  - [Learn from samples](#)
- [How-To](#)
  - [Develop](#)
    - [Send messages](#)
    - [Send cards](#)
    - [Ask for user input](#)
    - [Manage conversation flow](#)
    - [Define triggers](#)
    - [Define intents with entities](#)
    - [Add LUIS to your bot](#)

[Add QnA to your bot](#)

[Create QnA Maker knowledge base](#)

[Create bots in multiple languages](#)

[Add user authentication](#)

[Send an HTTP request](#)

[Migrating an SDK-first bot to Composer](#)

[Update from Virtual Assistant to Composer](#)

[Extend Composer with extensions](#)

[Host Composer in the cloud](#)

[Manage packages for a bot](#)

[Create and share packages](#)

[Extend a bot with code](#)

[Create a custom action](#)

[Create a custom trigger](#)

[Skills](#)

[Create a local skill](#)

[Connect to a remote skill](#)

[Add single sign-on for a skill](#)

[Test](#)

[Test bots in the Emulator](#)

[Test bots in Web Chat](#)

[Debug](#)

[Validation](#)

[Capture bot's telemetry](#)

[Provision](#)

[Provision Azure resources](#)

[Admin Handoff for LUIS](#)

[Admin Handoff for QnA Maker](#)

[Admin Handoff for Speech](#)

[Publish](#)

[Publish your bot to Azure](#)

[Publish your skill to Azure](#)

## Create a CI/CD pipeline (Preview)

[Glossary](#)

[Glossary](#)

[Resources](#)

[Language generation](#)

[Adaptive expressions](#)

[.lu file format](#)

[.qna file format](#)

[.lg file format](#)

[FAQ](#)

[Composer FAQ](#)

[Power Virtual Agents](#)

[Overview](#)

[Reference](#)

[Memory variables](#)

# Introduction to Bot Framework Composer

5/20/2021 • 5 minutes to read

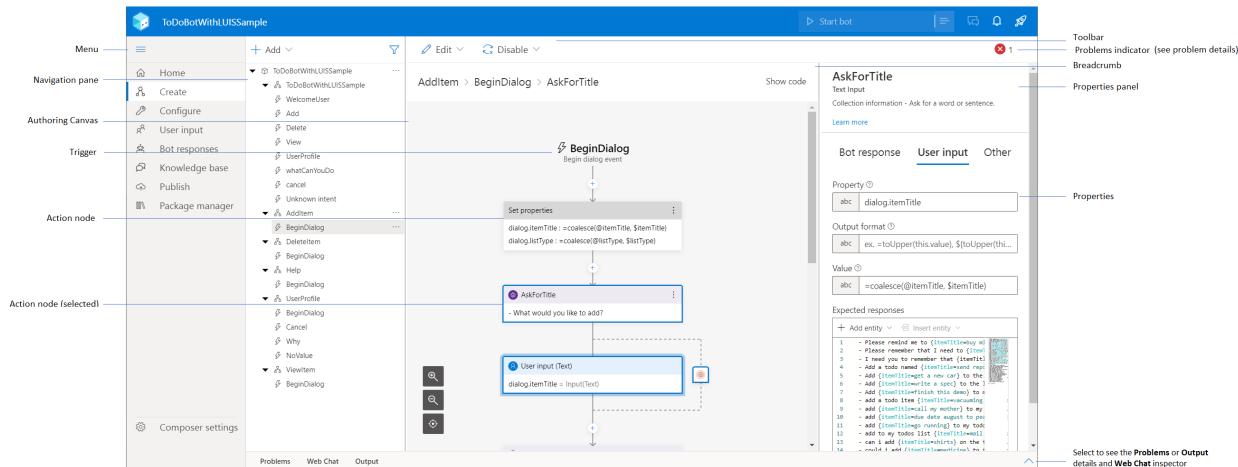
APPLIES TO: Composer v1.x and v2.x

Bot Framework Composer, built on the Bot Framework SDK, is an open-source IDE for developers to author, test, provision and manage conversational experiences. It provides a powerful visual authoring canvas enabling [dialogs](#), [language-understanding models](#), [QnAMaker knowledgebases](#) and [language generation](#) responses to be authored from within one canvas and crucially, enables these experiences to be extended with code for more complex tasks such as system integration. Resulting experiences can then be tested within Composer and provisioned into Azure along with any dependent resources.

Composer is available as [a desktop application](#) for Windows, OSX and Linux as well as [a web-based component](#) which can be customized and extended to suit your needs.

Authoring dialog experiences with a visual designer is more efficient and enables easier modeling of more sophisticated conversational experiences where context switching, interruption and more natural and dynamic conversation flows are important. More complex activities such as integrating with dependencies such as REST Web Services are best suited towards code and we provide an easy mechanism to extend Composer bots with code bringing the best of both together.

- [Composer v2.x](#)
- [Composer v1.x](#)



## What you can do with Composer

Composer is a visual editing canvas for building bots. You can use it to do the following:

- [Create a new bot using a template](#). This now incorporates the Virtual Assistant capabilities directly into Composer and new templates.
- Add NLU capabilities to your Bot using [LUIS](#) and/or QnA/FAQ type capabilities using [QnA Maker](#)
- Author text and if needed speech variation responses for your Bot using [language generation](#) templates
- Author bots in [multiple languages](#)
- [Test](#) directly inside Composer using embedded Web Chat
- [Publish bots](#) to *Azure App Service* and *Azure Functions*
- [Extend Power Virtual Agents with Composer \(Preview\)](#)
- Integrate external services such as [QnA Maker knowledge base](#)

Beyond a visual editing canvas, you can use Composer to do the following:

- Import and export dialog assets to share with other developers
- [Package manager](#) provides a range of re-usable conversational assets and code built by Microsoft and 3rd parties that quickly add pre-built functionality to your project.
- [Make any Bot available as a Skill for other Bots to call](#)
- [Connect to a skill](#)
- Extend the dialog authoring canvas with [Custom Actions] (how-to-add-custom-action.md) ([C# | JavaScript Preview](#))
- Integrate [Orchestrator](#) which is an advanced transformer model-based router that can delegate from a parent bot to [skills](#) based on a users utterance.
- [Host Composer in the cloud](#)
- [Extend Composer with plugins](#)

Under the hood, Composer harnesses the power of many of the components from the Bot Framework SDK.

When building bots in Composer, developers will have access to:

### **Adaptive dialogs**

Dialogs provide a way for bots to manage conversations with users. [Adaptive dialogs](#) and the event model simplify sophisticated conversation modelling enabling more natural, dynamic conversation flow, interruption and context switching. They also help you focus on the model of the conversation rather than the mechanics of dialog management. Read more in the [dialog concept article](#).

### **Language Understanding (LU)**

LU is a core component of Composer that allows developers and conversation designers to train language understanding models directly in the context of editing a dialog. As dialogs are edited in Composer, developers can continuously add to their bots' natural language capabilities using the [.lu file format](#), a simple Markdown-like format that makes it easy to define new [intents](#), [entities](#) and provide sample [utterances](#). In Composer, you can use both Regular Expression, [LUIS](#) and [Orchestrator](#). Composer detects changes and updates the bot's cloud-based natural-language understanding (NLU) model automatically so it is always up to date. Read more in the [language understanding concept article](#).

- [Composer v2.x](#)
- [Composer v1.x](#)

## Create a trigger

What is the type of this trigger?

What is the name of this trigger?

Trigger phrases

Trigger phrases	
+ Add entity ▾ <span style="font-size: small;">Insert entity ▾</span>	
1	- How can I buy {ProductType=Surface PRO}
2	- I want to buy {ProductType=Surface PRO}
3	- I want to buy {ProductType=Surface laptop}
4	- Can I buy {ProductType=Surface PRO} online?

[Cancel](#)

[Submit](#)

Show code

## ProductsCatalog

Adaptive dialog

This configures a data driven dialog via a collection of events and actions.

[Learn more](#)

Language Understanding ⓘ

Recognizer Type

Auto end dialog ⓘ

y/n	true
-----	------

Default result property ⓘ

abc	dialog.result
-----	---------------

Dialogs ⓘ

...
-----

> Dialog Interface

## Language generation (LG)

Creating grammatically correct, data-driven responses that have a consistent tone and convey a clear brand voice has always been a challenge for bot developers. Composer's integrated Language Generation (LG) that allows developers to create bot replies with a great deal of flexibility using the editor in the **Bot Responses** page or the **response editor** in the **Properties** panel. Read more in the [language generation](#) concept article.

- [Composer v2.x](#)
- [Composer v1.x](#)

The screenshot shows the Microsoft Bot Framework Composer interface. The left sidebar has a navigation menu with options like Home, Create, Configure, User input, Bot responses (which is selected), Knowledge base, Publish, Package manager, and Composer settings. The main area is titled "Bot Responses" and shows a table of responses for the "AskingQuestionsSample" bot. The table columns are Name, Responses, and Been used. There are four rows in the table:

Name	Responses	Been used
#SendActivity_068558	- \${WelcomeUser()}	✓
#SendActivity_581197	- \${WelcomeUser()}	✓
#SendActivity_xh61dm	- \${WelcomeUser()}	✓

At the bottom of the table, there is a button labeled "+ New template". Below the table, there are three tabs: Problems, Web Chat, and Output.

With Language Generation, you can achieve previously complex tasks easily such as:

- Including dynamic elements in messages.
- Generating grammatically correct lists, pronouns, articles.
- Providing context-sensitive variation in messages.
- Creating Adaptive Cards attachments, as seen above.
- Provide speech variations for each response, including SSML modifications which are key for speech based experiences such as telephony

## QnA Maker

[QnA Maker](#) is a cloud-based service that enables you to extract Question and Answer pairs from existing FAQ-style documents and websites into a knowledgebase that can be manually curated by knowledge experts. QnAMaker once integrated into a bot can be used to find the most appropriate answer for any given natural language input, from your custom knowledgebase (KB) of information.

### Bot Framework Emulator

[Emulator](#) is a desktop application that allows bot developers to test and debug bots built using Composer. This tool allows for more advanced scenarios (e.g., Authentication), which Composer's integrated WebChat feature doesn't support at this time.

## Advantage of developing bots with Composer

Some of the advantages of developing bots in Composer include:

- Authoring dialogs using the visual canvas can be more conducive to a conversational design versus code and enables you to focus development efforts on more complex tasks such as system integration.
- Design conversational experiences using a seamless blend of visual and code authoring.
- Existing dialogs authored in code can be leveraged by a Composer based bot
- Language Generation (LG) provides the ability to create more natural, personalized responses resulting in engaging conversational experiences.
- Composer streamlines your bot project's codebase and provides a more accessible visual design surface which provides a unified canvas to author dialogs, responses along with language and QnA resources.
- Integrated testing within the Composer authoring experience
- Azure provisioning for dependent resources is streamlined as part of the overall Composer experience

Apps created with Composer uses the Adaptive dialog format, a JSON specification shared by many tools provided by the Bot Framework.

The Composer bot projects contain reusable assets in the form of JSON and Markdown files that can be bundled and packaged with a bot's source code. These can be checked into source control systems and deployed along with code updates, such as dialogs, language understanding (LU) training data, and message templates.

Existing SDK-first bots can make use of Composer for new capabilities while still making use of existing waterfall dialogs, see [Migrating a bot built with the v4 SDK and waterfall dialogs to Bot Framework Composer](#) for more information. In addition, [Skills](#) are a technique that can be used to combine different bots together.

## Additional resources

- [Bot Framework SDK documentation](#)
- [Adaptive dialog](#)
- [Language generation](#)
- [Adaptive expressions](#)
- [.qna file format](#)

## Next steps

- Read [best practices for building bots using Composer](#).
- Learn how to [create an echo bot](#) using Composer.

# What's new May 2021

5/20/2021 • 4 minutes to read

Bot Framework Composer, a visual authoring tool for building conversational AI applications, has seen strong uptake from customers and positive feedback since entering general availability at Microsoft Build 2020. We continue to invest in ensuring Composer provides the best possible experience for our customers.

Welcome to the May 2021 release of Bot Framework Composer. This article summarizes key new features and improvements in the Bot Framework Composer [2.0 stable release](#). There are a number of updates in this version that we hope you will like.

## New User and Getting Started Experience

Composer has a New User and Getting Started Experience that makes it easier than ever to create bots as well as assess what they need to run.

### Templates

Developers can now create bots using templates, which are available from the Composer home screen. These templates use packages and take a dependency on the Adaptive Runtime. The runtime is now separated from your bot's code and added as a package reference, allowing you to focus on extending your bot without ever editing the runtime. Below is a screenshot of available C# templates.

### Select a template



Microsoft's templates offer best practices for developing conversational bots.

The screenshot shows the 'Select a template' dialog in Bot Framework Composer. On the left, a sidebar lists several C# templates: Empty Bot, Core Bot with Language, Core Bot with QnA Maker, Core Assistant Bot, Enterprise Assistant Bot, Enterprise Calendar Bot, and Enterprise People Bot. The 'Empty Bot' template is selected and highlighted in grey. On the right, detailed information for the 'Empty Bot' template is displayed:

- Empty Bot** 1.0.0-rc15
- A simple bot with a root dialog and greeting dialog.
- Recommended use**
  - Start from scratch, with a basic bot without additions
  - Good for first time bot developers, or seasoned pros
- Included capabilities**
  - Greeting new and returning users
- Required Azure resources**
  - This template does not rely on any additional Azure resources

At the bottom left is a link to 'Need another template? Send us a request'. At the bottom right are 'Cancel' and 'Next' buttons.

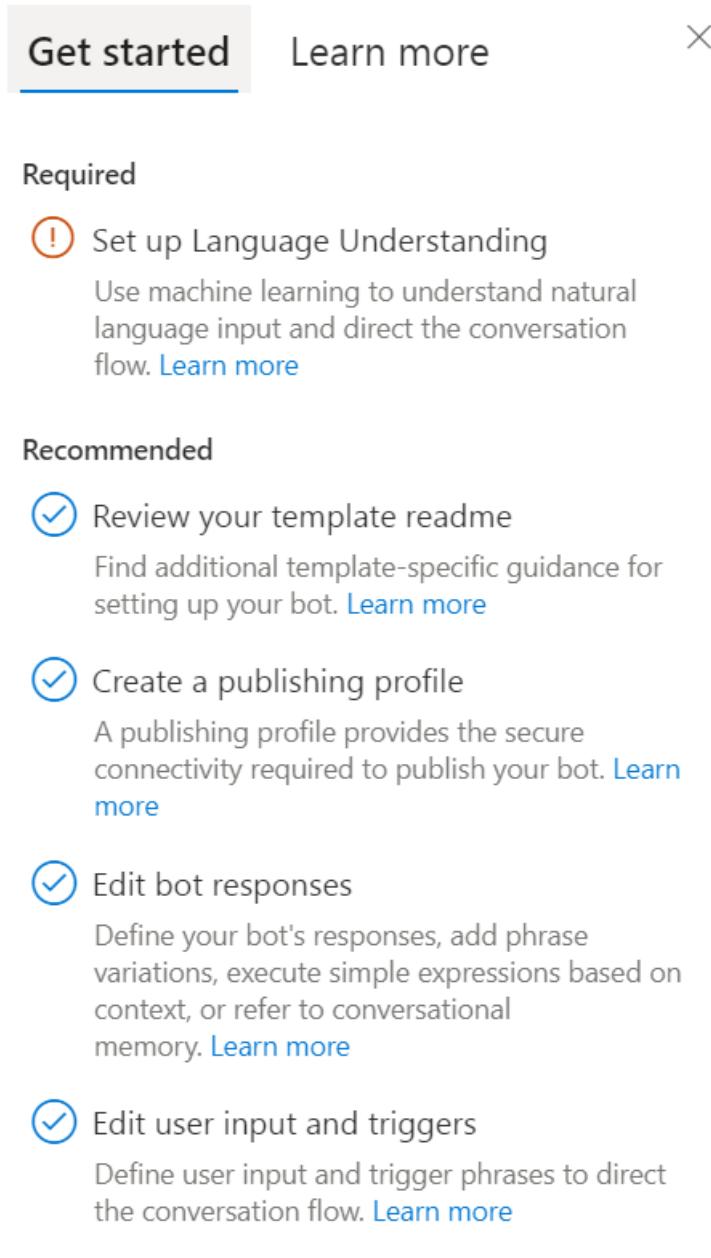
These templates bootstrap your development by providing pre-built dialogs and natural language models, support for common conversational scenarios, such as interruption, context switching, and even integration with Microsoft 365. Note that Node templates are still in preview.

The Enterprise Assistant template provides a starting point for those interested in creating a virtual assistant for common enterprise scenarios. With Composer, you have the flexibility to personalize the assistant to reflect the values, brand, and tone of your company.

For more information, see the [Enterprise Assistant overview](#) and [tutorial](#).

### Getting Started Experience

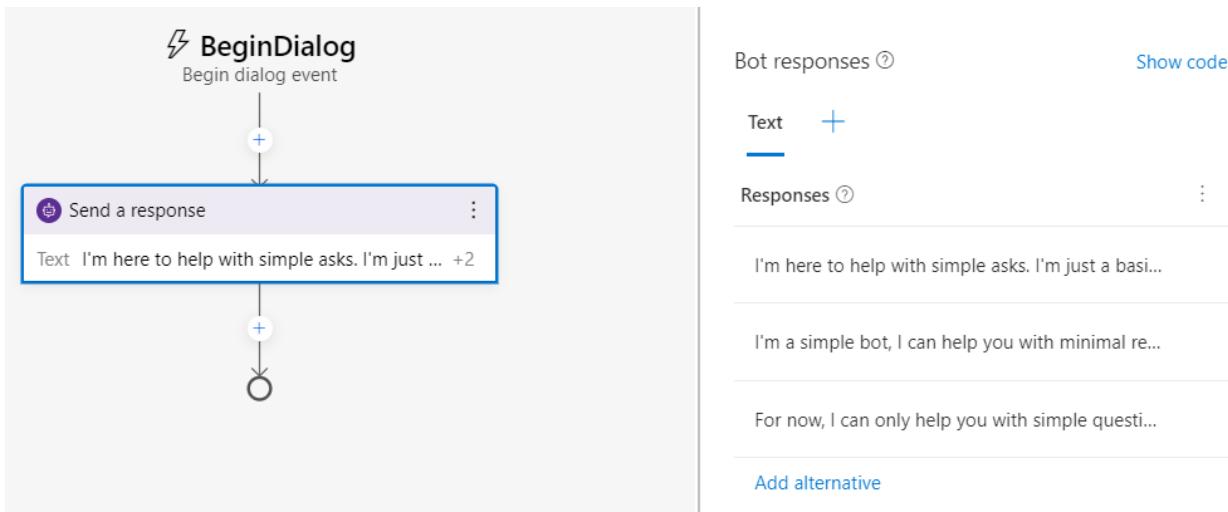
After creating a bot from a template, the Getting Started Experience assists you with what you need to add to your bot, with associated documentation to help you along the way. The example below is what the **Get Started** menu looks like after creating a bot using the C# Core Bot with Language template.



Shown will be any **Required**, **Recommended**, and **Optional** steps recommended when working with a specific type of bot.

## LG and LU authoring improvements

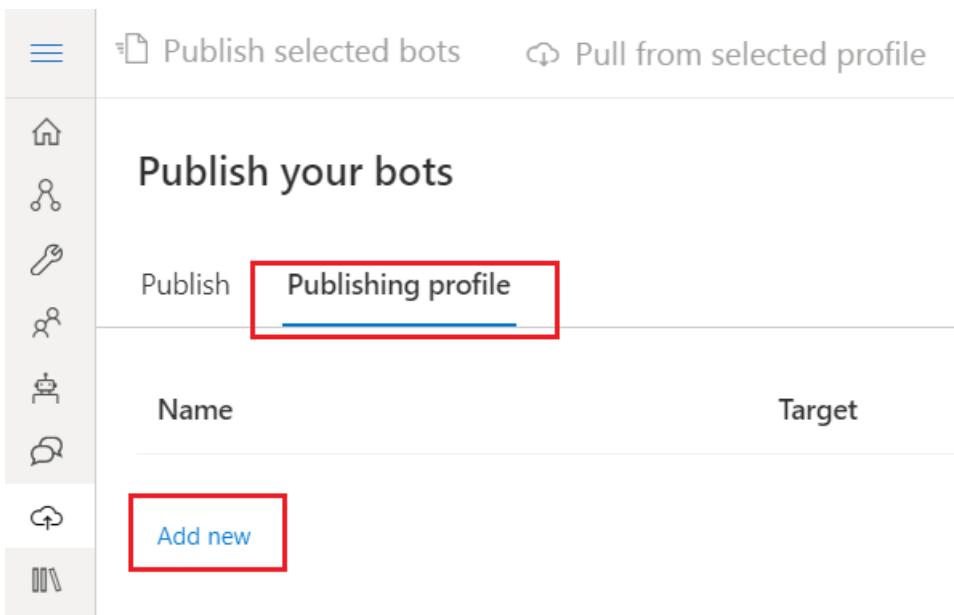
For this release, the UI for **Send a response** and **Ask a question** nodes (on the design canvas) have been updated to show details of the response content, instead of the activity template syntax. Users also have the ability to use an expanded editing area when editing LG in the code response editor, or editing attachment content.



Additionally, when adding Trigger phrases, along with User input sections (like in **Ask a question** actions), users are now presented with helpers to allow them to insert pre-built or machine learned entity definitions. You're also now able to tag one or more words within an utterance with an entity that has been defined, or simply insert an entity definition.

## Provisioning and publishing

This release improves on the built-in publish and provision capabilities of Composer. Users can now create a **publishing profile**, which they can provision new or existing resources to. After creating a publishing profile, you can easily deploy your bot to Azure.



See the [Provision Azure resources](#) and [Publish a bot to Azure](#) articles for more information.

Additionally, users can request help from an Azure administrator to create a publishing profile on their behalf. For more information about Azure administrators provisioning resources, see the [Admin handoff for provisioning Azure resources](#) section.

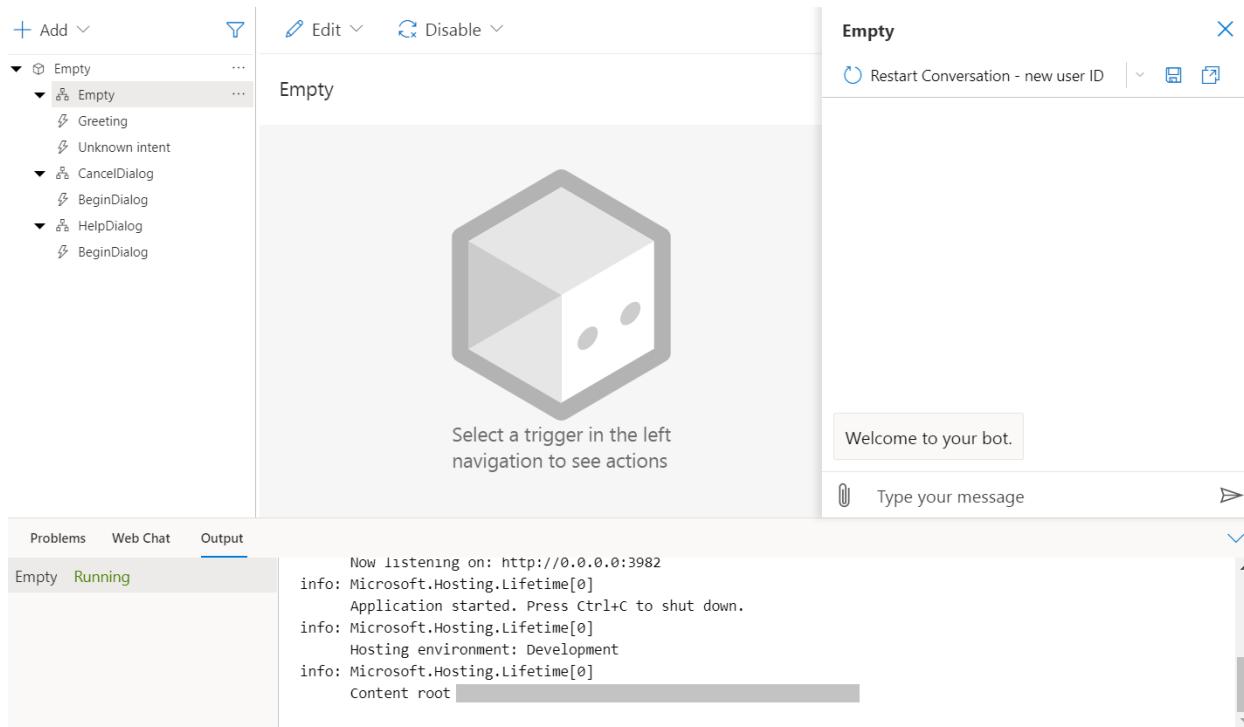
### Create a CI/CD pipeline (preview)

Developers can also create a CI/CD pipeline to manage their bot's updates and deployment. See the [Create a CI/CD pipeline \(Preview\)](#) article for more information.

## Testing and debugging

Testing and debugging features were added this release that make it easier than ever to assess and fix your bots.

Users can use **Web Chat** in the Composer UI, and can now inspect the **Output** while their bots run.



See the [Debugging and validation](#) and [Test and debug bots using Bot Framework Web Chat](#) articles for more information.

## Create, publish, and consume your own packages

This release marks the general availability of the component model, a key pillar of which is working with packages. Users can now create their own packages, publish them to NuGet, consume them in bots, update them, and remove them.

A screenshot of the Microsoft Bot Framework Package Manager interface. On the left is a sidebar with icons for Home, Components, Bot, Chat, and Help. The main area has a title 'Package Manager' and a sub-header 'Discover and use components that can be installed into your bot. [Learn more](#)'. Below this is a search bar with 'nuget' typed in and a 'Search' button. A list of packages is shown, with 'Microsoft.Bot.Builder.Adapters.Twilio' at the top, followed by 'Microsoft.Bot.Builder.Adapters.Slack', 'Microsoft.Bot.Builder.Adapters.Facebook', 'Microsoft.Bot.Builder.Adapters.Webex', and 'Microsoft.Bot.Builder.AI.Orchestrator'. Each package entry includes a small icon, the package name, and a brief description. A note 'Select an item from the list to view a detailed description.' is displayed on the right. A scroll bar is visible on the right side of the main content area.

See the [Understanding packages](#), [Creating packages](#) and [Managing packages for your bot](#) articles for more information.

## Orchestrator

Orchestrator is an alternative approach to consuming language understanding (LU) data, and helps arbitrate between multiple LUIS and QnA Maker applications to route user input to an appropriate skill or to subsequent language processing services.

Orchestrator is now out of preview and available as a [package](#) in the Package manager. Once the package is added, the **Orchestrator** recognizer will appear as a **Recognizer type** in Composer.

See the [Orchestrator](#) article for more information.

## Azure Bot Service handoff

You can now create Composer bot resources in the Azure portal and open Composer from Azure.

See the [Create a bot in Azure](#) article for more information.

### **Admin handoff for provisioning Azure resources**

An Azure administrator can also provision Azure resources for users who don't have access to Azure.

See the following articles for more information:

- [Admin Handoff for LUIS](#)
- [Admin Handoff for QnA](#)
- [Admin Handoff for Speech](#)

## Skills

There were a number of skills related improvements this release. The skill connection experience was simplified, and less questions are asked now, allowing the automation of wiring up your skill to reduce developer complexity. You can now add a remote skill to your bot project, create a root bot which calls a skill, and create, add, and update a local skill.

See the [Create a local skill](#), [Connect to a remote skill](#), and [Add single sign on for a skill](#) articles for more information.

## Additional information

- Read more in Composer's [2.0.0 release notes](#).
- Download the [nightly build](#) and try the latest updates as soon as they are available.

# Install Bot Framework Composer

6/18/2021 • 3 minutes to read

**APPLIES TO:** Composer v1.x and v2.x

Get started with Bot Framework Composer by installing the desktop application for your preferred operating system (OS). For more advanced scenarios where you wish to customize Composer, you can [build Composer from source](#).

Composer is optimized for `x64` architectures on Windows, macOS, and Linux. For more information, see [Recommended operating systems and architectures](#).

## Download Composer

Use the tabs below to select an operating system and download Composer.

- [Windows](#)
- [macOS](#)
- [Linux](#)

[Download Composer for Windows](#)

## Prerequisites

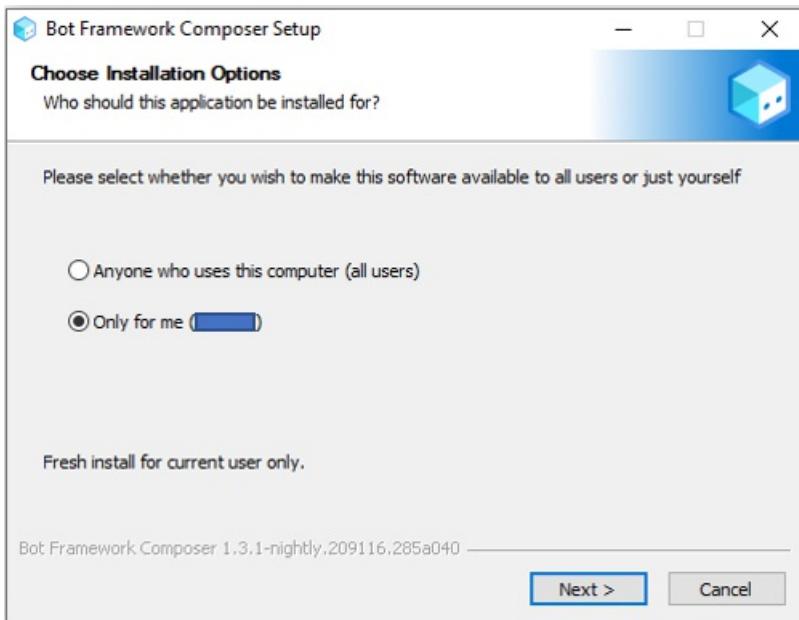
- [Node.js](#) LTS 12.x or 14.x is required.
- [.NET Core SDK 3.1](#) or later is required if you are building a bot using .NET.

When you create a new project, Composer will warn you if these aren't installed.

## Install and run Composer

- [Windows](#)
- [macOS](#)
- [Linux](#)

1. Launch the Composer installer. Then follow the instructions provided by the install wizard.
2. Choose who can access Composer. Then select **Next**.



3. Select a location to install Composer. Then select **Install**. This will take a few minutes.
4. Select **Finish** to close the installation wizard.
5. Select the Composer icon to start the application and enjoy your bot-building journey!

## Enable auto updates

You can enable auto updates from Composer's **Application Settings** page. When auto updates are enabled, Composer will check for and install updates each time it's closed. Follow these steps to enable auto updates:

1. Select **Composer settings** from the main menu, then select **Application Settings**.
2. Locate **Auto update** in the **Application updates** section. Set the toggle to **On**.
3. That's it. Auto updates are now enabled for Composer.

If you'd like to learn more about Composer settings, see [Application settings](#).

## Manually update Composer

If you prefer to manually update Composer, follow these steps:

- [Windows](#)
- [macOS](#)
- [Linux](#)

1. To check for updates, select **Help** from Composer's top menu bar, then select **Check for updates**.
2. The **New update available** window shows the most recent Composer version. There are two update options available: **Install the update and restart Composer** and **Download now and install when you close Composer**. Select the option that meets your needs.
3. After you see the **Update complete** window, select **Okay** to reinstall and restart Composer.
4. Go through the installation wizard to install the new version of Composer on your computer.

If you have an older version of Composer and want to update to the latest stable release, see [Bot Framework Composer releases](#).

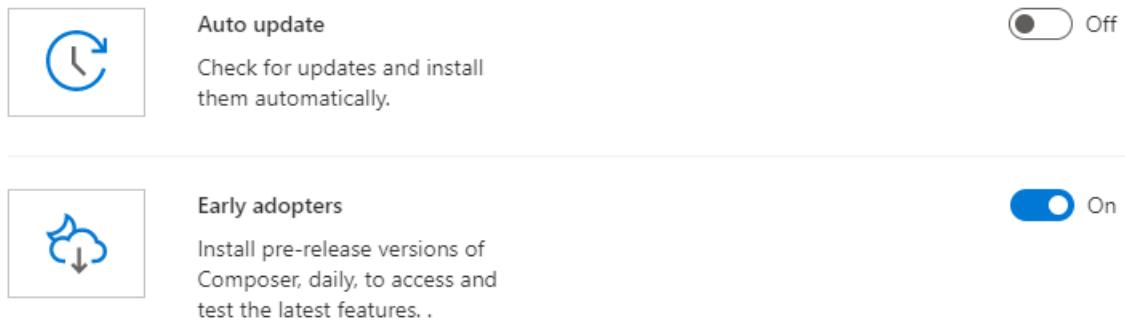
## Use nightly builds

You may want to try new features as soon as they are available, and you can do this with [nightly builds](#). Follow these instructions to check for and use the nightly builds:

- Windows
- macOS
- Linux

1. Select **Composer settings** from the main menu then select **Application Settings**.
2. In the **Application Updates** section, set **Early adopters** to **On**.

#### Application Updates



3. Go to the application's menu bar and select **Help**, then select **Check for Updates**. You will see the most recent nightly build available.
4. Follow the installation wizard to install the most recent nightly build of Composer on your computer.

## Next steps

[Create your first bot](#)

# How to build Bot Framework Composer from source

6/18/2021 • 2 minutes to read

You can download and run the Bot Framework Composer as a [desktop application on Windows, Linux, and macOS](#) or you can build it from source and run it as a web application. In this guide, you'll learn how to build Composer from source and run it locally as a web application using Yarn.

## Prerequisites

Make sure you have the following software installed before building Composer:

- [Git](#)
- [Node.js](#) LTS 14.x or 12.x
- [Yarn](#)
- [Bot Framework Emulator](#)
- [.NET Core SDK 3.1](#) or later

Composer is built to run on `x64` architectures. For a full list of supported operating systems, see [Supported operating systems and architectures](#).

## Build and run Composer

Follow these steps to build Composer from source and run it locally:

1. Clone the Composer repository:

```
git clone https://github.com/microsoft/BotFramework-Composer.git
```

2. Switch to the Composer directory:

```
cd BotFramework-Composer/Composer
```

3. Install dependencies:

```
yarn install
```

4. Build Composer with extensions and libraries:

```
yarn build
```

5. Start the client and server:

```
yarn startall
```

6. Open a browser and navigate to `http://localhost:3000`.

7. That's it, you're ready to use Composer.

## Next steps

[Create your first bot](#)

## See also

- [How to host Composer on Azure](#)

# Recommended operating systems and architectures

6/18/2021 • 2 minutes to read

The following tables show the recommended operating system (OS) versions and architectures for use with the Bot Framework Composer v1.3 or later. Composer is designed for `x64` architectures.

## Windows

OS	Version	Architectures
Windows 10	Version 1607+	x64
Windows Server	2012 R2+	x64
Nano Server	Version 1803+	x64

## macOS

OS	Version	Architectures
Mac OS X	10.14+	x64

## Linux

OS	Version	Architectures
Ubuntu	18.04, 20.04	x64
Debian	9+	x64

## Node.js (built from source)

If you are building Composer from source, the table below shows the Node.js versions that Composer will run on.

Node.js	Version	Architectures
LTS	v10, v12, v14	x64

## Next steps

[Install Bot Framework Composer](#)

# Application settings

5/28/2021 • 2 minutes to read

**APPLIES TO:** Composer v1.x and v2.x

The **Application Settings** page contains the settings you can customize for your Composer application. This article introduces each section of the **Application Settings** page and how to use them to customize your Composer application.

To access the **Application Settings** page:

1. Select **Composer Settings** from the Composer menu.
2. Select **Application Settings** from the **Navigation** pane.

SETTING	DESCRIPTION
<a href="#">Application Language settings</a>	Set the language to display your Composer user interface.
<a href="#">Property editor preferences</a>	This section contains settings of the property editors.
<a href="#">Application updates</a>	This section contains settings related to the preview features of Composer. When one or more preview features are selected, Composer will enable them for you to try.
<a href="#">Data Collection</a>	This setting enables the Composer team to collect Composer usage data. When enabled, Composer usage data can be used to help improve your bot development experience.

## Prerequisites

- Install [Composer](#)

## Application Language settings

Set the language in the **Application Language settings** section to view and experience the Composer application in a different language than English. English (US) is the language of Composer user interface by default. You can set the language settings back to English (US) anytime.

### Application Language settings

Composer language is the language of Composer UI [\(?\)](#)

A screenshot of a dropdown menu for selecting the application language. The menu is titled "Composer language is the language of Composer UI ()". It contains four options: "English (US)" (which is highlighted in blue), "English (US)" (disabled, shown in gray), "Čeština Czech" (disabled, shown in gray), and "Deutsch" (disabled, shown in gray). A small downward arrow icon is located at the top right of the menu.

## Property editor preferences

The **Property editor preferences** section contains settings to property editors.

## Property editor preferences

 **Minimap**  Off

A minimap gives an overview of your source code for quick navigation and code understanding.

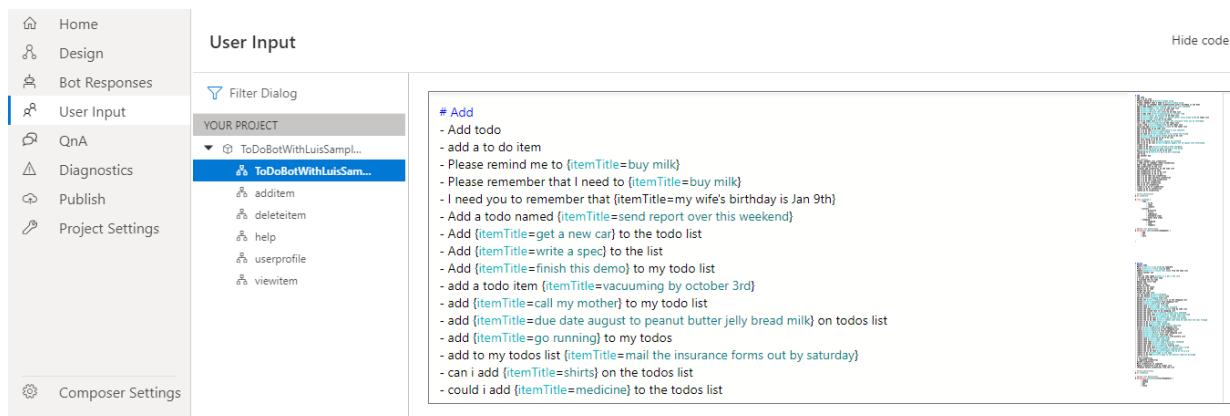
 **Line numbers**  Off

Enable line numbers to refer to code lines by number.

 **Sentence wrap**  Off

Display lines that extends beyond the width of the editor on the next line.

**Minimap:** When **Minimap** is turned **On**, a mini map will be enabled in the **Bot Responses** and **User Input** editors.



The screenshot shows the Microsoft Bot Framework Composer interface. On the left is a sidebar with navigation links: Home, Design, Bot Responses, User Input (which is selected), QnA, Diagnostics, Publish, and Project Settings. Below the sidebar is a "Composer Settings" button. The main area is titled "User Input". It contains a "Filter Dialog" section with a dropdown menu showing "YOUR PROJECT" and "ToDoBotWithLuisSampl...". Under this, there's a tree view with nodes like "# Add", "Add todo", "add a to do item", etc. To the right of the tree view is a large "minimap" window showing a grid of code snippets from the project. At the top right of the main area is a "Hide code" button.

**Line numbers:** When **Line number** is turned **On**, your LU and response editors will show the line numbers both in the inline editors, and in the editors in the **Bot Responses** and **User Input** screens.

### Trigger phrases

- 1 - help me
- 2 - please help
- 3 - I need help

### Language Generation ?

- 1 - This is a testing text.
- 2 - Testing.
- 3 - Hello, testing.

**Sentence wrap:** When **Sentence wrap** is turned **On**, the sentence wrap functionality will be enabled in the LU and response editors.

## Application Updates

The **Application Updates** section contains settings related to **Preview features**. Select one or more preview features to enable them in your Composer application. For more information on the **Preview features** currently available, select the **Learn More** link associated with that feature.



#### Preview features

Try new features in preview and help us make Composer better. You can turn them on or off at any time.

**Form dialogs.** Automatically generate dialogs that collect information from a user to manage conversations. [Learn More](#).

**New Creation Experience.** Preview the new bot creation experience. Create new bots that use the Adaptive Runtime, and can be enhanced using Package Manager. [Learn More](#).

## Data Collection

You can enable the telemetry feature from the **Data Collection** section. The telemetry information will help the Composer team to understand better how the tool is being used and how to improve it. To help us provide the usage data, toggle the switch to **On**.

### Data Collection

#### Data collection

 Off

Composer includes a telemetry feature that collects usage information. It is important that the Composer team understands how the tool is being used so that it can be improved.

## Next steps

- Learn [how to build an echo bot](#).

# Create your first bot with Composer

5/20/2021 • 4 minutes to read

In this quickstart you will learn how to create a basic bot in Composer and test it in Web Chat.

## Prerequisites

- [Install Bot Framework Composer](#)

## Create a bot

- [Composer v2.x](#)
- [Composer v1.x](#)

### Create a bot from a template

1. Open Composer.
2. Select **Create New (+)** on the homepage.
3. A list of [templates](#) are then shown which provide a starting point for your new bot. For the purposes of this quick-start, select the **Empty bot** template under the **C#** section. This template creates a bot containing a root dialog, initial greeting dialog, and an unknown intent handler. Then select **Next**.
4. Fill in the **Name** for the bot as *Menu\_bot*. Then select Azure Web App from the **Runtime type**, and a location for your bot on your machine.
5. Select **OK**. It will take a few moments for Composer to create your bot from the template.

### Add bot functionality

The bot created from the Empty bot template contains an unknown intent trigger. This acts as a fallback whenever your bot doesn't know how to respond. When your bot doesn't recognize the intent of user input, it will send a response letting the user know. To add an unknown intent trigger, do the following:

This section describes how to add basic bot functionality to your empty bot template. You will add:

- A [dialog](#) called **Menu** that displays a text message with menu items.
- A [trigger](#) for the **Menu** dialog. This makes it possible for your bot to recognize the intent, and connects the menu dialog above.

### Add a dialog

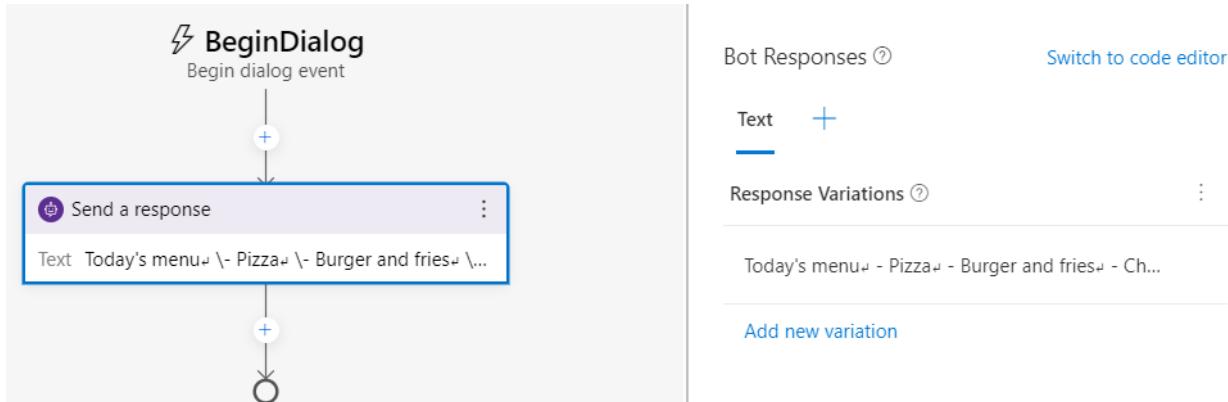
Now you can add functionality for intents that you want your bot to recognize.

Dialogs are a convenient way to organize conversational logic. This section shows you how to add a dialog. In the next section, you add a trigger to the bot's main dialog to call the new dialog.

1. Click the three dots next to your bot project, **Menu\_bot**, and select **+ Add a dialog**.
2. In the **Name** field, enter *Menu*, and in the **Description** field, enter *A dialog that shows the menu..* Then select **OK**.
3. Select **BeginDialog** underneath the **Menu** dialog.
4. In the authoring canvas, select the **+** button under **BeginDialog**. Then select **Send a response**.
5. Enter the following menu text in the response editor on the right:

```
Today's menu
- Pizza
- Burger and fries
- Chicken sandwich
```

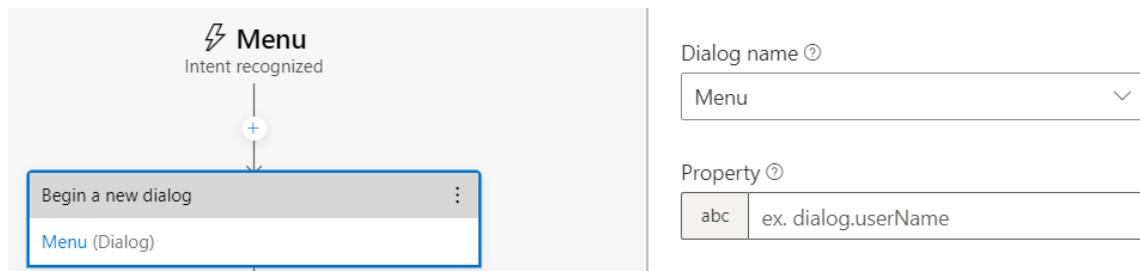
Your authoring canvas should look like the following:



#### Create a trigger to recognize the menu intent

Now that your dialog is ready to send a response of menu options, you need to create a trigger so that your bot recognizes when users want the menu item displayed.

1. Select your bot project, **Menu\_bot**.
2. On the right, select the **Regular expression recognizer** for the **Recognizer type**. This recognizer provides a simple example without adding natural language understanding. For a real-world bot, explore [language understanding](#).
3. Now, add a trigger to the **Menu\_bot** dialog.
  - a. The default trigger type is **Intent recognized**.
  - b. Enter **Menu** in the **What is the name of this trigger (RegEx)** field.
  - c. Enter **menu** in the **Please input regEx pattern** field.
  - d. Select **Submit**. This creates a trigger, named **Menu**, which will fire when the user sends a *menu* message to the bot.
4. Now you need to start the **Menu** dialog you created previously from the trigger. Select the **+** under the **Menu** trigger on the authoring canvas. Go to **Dialog management** and select **Begin a new dialog**.
5. Go to the right and select the box underneath **Dialog name**. You should see the **Menu** dialog you created previously; select it. Your authoring screen should look like the following:



Your menu bot is now ready to test!

## Test

- [Composer v2.x](#)
- [Composer v1.x](#)

1. Select the **Start bot** button from the top right of Composer. The **Local runtime manager** will appear once the bot has finished building, seen below:

### Local bot runtime manager

Start and stop local bot runtimes individually.

Bot	Status	
 weather_bot	Running	 Open Web Chat  Test in Emulator

2. Select **Open in Web Chat**. The Web Chat panel will appear on the right.

3. Try testing different phrases. Notice that your bot will respond with the message in the **Menu** dialog if your response includes the word *menu*. Otherwise, the bot will display one of the responses from the **Unknown intent** trigger.

Welcome to your bot.



Type your message



Congratulations! You've successfully created an echo bot!

## Next Steps

- Create a [weather bot](#) using Composer.

# Create a bot with Azure

5/20/2021 • 3 minutes to read

**APPLIES TO:** Composer v2.x

This article shows how to create a bot using the **Azure Bot** resource.

## Prerequisites

- [Bot Framework Composer](#)
- [Azure account](#)

## Create an Azure Bot resource

The Azure bot resource provides the infrastructure that allows a bot to access secured resources. It also allows the user to communicate with the bot via several channels such as Web Chat.

1. Go to the [Azure portal](#).
2. In the right pane, select **Create a resource**.
3. In the search box enter *bot*, then press **Enter**.
4. Select the **Azure Bot** card.



Azure Bot

Microsoft

Azure Service

Build enterprise-grade conversational  
AI experiences with Bot Framework  
Composer or SDK.

Create ▾



5. Select **Create**.



**Azure Bot** Add to Favorites

Microsoft

0.0 (0 ratings)

**Create**

6. Enter the required values.

## Create an Azure Bot

Basics Tags Review + create

**Project details**

Select the subscription to manage deployed resources and costs. Use resource groups like folders to organize and manage all your resources.

Bot handle \*

Subscription \*

Resource group \*  [Create new](#)

**Pricing**

Select a pricing tier for your Azure Bot resource. You can change your selection later in the Azure portal's resource management. Learn more about available options, or request a pricing quote, by visiting the [Azure Bot Services pricing](#).

Pricing tier

**Microsoft App ID**

A Microsoft AppID is required to create an Azure Bot resource. An App ID can be automatically created below, or you can manually create your own, then return here to input your new App ID and password.

[Manually create App ID](#)

 The app secret will be stored in Azure Key Vault in the same resource group as your Azure bot. [Learn More](#).

Microsoft App ID  Create new Microsoft App ID  Use existing app registration

[Review + create](#) [< Previous](#) [Next : Tags >](#)

7. Select **Review + create**.

8. If the validation passes, select **Create**. You should see the **Azure Bot** and the related key vault resources listed in the resource group you selected.

9. Select **Open in Composer**.



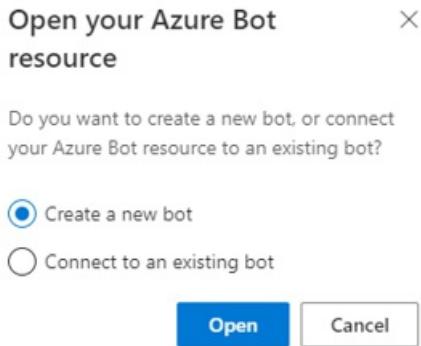
### Get started with Composer

Download Bot Framework's visual authoring canvas for developers, Composer. Open your bot directly in Composer and start developing.

 [Open in Composer](#)

The Composer application opens. If the application is not installed, you will be asked to install it before you can proceed with the next steps.

1. In the pop-up window, select **Create a new bot**.



2. Select **Open**.

# Create a bot

Create a bot by following the steps described below.

## Create a bot from a template

1. Open Composer.
2. Select **Create New (+)** on the homepage.
3. A list of [templates](#) are then shown which provide a starting point for your new bot. For the purposes of this quick-start, select the **Empty bot** template under the **C#** section. This template creates a bot containing a root dialog, initial greeting dialog, and an unknown intent handler. Then select **Next**.
4. Fill in the **Name** for the bot as *Menu\_bot*. Then select Azure Web App from the **Runtime type**, and a location for your bot on your machine.
5. Select **OK**. It will take a few moments for Composer to create your bot from the template.

## Add bot functionality

The bot created from the Empty bot template contains an unknown intent trigger. This acts as a fallback whenever your bot doesn't know how to respond. When your bot doesn't recognize the intent of user input, it will send a response letting the user know. To add an unknown intent trigger, do the following:

This section describes how to add basic bot functionality to your empty bot template. You will add:

- A [dialog](#) called **Menu** that displays a text message with menu items.
- A [trigger](#) for the **Menu** dialog. This makes it possible for your bot to recognize the intent, and connects the menu dialog above.

### Add a dialog

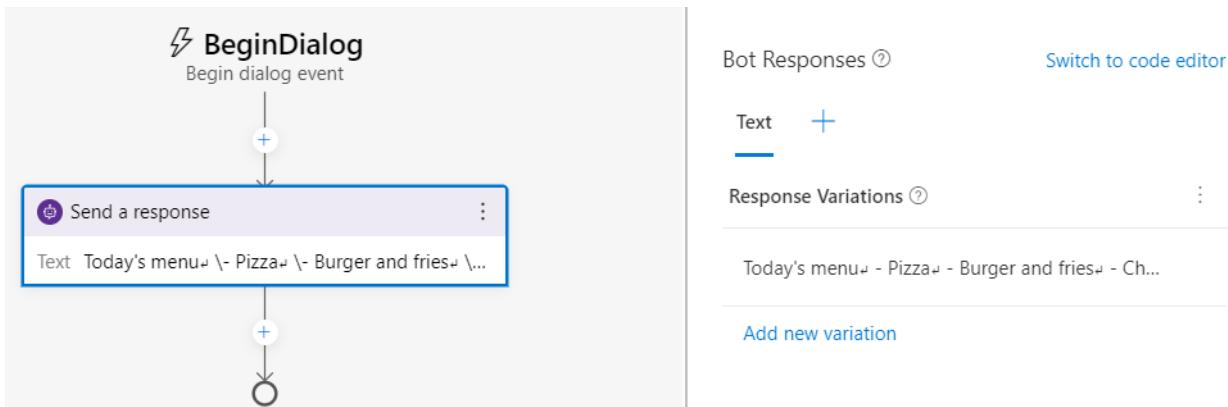
Now you can add functionality for intents that you want your bot to recognize.

Dialogs are a convenient way to organize conversational logic. This section shows you how to add a dialog. In the next section, you add a trigger to the bot's main dialog to call the new dialog.

1. Click the three dots next to your bot project, **Menu\_bot**, and select **+ Add a dialog**.
2. In the **Name** field, enter *Menu*, and in the **Description** field, enter *A dialog that shows the menu..* Then select **OK**.
3. Select **BeginDialog** underneath the **Menu** dialog.
4. In the authoring canvas, select the **+** button under **BeginDialog**. Then select **Send a response**.
5. Enter the following menu text in the response editor on the right:

```
Today's menu
- Pizza
- Burger and fries
- Chicken sandwich
```

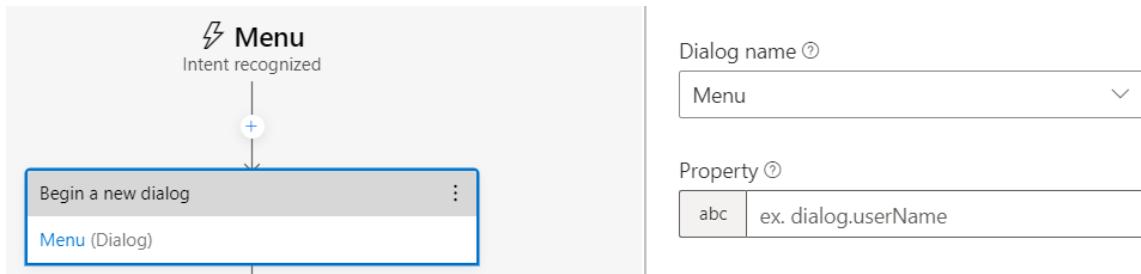
Your authoring canvas should look like the following:



#### Create a trigger to recognize the menu intent

Now that your dialog is ready to send a response of menu options, you need to create a trigger so that your bot recognizes when users want the menu item displayed.

1. Select your bot project, **Menu\_bot**.
2. On the right, select the **Regular expression recognizer** for the **Recognizer type**. This recognizer provides a simple example without adding natural language understanding. For a real-world bot, explore [language understanding](#).
3. Now, add a trigger to the **Menu\_bot** dialog.
  - a. The default trigger type is **Intent recognized**.
  - b. Enter `Menu` in the **What is the name of this trigger (RegEx)** field.
  - c. Enter `menu` in the **Please input regEx pattern** field.
  - d. Select **Submit**. This creates a trigger, named **Menu**, which will fire when the user sends a *menu* message to the bot.
4. Now you need to start the **Menu** dialog you created previously from the trigger. Select the **+** under the **Menu** trigger on the authoring canvas. Go to **Dialog management** and select **Begin a new dialog**.
5. Go to the right and select the box underneath **Dialog name**. You should see the **Menu** dialog you created previously; select it. Your authoring screen should look like the following:



Your menu bot is now ready to test!

## Next Steps

[Publish a bot to Azure](#)

# The Bot Framework Composer tutorials

5/20/2021 • 2 minutes to read

**APPLIES TO:** Composer v1.x and v2.x

Welcome to the Bot Framework Composer tutorials. You will start with the creation of a simple bot. With each successive tutorial you'll build on your bot by adding more capabilities. At the end of the tutorials, you will have a weather bot.

You will learn how to:

- [Create a simple bot and test it](#)
- [Add dialogs](#) to help your bot fulfill more than one scenario
- [Use prompts to ask questions](#) and get responses from an HTTP request
- [Handle interruptions](#) in the conversation flow in order to add global help and the ability to cancel at any time
- [Use response generation](#) to power your bot's responses
- [Send responses with cards](#)
- [Use language understanding in your bot](#)

## Prerequisites

A good understanding of the material covered in the [Introduction to Bot Framework Composer](#), including the naming conventions used for elements in the Composer.

## Next steps

[Create a bot](#)

# Tutorial: Create and test a bot

6/11/2021 • 5 minutes to read

**APPLIES TO:** Composer v1.x and v2.x

This tutorial shows you how to create a basic bot.

You will learn how to:

- Create a basic bot using Bot Framework Composer
- Run the bot locally and test it using the Bot Framework Emulator

## Prerequisites

- [Composer v2.x](#)
- [Composer v1.x](#)
- [Bot Framework Composer](#)

## Create the bot

- [Composer v2.x](#)
- [Composer v1.x](#)

The first step in creating a bot with Composer is to start a project from a template. This will create a new folder on your local computer with all the files necessary to build, test and run the bot.

1. Start Composer.
2. Select **+ Create new**.
3. Select the **C# Empty Bot** template. Then select **Next**.

## Select a template

X

Microsoft's templates offer best practices for developing conversational bots.

C# Node (Preview)



Core Bot with Language

Core Bot with QnA Maker

Core Assistant Bot

Enterprise Assistant Bot

Enterprise Calendar Bot

Enterprise People Bot



Empty Bot  
1.0.0-rc12

A simple bot with a root dialog and greeting dialog.

### Recommended use

- Start from scratch, with a basic bot without additions
- Good for first time bot developers, or seasoned pros

### Included capabilities

- Greeting new and returning users

### Required Azure resources

- This template does not rely on any additional Azure resources

[Need another template? Send us a request](#)

Cancel

Next

The Empty Bot template creates a bot with two triggers:

- Greeting: when a user connects to the bot, sends the user a greeting.
- Unknown intent: when the user sends a message the bot doesn't recognize, the bot sends a message telling the user it did not understand.

4. Now fill in the following values on the **Create your bot project** window:

- Name: weather\_bot
- Runtime type: Select Azure Web App
- Location: Select a location to store your bot.

5. Select **Submit** to create a bot from the template. Note that this may take a few moments.

After creating the bot, Composer will load the bot's main dialog.

#### NOTE

A dialog contains one or more **triggers** that define the actions available to the bot while the dialog is active.

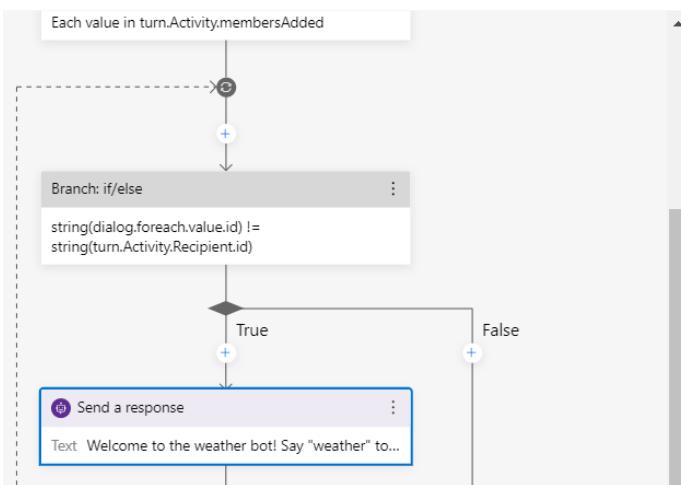
When you create a new bot, an **Activities** trigger of type **Greeting (ConversationUpdate activity)** is automatically provisioned. Triggers help your dialog capture events of interest and respond to them using actions.

6. Select the **Greeting** trigger on the left. Scroll to the bottom and select the **Send a response** action.

7. Select the text on the right (*Welcome to your bot*). Delete the text and replace it with the following:

Welcome to the weather bot! Say "weather" to get started.

Your screen should look like the following:



## Send a response

Send Activity

Respond with an activity.

[Learn more](#)

Bot responses [?](#)

[Show code](#)

Text [+](#)

Responses [?](#)

Welcome to the weather bot! Say "weather" to ...

[Add alternative](#)

Your bot is now ready to test.

## Test the bot

- [Composer v2.x](#)
- [Composer v1.x](#)

Now that your new bot has its first simple feature, you can launch it in Web Chat and verify that it works.

1. Select **Start bot** in the upper right corner of the screen. This tells Composer to launch the bot's runtime, which is powered by the Bot Framework SDK.

**▷ Start bot**

### NOTE

After selecting **Start bot**, the button text changes to **Starting bot**. Once the bot's runtime is running, it switches to **Restart bot**.

2. Once the bot's runtime has started, the **Local bot runtime manager** will open. You can also select the icon on the right of the text **Restart bot**, to open and close the **Local bot runtime manager**.
3. Select **Open Web Chat** in the **Local bot runtime manager**.

### Local bot runtime manager

Start and stop local bot runtimes individually.

Bot	Status	
weather_bot	Running	<a href="#">Open Web Chat</a> <a href="#">Test in Emulator</a>

4. The Web Chat pane will appear on the right, and you should be greeted with the message in the **Greeting** trigger. Try typing anything to activate the **Unknown intent** trigger.
5. When you're done, you can stop the bot by selecting the **Stop Bot** button in the lower left of the **Local bot runtime manager**.

You now have a working bot, and you're ready to add more functionality!

## Next steps

[Add dialogs to your bot](#)

# Tutorial: Add dialogs to your bot

5/28/2021 • 5 minutes to read

**APPLIES TO:** Composer v1.x and v2.x

This tutorial shows how to add dialogs to the bot you created in the [Tutorial: Create and test a bot](#) and test them either by using the Bot Framework Emulator or by opening Web Chat directly in Composer. (Web Chat is available in Composer versions 1.4 and higher.)

A dialog contains one or more [triggers](#). Each trigger consists of one or more actions which are the set of instructions that the bot will execute. Dialogs can also call other dialogs and can pass values back and forth between them.

## NOTE

It's useful to group functionality into different [dialogs](#) when adding features to your bot with Composer. This helps keep the dialogs organized and allows sub-dialogs to be combined into larger and more complex ones.

In this tutorial, you learn how to:

- Add dialogs to a basic bot.
- Run the bot locally and test it.

## Prerequisites

- Completion of the tutorial [Tutorial: Create and test a bot](#)
- Knowledge about [dialogs in Composer](#)

## What are you building?

The main function of the bot is to report current weather conditions.

To do this, you will create a dialog that:

- Prompts the user to enter a postal code to use as a location for weather lookup.
- Calls an external API to retrieve the weather data for the specified postal code.

## TIP

Create all of your bot components and make sure they work together before creating detailed functionality.

## Create a new dialog

- [Composer v2.x](#)
- [Composer v1.x](#)

1. Start Composer.
2. Select the **weather\_bot** bot project from the **Recent** bot list on the homepage.
3. Select the three dots next to your **weather\_bot** bot project. Then select **+ Add a dialog**.



Stop this bot

Export as skill

Export as .zip

Bot settings

4. In the pop-up window, enter the following:

a. **Name:** *getWeather*

b. **Description:** *Get the current weather conditions.*

**Create a dialog**

Specify a name and description for your new dialog.

**Name \***  
getWeather

**Description**  
Get the current weather conditions.

**Cancel** **OK**

5. Select **OK** to create the dialog.

6. Now select the **+** button under the **BeginDialog** dialog event in the center of the authoring canvas.

Select **Send a response**.

7. On the right in the response editor, enter the following:

Let's check the weather.

**BeginDialog**  
Begin dialog event

**Send a response**

Text Let's check the weather.

**Bot responses** **Show code**

**Responses**

Let's check the weather.

Add alternative

We will add more functionality later, like retrieving the weather forecast, but first we need to connect the **getWeather** dialog to the bot with a trigger.

## Connect the new dialog

You can break down a conversation flow into different dialogs and then connect them. The following steps explain how to connect the newly created **getWeather** dialog to the main dialog.

- [Composer v2.x](#)
- [Composer v1.x](#)

1. Select the **weather\_bot** dialog. Then go over to the right and change the **Recognizer Type** to **Regular expression recognizer**.

Language Understanding ⓘ

Recognizer Type

Regular expression recognizer

Auto end dialog ⓘ

y/n true

Default result property ⓘ

abc dialog.result

2. Now select the three dots next to the **weather\_bot** dialog on the left and select + Add new trigger.

+ Add new trigger

+ Add QnA Maker knowledge base

3. In the **Create a trigger** pop-up window, enter the following information:

- In both **What is the name of this trigger (RegEx)** and **Please input regex pattern** text boxes, enter *weather*.
- Select **Submit**.

#### Create a trigger

What is the type of this trigger?

Intent recognized

What is the name of this trigger (RegEx)?

weather

Please input regEx pattern

weather

Cancel

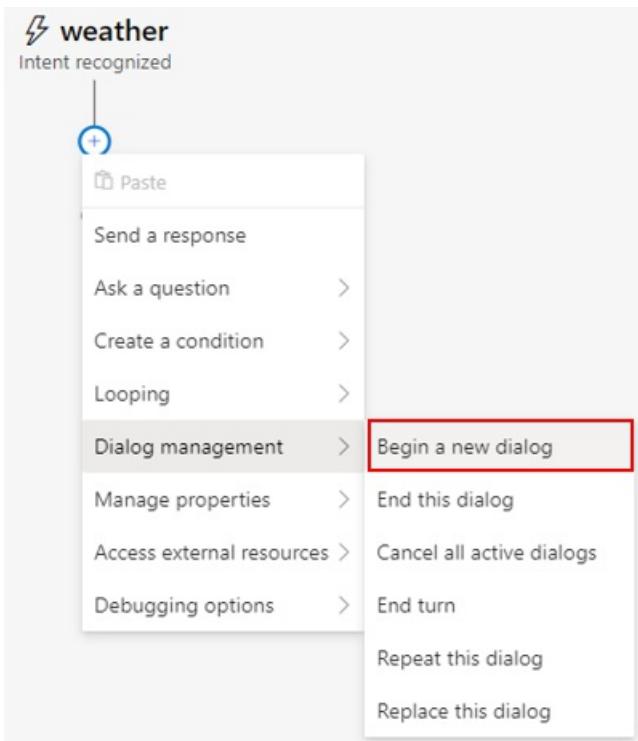
Submit

#### NOTE

This tells the bot to look for the word *weather* anywhere in an incoming message. Regular expression patterns are generally much more complicated, but this is adequate for the purpose of this example.

4. In the center of the authoring canvas, under the **weather** Intent recognized trigger, select the + button.

5. Hover over **Dialog management** and then select **Begin a new dialog**.



6. On the right, under **Dialog name**, select **getWeather**. Now your weather bot is connected to the

### Begin a new dialog

Begin dialog

Begin another dialog.

[Learn more](#)

Dialog name ⓘ

getWeather

**weather** trigger.

You can now test your bot, and the trigger and dialog you added to it.

## Test the bot

- [Composer v2.x](#)
- [Composer v1.x](#)

1. Go to the top right of Composer and select **Start bot**. It may take a few moments for your bot to start.

#### NOTE

If the bot is still running from the previous tutorial, you can select **Restart bot**. This will update the bot runtime app with all the new content and settings.

2. The **Local bot runtime manager** will open. Select **Open Web Chat**. The Web Chat pane on the right will

 Restart bot

## Local bot runtime manager

Start and stop local bot runtimes individually.

Bot	Status	
weather_bot	Running	 Open Web Chat

- appear.
- Now test some different phrases. Notice that the bot will send the message in the `getWeather` dialog if the word `weather` is in your response. Otherwise the bot will send the message in the `Unknown intent` trigger.

## Next steps

[Add actions to your dialog](#)

# Tutorial: Add actions to your dialog

5/28/2021 • 14 minutes to read

**APPLIES TO:** Composer v1.x and v2.x

This tutorial shows how to add actions to a dialog. This allows the bot to prompt a user for their postal code. The bot obtains the weather data by querying an external service, and it responds with the weather forecast for the location at the specified postal code.

In this tutorial, you will learn how to:

- Add an action to prompt the user for information
- Create properties with default values
- Save data into properties for later use
- Retrieve data from properties to accomplish tasks
- Make calls to external services

## Prerequisites

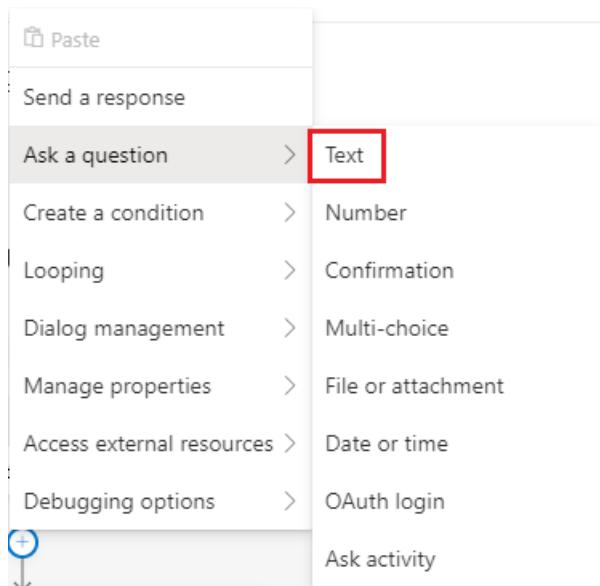
- Completion of the tutorial [Adding dialogs to your bot](#).
- Knowledge about [dialogs in Composer](#), specifically [actions](#).
- Knowledge about [conversation flow and memory](#).
- [OpenWeather website](#) API token. The [Add an HTTP request](#) section explains how to get this token.

## Get weather report

Before the bot can get the weather forecast, it needs to know the specific location for which the weather is requested. To do this, you create a **Text Input** action to prompt the user for a postal code. The bot then uses the code to obtain data from a weather service.

- [Composer v2.x](#)
- [Composer v1.x](#)

1. Start Composer.
2. Select the **weather\_bot** bot project from the **Recent** bot list on the homepage.
3. Select the **BeginDialog** dialog event under the **getWeather** dialog.
4. Now select the **+** button under the property you created. Hover over **Ask a question** and select **Text**.



After selecting **Text** from the **Ask a question** menu, you will notice that two new nodes appear in the *Authoring canvas*. Each node corresponds to a tab in the **Prompt for text** pane. See the image below.

- **Bot response** refers to the prompt issued by the bot to ask the user for information.
- **User input** lets you assign the user input to a property that is saved in memory and can be used by the bot for further processing.
- **Other** lets you validate the user input and to respond with a message if invalid input is entered.

See the [Asking for user input](#) article for more information about requesting and validating different data types.

5. In the **Bot response** section, select **Add alternative** under **Responses**. Enter the following:

The screenshot shows the Microsoft Bot Framework Authoring canvas. A conversation flow is visible with a 'Text' node containing 'Let's check the weather.' and a 'Prompt for text' node containing 'What's your postal code?'. The 'Prompt for text' node is selected, indicated by a blue border around its box. To the right, the 'Ask a question - text' pane is open, showing the 'Text' tab selected. The 'Responses' section contains the question 'What's your postal code?' and a link to 'Add alternative'.

This is the question that the bot will ask the user.

6. Now select **User input** on the right. Under **Property**, enter `user:postalcode`. This will save the user's postal code in memory to the `user.postalcode` property.
7. Under **Output format**, enter `=trim(this.value)`. `trim()` is a prebuilt function in [adaptive expressions](#). It trims any leading and trailing spaces in the user's input before it's validated and assigned to the property defined in the Property field.

## Prompt for text

Text Input

Collection information - Ask for a word or sentence.

[Learn more](#)

Bot response

User input

Other

Property ?

abc	user.postalcode
-----	-----------------

Output format ?

abc	=trim(this.value)
-----	-------------------

### NOTE

After the user enter the postal code the first time, the bot won't prompt for a value again. If you want the bot to ask for a new postal code, in the **Other** tab, you must set **Always prompt** to *true*.

## Set validation rules

This validation rule states that the user input must be 5 characters long. If the user input is shorter or longer than 5 characters the bot will send an error message, as defined in the **Invalid prompt** field.

- [Composer v2.x](#)
- [Composer v1.x](#)

1. Select the **Other** tab. This is where you specify validation rules for the prompt, as well as error messages displayed to the user if they enter an invalid value based on the **Validation Rules** you create.
2. Expand the **Recognizers** section. In the **Unrecognized Prompt** field, under **Responses**, select **Add alternative**. Enter the following:

Sorry, I do not understand '\${this.value}'. Please specify a 5 digit postal code in the format 12345.
--

3. Expand the **Validation** section. In the **Validation Rules** field enter the following:

`length(this.value) == 5`

This validation rule states that the user input must be 5 characters long. If the user input is shorter or longer than 5 characters the bot will send an error message, as defined in the **Invalid prompt** field.

### IMPORTANT

Make sure to press the Enter key after entering the validation rule. If you don't press Enter, the rule will not be added.

4. In the **Invalid prompt** field, under **Responses**, select **Add alternative**. In the text box enter the following:

Sorry, '\${this.value}' is not valid. I'm looking for a 5 digit number as postal code. Please specify a 5 digit postal code in the format 12345.
--

5. Expand the **Prompt Configurations** section. In the **Default value** field, enter **98052**.

**NOTE**

By default prompts are configured to ask the user for information the number of times specified in the **Max turn count** field (defaults to 3). When the *max turn count* is reached, the prompt will stop and the property will be set to the value defined in the **Default value** field before continuing the conversation.

You now have an action in your BeginDialog trigger that will prompt the user for their postal code and place it into the **userpostalcode** property. Next, you will pass the value of that property in an HTTP request to a weather service. If it passes your validation, the bot displays the weather report to the user.

### Add an HTTP request

In this section you will learn how to connect to a weather service by issuing an HTTP request. You will capture the response into a property, then determine what action to take depending on the results.

Before you go any further assure that you subscribed to the free version of the API at the [OpenWeather website](#). Once at the website, select API in the top menu bar. Then under **Current Weather Data** select the **Subscribe** button and follow the steps. You should get an API token to use in the HTTP Get request as shown in the steps below.

- [Composer v2.x](#)
- [Composer v1.x](#)

1. In the Authoring canvas, select the + button under the last action (**User input (Text)**). Hover over **Access external resources** and then select **Send an HTTP request**.
2. On the right, select **GET** from the **HTTP method** drop down menu.
3. In the **Url** field, enter the following address:

[http://api.openweathermap.org/data/2.5/weather?zip=\\${userpostalcode},us&appid=Your\\_API\\_Token](http://api.openweathermap.org/data/2.5/weather?zip=${userpostalcode},us&appid=Your_API_Token)

This lets your bot make an HTTP request to the specified URL. The reference to  **\${userpostalcode}** will be replaced by the value from the bot's **userpostalcode** property. You must replace *your-API-token* with the value you can get for free from the [OpenWeather website](#) as described before.



4. In the Result property box, enter *dialog.api\_response*.

Result property 

abc	dialog.api_response
-----	---------------------

Content type 

abc	Hello \${user.name}
-----	---------------------

Response type 

abc	json
-----	------

The **Result property** is where the result of this action will be stored. The result can include any of the following values from the HTTP response:

- *statusCode*: This can be accessed via `dialog.api_response.statusCode`.
- *reasonPhrase*: This can be accessed via `dialog.api_response.reasonPhrase`.
- *content*: This can be accessed via `dialog.api_response.content`.
- *headers*: This can be accessed via `dialog.api_response.headers`.

Set the **Response type** to **json**. If the **Response type** is JSON, the response will be a deserialized object available via the `dialog.api_response.content` property.

5. After making an HTTP request, you need to test the status of the response and handle errors as they occur. You can use an **If/Else branch** for this purpose. To do this, select plus (+) icon that appears beneath the **Send HTTP Request** action you created. Then select **Branch: If/else** from the **Create a condition** menu.
6. In the **Branch: If/else** properties pane on the right, in the **Condition** box, select **Write an expression** from the drop-down menu. An equal sign (=) will appear in the field. After the equal sign enter the value `dialog.api_response.statusCode == 200`. This is the expression to evaluate for the **if** statement.
7. In the **Authoring canvas**, under the **True** branch, select the plus (+) icon. In the drop-down menu, select **Manage properties**, and then **Set properties**.
8. In the **Set properties** pane on the right, for each property select the **Add new** button and enter the values described below.

PROPERTY	VALUE
<code>dialog.weather</code>	<code>=dialog.api_response.content.weather[0].description</code>
<code>dialog.icon</code>	<code>=dialog.api_response.content.weather[0].icon</code>
<code>dialog.city</code>	<code>=dialog.api_response.content.name</code>
<code>dialog.country</code>	<code>=dialog.api_response.content.sys.country</code>
<code>dialog.kelvin</code>	<code>=dialog.api_response.content.main.temp</code>
<code>dialog.fahrenheit</code>	<code>=round((9/5 * (dialog.kelvin-273)) + 32,0)</code>
<code>dialog.celsius</code>	<code>=round(dialog.kelvin-273.15)</code>

For information about the weather API and related JSON response, see [Current weather data](#).

9. While still in the **True** branch, select the plus (+) icon that appears beneath the action created in the

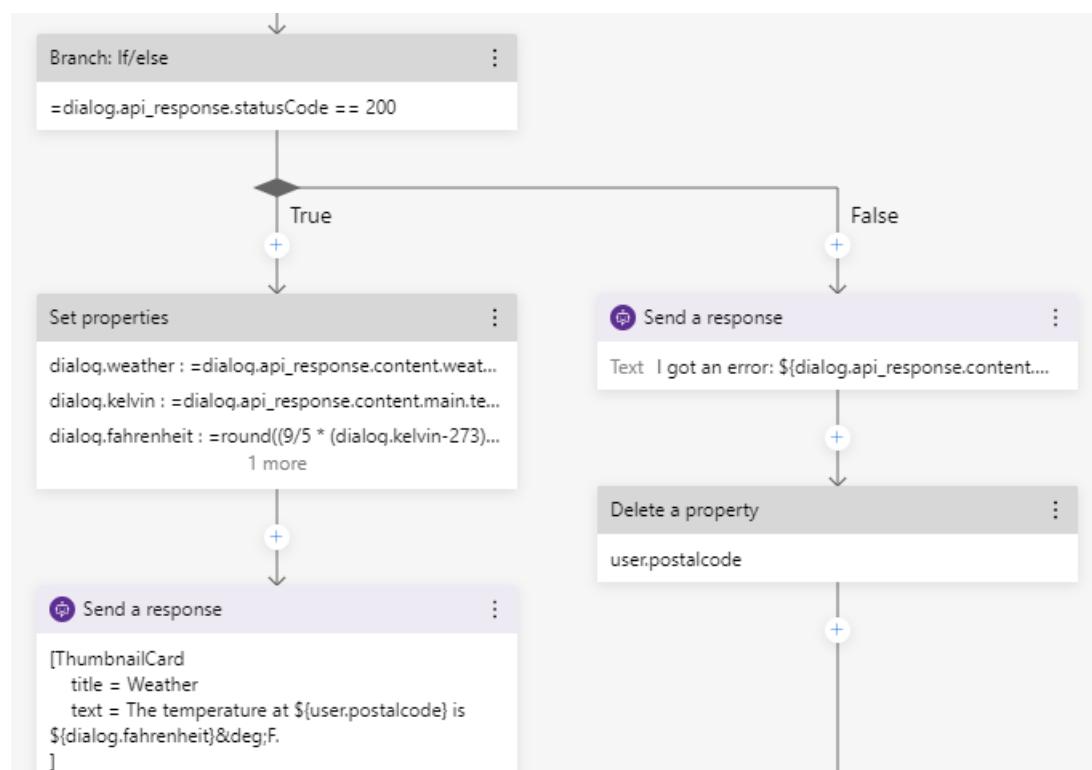
previous step. In the pop up menu, select **Send a response**.

10. In the **Send a response** pane on the right, enter the following response:

*The weather is \${dialog.fahrenheit}F or \${dialog.celsius}C and \${dialog.weather}*

Now, you should tell the bot what to do in the event that the **status code** returned is not 200.

11. Select the plus (+) icon in the **False** branch. From the pop-up menu, select **Send a response**.
12. In the **Send a response** pane on the right, in the text box enter *I got an error:*  
*/\${dialog.api\_response.content.message}*.
13. For the purpose of this tutorial we will assume that if you are in this branch, it's because the postal code is invalid, and if it's invalid it should be removed so that the invalid value doesn't persist in the **user.postalcode** property. To remove the invalid value from this property, select the + button that follows the **Send a response** action you created in the previous step. From the pop-up menu, select **Manage properties**, then **Delete a property**.
14. In the **Delete a property** pane on the right, enter **user.postalcode** into the **Property** field. The flow in the **Authoring canvas** should be as follows:



You have now completed adding an HTTP request to your **BeginDialog** trigger. The next step is to validate that these additions to your bot work correctly. To do that you can test it in Web Chat.

## Test

- [Composer v2.x](#)
- [Composer v1.x](#)

1. Select the **Restart bot** button in the upper right corner of the Composer interface.
2. In the **Local bot runtime manager**, select **Open Web Chat**.
3. After the greeting, send *weather* to the bot. The bot will prompt you for a postal code. Give it your home postal code, and seconds later, you should see the current weather conditions.

## Next steps

[Tutorial: Add Help and Cancel](#)

# Tutorial: Add Help and Cancel to your bot using interruptions

5/28/2021 • 8 minutes to read

**APPLIES TO:** Composer v1.x and v2.x

This tutorial shows you how to add interruptions to the conversation flow. Specifically, you can add help information to your bot and let users exit at any time during the conversation.

In this tutorial, you will learn how to

- Create help topics that can be accessed from anywhere in the flow at any time.
- Interrupt your bot's conversation flow to let users exit out of any process.

## Prerequisites

- Completion of the tutorial [Add actions to your dialog](#).

## Add Help and Cancel

With even a simple bot, it is a good practice to provide help. You'll also want to provide a way for users to exit at any point in the flow.

- [Composer v2.x](#)
- [Composer v1.x](#)

### Install the Help and Cancel package

1. Select the **Package Manager** icon on the left.



2. In the search box, enter **help**. Then select the **Microsoft.Bot.Components.HelpAndCancel** package.
3. On the right, select the **Install** button to install the package.

## Package Manager

Discover and use components that can be installed into your bot. [Learn more](#)

The screenshot shows the Azure Bot Service Package Manager interface. At the top, there are two tabs: 'Browse' and 'Installed'. Below them is a search bar with the word 'nuget' in it. To the right of the search bar is a button labeled 'help' with a magnifying glass icon, which is highlighted with a red box. On the far right of the header is a blue button labeled 'Install 1.0.0 | v'. Below the header, a package named 'Microsoft.Bot.Components.HelpAndCancel' is listed. It includes a description: 'Contains Adaptive Dialog assets to support Help and Cancel conversational flows in a b...', a Microsoft logo, and a link to 'View documentation'.

4. The **Project Readme** will appear on the screen. Read through it and then select **OK** on the bottom right. The package may take a few moments to install.

5. Now go back to your authoring canvas by selecting the **Create** icon on the left.

### Connect the Cancel dialog

1. Select the **Create** icon on the left to go back to the authoring canvas. You will notice two dialogs were added to your bot from the package.
2. Select the three dots next to your **weather\_bot** root dialog. Then select **+ Add new trigger**.
3. Enter the following values:

- a. **What is the name of this trigger (RegEx):** *cancel*
- b. **Please input regEx pattern:** *cancel|stop|quit*

## Create a trigger

What is the type of this trigger?

Intent recognized

What is the name of this trigger (RegEx)

cancel

Please input regEx pattern

cancel|stop|quit

Cancel

Submit

The regular expression is written so that the bot will recognize the cancel trigger whenever a user responds with *cancel*, *stop*, or *quit*.

4. Select **Submit** to create the new trigger.
5. Select the **+** button underneath the **cancel** Intent recognized trigger. Then hover over **Dialog management** and select **Begin a new dialog**.
6. On the right, select the **CancelDialog** under **Dialog name** to connect the **CancelDialog** to the **cancel** trigger.

### Connect the Help dialog

1. Select the three dots next to your **weather\_bot** root dialog. Then select **+ Add new trigger**.
2. Enter the following values:

- What is the name of this trigger (RegEx): *help*
- Please input regEx pattern: *help/support/advice*

## Create a trigger

What is the type of this trigger?

Intent recognized

What is the name of this trigger (RegEx)

help

Please input regEx pattern

help|support|advice

Cancel

Submit

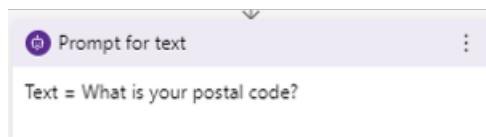
The regular expression is written so that the bot will recognize the cancel trigger whenever a user responds with *help*, *support*, or *advice*.

- Select **Submit** to create the new trigger.
- Select the + button underneath the **help** Intent recognized trigger. Then hover over **Dialog management** and select **Begin a new dialog**.
- On the right, select the **HelpDialog** under **Dialog name** to connect the **HelpDialog** to the **help** trigger.

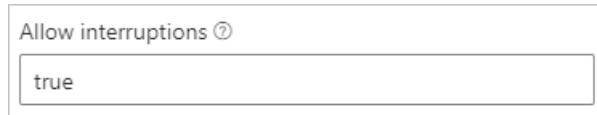
### Allowing interruptions

The **getWeather** dialog knows how to get the weather forecast, but it doesn't know how to respond to requests for help. To enable the bot to respond to requests for help while in the **getWeather** dialog, you will need to configure it to handle *interruptions*. This will let you call the new help functionality from the **getWeather** dialog, and then once done the conversational flow will seamlessly return to where it left off.

- In the **getWeather** dialog, select the **BeginDialog** trigger.
- In the **Authoring canvas**, select the **Prompt for text** action.



- In the **Prompt for text** properties panel, select the **Other** tab.
- Expand **Prompt Configurations** and set the **Allow interruptions** field to `true`.



This tells to the Bot Framework to consult the parent dialog's recognizer, which allows the bot to respond to **help** at the user's prompt as well.

Your bot is now ready to test.

## Test the bot

- [Composer v2.x](#)
- [Composer v1.x](#)

1. In the upper right of Composer user interface, select **Start bot** or **Restart bot**.
2. In the **Local bot runtime manager**, select **Open Web Chat**.
3. In the input box, enter any message containing the word *weather*. The bot will ask for a postal code.
4. Enter *help*, *support*, or *advice*. The bot will provide a help response.
5. Then enter *cancel*, *stop*, or *quit*. Notice that the bot doesn't resume the weather dialog. Instead, it sends the a cancel response, and then waits for your next message.

## Next steps

[Tutorial: Add language generation to your bot](#)

# Tutorial: Add language generation to your bot

5/28/2021 • 5 minutes to read

**APPLIES TO:** Composer v1.x and v2.x

One of the primary challenges for a developer is to enable a bot to understand what a user means conversationally and contextually and to respond with useful information. To help performing these complex tasks, Composer includes the Bot Framework language generation library, which gives a greater control on how the bot processes user's input and provides a meaningful response. This article shows you how to integrate the language generation in your bot.

In this tutorial, you learn how to:

- Integrate language generation into your bot using Composer

## Prerequisites

- Completion of the tutorial [Add Help and Cancel to your bot using interruptions](#)
- Knowledge about [language generation](#)

## Language generation

Let's start by adding some variation to the welcome message.

- [Composer v2.x](#)
- [Composer v1.x](#)

1. Start Composer.
2. Select the **weather\_bot** bot project from the **Recent** bot list on the homepage.
3. Select the **Greeting** trigger on the left. Then scroll down to the **Send a response** action in the **True** branch.
4. In the properties pane on the right, click on the **Add alternative** button. This will open a new text field.
5. Enter the text below and press Enter.

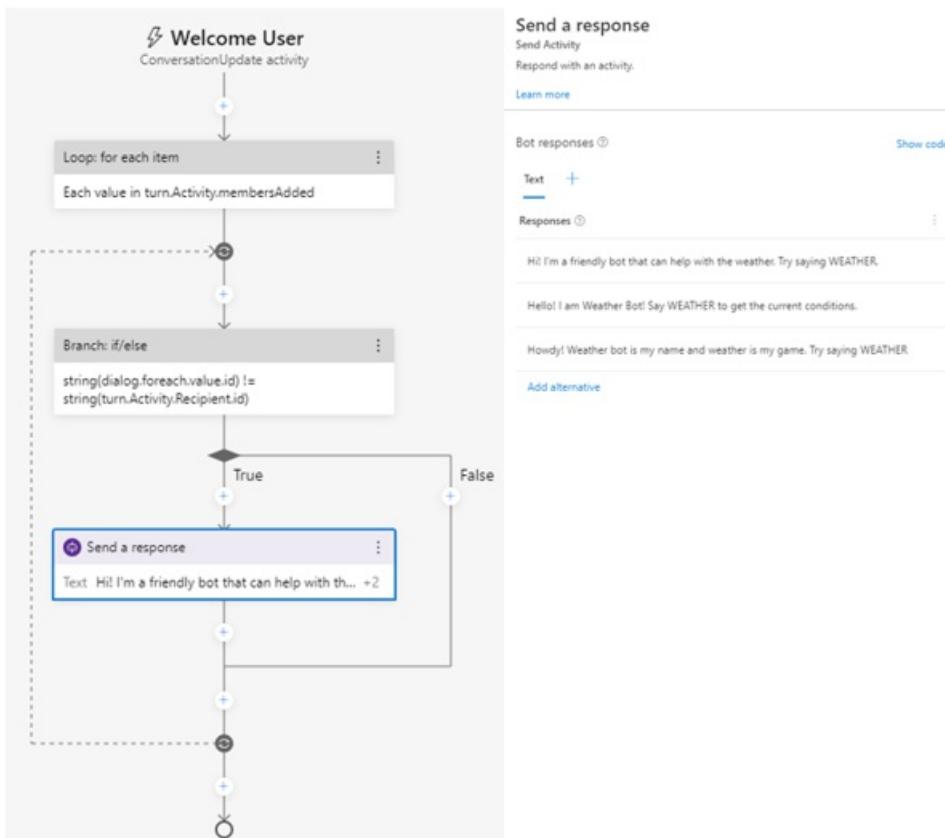
```
Hi! I'm a friendly bot that can help with the weather. Try saying WEATHER.
```

6. In the new text field, enter the text below and press Enter:

```
Hello! I am Weather Bot! Say WEATHER to get the current conditions.
```

7. Repeat this step with the text below:

```
Howdy! Weather bot is my name and weather is my game. Try saying WEATHER.
```



The bot selects any of the above sentences when responding to the user. For more information about response templates, see the section [Template](#), [Anatomy of a template](#), and [Structured response template](#) in the [language generation](#) article.

- To test your new response settings, select the **Start bot** button on the top right. Then select **Open Web Chat**. In Web Chat, select **Restart conversation** a few times to see the greetings being selected.

## Add language generation

Currently, the bot reports the weather in a very robotic manner. You can improve the message used when delivering the weather conditions by utilizing language generation (LG) features such as: *conditional messages* and *parameterized messages*.

- [Composer v2.x](#)
- [Composer v1.x](#)

- In the outermost menu on the left, select **Bot responses**.

You will notice that every message you created in the flow editor also appears here, and that these LG templates are grouped by dialogs. They're linked, and any changes you make in this view will be reflected in the flow as well.

- In the navigation pane, select the **getWeather** dialog.
- In the upper-right of the **Bot Responses** view, select **Show code**. This will enable a syntax-highlighter response editor. You can now edit LG templates for the selected dialog **getWeather**.
- Scroll to the bottom of the editor and paste the following text:

```

# DescribeWeather(weather)
- It is "${dialog.weather}" in ${user.postalcode} and the temperature is ${dialog.fahrenheit}&deg;F or ${dialog.celsius}&deg;C. Have a nice day.

```

This creates a new language generation template named `DescribeWeather`.

The template lets the LG system use the data returned from the weather service to generate a friendlier response.

5. In the outermost menu on the left, select **Create**.
6. In the **Navigation** pane, select the **getWeather** dialog, and then select its **BeginDialog** trigger.
7. In the **Authoring canvas**, select the **Send a response** action that starts with *The weather is....*
8. Select **Add alternative**. In the field that appears, add the following and press Enter:

```
 ${DescribeWeather(dialog.weather)}
```

This syntax lets you nest the `DescribeWeather` template *inside* another template. LG templates can be combined in this way to create more complex ones.

9. Delete the response alternative that starts with *The weather is....* Hover over the response variation, and select the **Remove variation** trash can button.

You're now ready to test your bot in the Web Chat.

## Test the bot

- [Composer v2.x](#)
- [Composer v1.x](#)

1. Click the **Restart Bot** button on the top right, and then click **Open Web Chat**.

Now, when you ask for the weather the bot will send you a message that sounds much more natural than it did previously.

Howdy! Weather bot is my name and weather is my game. Try saying WEATHER.

weather

Just now

Let's check the weather.

What's your postal code?

98052

Just now

It is "broken clouds" in 98052 and the temperature is 48°F or 16°C. Have a nice day.



Type your message



## Next steps

[Tutorial: Incorporating cards and buttons into your bot](#)

# Tutorial: Add cards and buttons to your bot

5/28/2021 • 4 minutes to read

**APPLIES TO:** Composer v1.x and v2.x

This tutorial expands what you learned so far by adding richer message content to your bot using cards and buttons.

In this tutorial, you learn how to:

- Add cards and buttons to your bot using Composer

## Prerequisites

- Completion of the tutorial [Adding language generation to your bot](#)
- Knowledge about [Language Generation](#)
- Knowledge about [Cards](#)
- Knowledge about [Sending responses with cards](#) in Composer

## Adding buttons

Buttons are added as *suggested actions*. You can add preset buttons to your bot that the user can select to provide input. Suggested actions improve the user experience by letting users answer questions or make selections with a simple push of a button, instead of having to type their responses.

First, update the prompt for the user's postal code to include suggested actions for *help* and *cancel* actions.

- [Composer v2.x](#)
- [Composer v1.x](#)

1. Start Composer.
2. Select the **weather\_bot** bot project from the **Recent** bot list on the homepage.
3. In the **Navigation** pane, select the **BeginDialog** trigger under the **getWeather** dialog.
4. Select the **Prompt for text** action, the second action in the flow.
5. Click on the + button next to the **Text** tab.
6. In the drop-down menu, select **Suggested Actions**.
7. A table will appear. Select **Add suggested action**.
8. In the field that appears, enter "help" and press Enter.
9. Select **Add suggested action** again, and in the field that appears, enter "cancel" and press Enter.

## NOTE

Behind the scenes, you are editing the structured response template to include suggestion actions. Alternatively, you could do this manually as follows:

1. In the **Properties pane**, select **Show code**.
2. Update the **Prompt** to include suggested actions in the form of **help** and **cancel** buttons, as shown below.

The screenshot shows the Microsoft Bot Framework Composer interface. On the left, the **Properties pane** displays the following JSON code:

```
[Activity
  Text = What is your postal code?
  SuggestedActions = help | cancel
]
```

Below the Properties pane, a **Prompt for text** node is selected, showing its configuration:

- Text**: What is your postal code?
- Sug. Actions**: help +1

On the right, the **Ask a question - text** response editor shows the updated code:

```
Ask a question - text ⓘ Show response editor
[Activity
  Text = What is your postal code?
  SuggestedActions = help | cancel
]
```

10. Let's test the change. Select the **Start bot** in the top right and then select **Open Web Chat**.

Now when you enter a message containing the word *weather* in Web Chat, the bot asks you for the postal code, and also displays help and cancel buttons as suggested actions.

## Adding cards

Now you can change the weather report to include a card.

- [Composer v2.x](#)
- [Composer v1.x](#)

1. Scroll to the bottom of the **Authoring canvas** and select the **Send a response** node in the *True* branch that starts with  `${DescribeWeather(dialog.weather)}`.
2. In the **Properties** pane on the right, select the Add button (+). In the drop-down menu, select **Attachments**.
3. A new tab called **Attachments** will appear with an empty table below it. Select **Add new attachment**. In the drop-down that appears, select **Create from template** then **Thumbnail card**.
4. A code editor will appear with an empty thumbnail card template. Replace the existing template with the following `ThumbnailCard` template:

```
[ThumbnailCard
  title = Weather in ${dialog.city} in ${dialog.country}
  text = ${DescribeWeather(dialog.weather)}
  image = http://openweathermap.org/img/wn/${dialog.icon}@2x.png
]
```

The screenshot shows the Microsoft Bot Framework Composer interface. On the left, there's a card template with the title "Send a response". The code inside the template is:

```
[ThumbnailCard
  title = Weather
  text = The temperature at ${user.postalcode} is ${{
  ]}
```

On the right, there's a preview window showing the card template with the title "Weather" and the text "The temperature at \${user.postalcode} is \${dialog.fahrenheit}&deg;F.". Below the preview, it says "Template name: #SendActivity\_TZRFsp()

## Test the bot

- [Composer v2.x](#)
- [Composer v1.x](#)

1. Select **Restart Bot** in the top right. Once your bot has restarted click **Open Web Chat**.

In Web Chat, go through the bot flow, say *weather*, and enter a postal code. Notice that the bot now responds with a card that contains the results along with the card title and text.

Howdy! Weather bot is my name and weather is my game. Try saying WEATHER.

**weather**

Just now

Let's check the weather.

What's your postal code?

**98052**

Just now

**Weather in Redmond in US**



It is “broken clouds” in 98052 and the temperature is 48°F or 16°C. Have a nice day.



Type your message



## Next steps

[Tutorial: Add LUIS for language understanding](#)

# Tutorial: Add LUIS for language understanding

5/28/2021 • 9 minutes to read

**APPLIES TO:** Composer v1.x and v2.x

The **Luis recognizer** is one of the recognizers available in Composer. It uses the Language Understanding (LUIS) service, which allows the bot understand the user's response and determine what to do next in a conversation flow.

Language Understanding (LUIS) is a service within Azure Cognitive Services that applies natural language processing to conversational text to predict meaning and extract relevant information.

This tutorial shows how to train the LUIS recognizer to capture the user's **intent** contained in a message. Then how to pass the information to the triggers, which defines how the bot will respond.

In this tutorial, you learn how to:

- Add the LUIS recognizer to your bot.
- Determine user intent and entities to generate helpful responses.

## Prerequisites

- Completion of the tutorial [Incorporating cards and buttons into your bot](#)
- Knowledge about [Language Understanding](#) concept article
- A [LUIS](#) account. If you don't have a LUIS account, you can sign up for one on the [LUIS](#) site.

## Update the recognizer type

- [Composer v2.x](#)
- [Composer v1.x](#)

1. Start Composer.
2. Select the **weather\_bot** bot project from the **Recent** bot list on the homepage.
3. In the **Navigation** pane, select the main dialog **weather\_bot**.
4. In the **Properties** pane, select **Default recognizer** from the **Recognizer Type** drop-down list. The **Default recognizer** uses LUIS.

[Language Understanding](#) ⓘ

**Recognizer Type**

Default recognizer

## Add language understanding data and conditions

In this section, you will learn how to create three **Intent recognized** triggers using LUIS recognizer. You can ignore or delete the **Intent recognized** triggers you created using regular expressions in the [add help and cancel command](#) tutorial.

- [Composer v2.x](#)

- [Composer v1.x](#)

1. Select the **cancel** trigger you created using regular expressions. In the **Properties** pane on the right, and enter the following **Trigger phrases**:

```
- cancel  
- please cancel  
- stop that
```

2. In the **Condition** property from the drop-down list, select **Write an expression**. An = sign will appear. After the equal sign, enter `#Cancel.Score >= 0.8`.

Trigger phrases [?](#)

+ Add entity	▼	Insert entity	▼
<ul style="list-style-type: none"><li>- cancel</li><li>- please cancel</li><li>- stop that</li></ul>			

Intent name: [#cancel](#)

Condition [?](#)

y/n	=#Cancel.Score >= 0.8
-----	-----------------------

This tells the bot not to fire the cancel trigger if the confidence score returned by LUIS is lower than 80%. LUIS is a machine learning based intent classifier and can return a variety of possible matches, so you will want to avoid low confidence results.

3. Now select **help** trigger you created using regular expressions. In the **Properties** pane on the right, and enter the following **Trigger phrases**:

```
- help  
- I need help  
- please help me  
- can you help
```

4. In the **Condition** property from the drop-down list, select **Write an expression**. An = sign will appear. After the equal sign, enter `#Help.Score >= 0.5`.

5. Select the **weather** trigger. In the **Properties** pane, enter the following **Trigger phrases**:

```
- get weather  
- weather  
- how is the weather
```

## Configure a Language Understanding resource

- [Composer v2.x](#)
- [Composer v1.x](#)

You will need to create a Language Understanding (LUIS) resource in Azure. This resource will be used for authoring your language models. The following steps will guide you through creating a Language Understanding resource in Azure or sending a request to your Azure administrator to create a Language Understanding resource on your behalf.

1. You will notice an error icon next to your **weather\_bot** bot in the navigation. Select the icon, and then select **Fix in bot settings**.

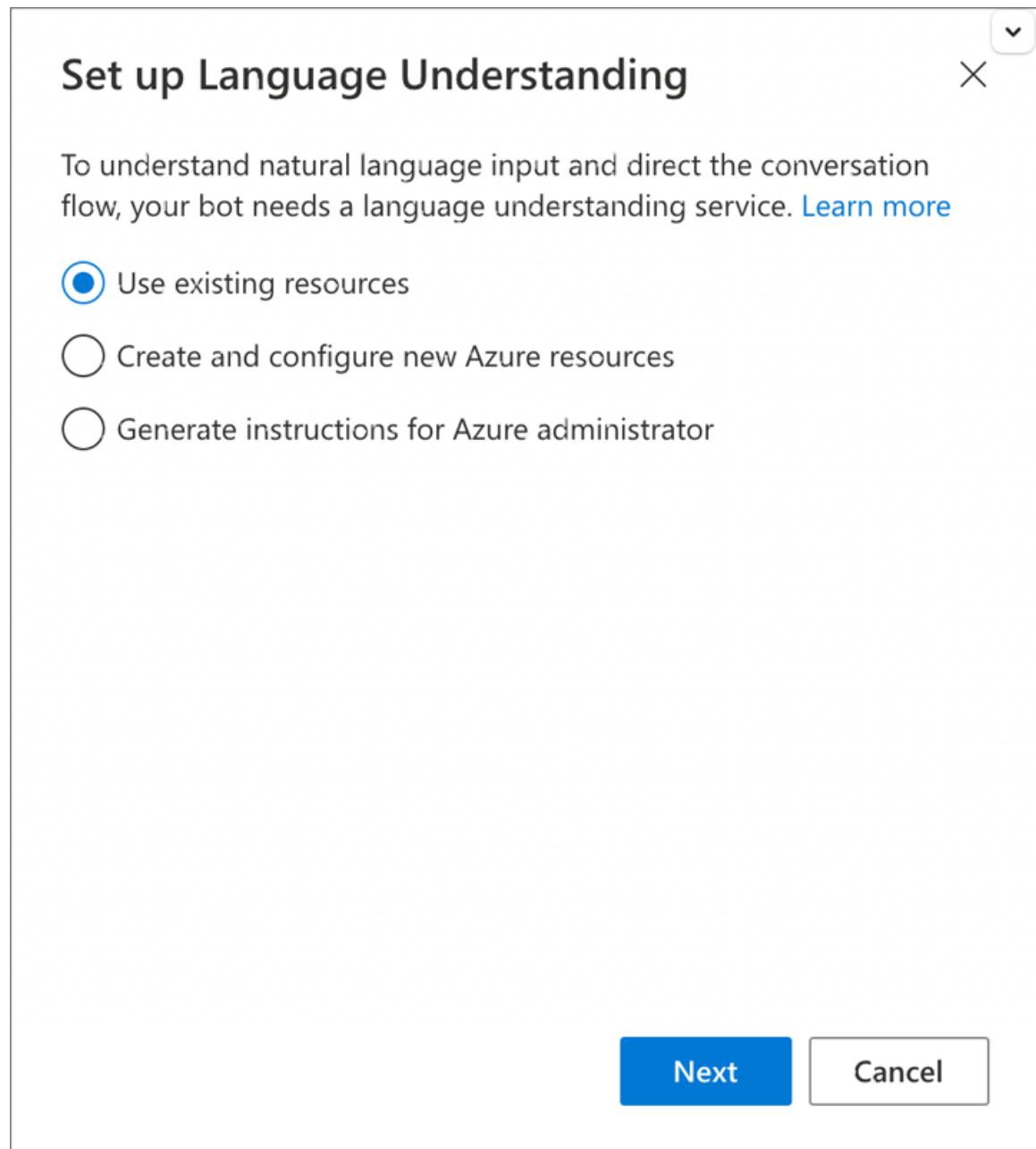
This will navigate you to the **Development resources** tab of the Configure section. Alternatively, you can select the wrench icon in the main menu and then select the **Development resources** tab.

2. You will see the field labeled **Language Understanding authoring key** field is highlighted with an error message.

**NOTE**

The following steps will guide you through creating a new LUIS resource. If already you have a LUIS authoring resource and know the key, you can enter the key directly in this field and skip to the next section.

1. Select the button **Set up Language Understanding**. This window will appear:



2. If you have an Azure account and have permissions to create new resources, select **Create and configure new Azure resources** and select **Next**.

**NOTE**

If you need an Azure administrator to create Azure resources for you, select **Generate instructions for Azure administrator** to create a request you can send to your Azure administrator. You won't be able to continue without a LUIS key.

3. Follow the prompts for signing in to Azure.
4. Once you have signed in to Azure, you will land on the "Create Language Understanding resources" window:

The screenshot shows a modal window titled "Create Language Understanding resources" with a close button (X) in the top right corner. The main instruction text reads: "Select the resource group and region in which your Language Understanding service will be created." Below this, there are three input fields: "Azure resource group" (dropdown menu), "Region \*" (dropdown menu), and "Language Understanding resource name \*" (text input field). At the bottom, there are three buttons: "Back" (disabled), "Next" (disabled), and "Cancel".

5. Select your subscription and then **Next**. The window will appear allowing you to name your resource group, choose a region, and name your resource:

# Create Language Understanding resources

X

Select your Azure directory, then choose the subscription where you'd like your new Language Understanding resource.[Learn more](#)

Azure directory \*

Azure subscription \*

Select subscription

Back

Next

Cancel

6. Enter a name for your Azure resource group, select a region, and then name your Language Understanding resource. Select **Next**.

7. A window will appear, confirming the resource you created:

## Create Language Understanding resources

X

The following Language Understanding resource was successfully created and added to your bot project:

Subscription

Resource Group

Region

Resource name

Done

### 8. Select Done.

You will notice that the key for your Language Understanding resource has been copied into the field. You're now ready to test your bot!

### 1. Select Start bot on the top right, and then select Open Web Chat.

With LUIS, you no longer have to type in exact regex patterns to trigger specific scenarios for your bot. Try phrases like:

- "How is the weather"
- "Weather please"
- "Cancel for me"
- "Can you help me?"

## Using LUIS for entity extraction

You can use LUIS to recognize [entities](#). An entity is a word or phrase extracted from the user's utterance that helps clarify their intent.

For example, the user could say "How is the weather in 98052?" Instead of prompting the user again for a zip

code, your bot could respond with the weather. This is a very simple example of very powerful capabilities. For more information on how to use LUIS for entity extraction in Composer, read the [How to define intent with entities](#) article.

The first step is to add a regex entity extraction rule to the LUIS app.

- [Composer v2.x](#)
- [Composer v1.x](#)

1. Select **User Input** and then select the **weather\_bot** dialog in the navigation pane. (Be sure to select the **weather\_bot** dialog and not the **weather\_bot** root bot.) Select **Show code** and in the **Language Understanding** editor add the following entity definition at the end of the LU content:

```
> Define a postal code entity. Any time LUIS sees a five digit number, it will flag it as 'postal
  code' entity.
$ postalcode:/[0-9]{5}/
```

```
# cancel
- cancel
- please cancel
- stop that

# help
- help
- I need help
- please help me
- can you help

# weather
- get weather
- weather
- how is the weather

> Define a postal code entity. Any time LUIS sees a five digit number, it will flag it as 'postal code' entity.
$ postalcode:/[0-9]{5}/
```

The next step is to create an action in the **BeginDialog** trigger to set the `userpostalcode` property to the value of the `postalcode` entity.

2. Select the **getWeather** dialog in the **Navigation** pane, then the **BeginDialog** trigger.
3. Select **+** in the **Authoring Canvas** to insert an action after the **Send a response** action (that has the prompt *Let's check the weather*). Then select **Set a property** from the **Manage Properties** menu.
4. In the **Properties** field on the right enter `userpostalcode` into the **Property** field and `=@postalcode` in the **Value** field. The `userpostalcode` property will now be set to the value of the `postalcode` entity. If the user enters a postal code as part of the message, they will no longer be prompted for it.

### Set a property

Set Property

Set property to a value.

[Learn more](#)

Property \* ⑦

abc	userpostalcode
-----	----------------

Value \* ⑦

abc	postalcode
-----	------------

Your bot is now ready to test.

## Test the bot

- [Composer v2.x](#)
- [Composer v1.x](#)

1. Select the **Restart Bot** button on the top right and wait for the LUIS application to finish updating your changes. Then select **Open Web Chat**.

Now when you say "how is the weather in 98052", the bot will respond with the weather for that location instead of prompting you for a postal code.

Congratulations, you have completed tutorial and created a weather bot!

# Templates Overview

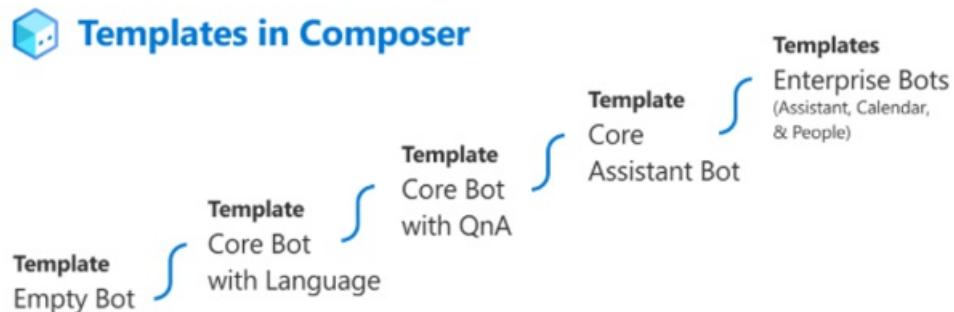
5/20/2021 • 6 minutes to read

**APPLIES TO:** Composer v2.x

In order to simplify the process of getting starting with development of your own conversational experiences, we now provide a set of templates upon which you can build your own conversational experience.

The goal of these templates are to provide common conversational capabilities across a range of popular scenarios which simplify the act of getting started and enable you to focus instead on your own unique conversational capabilities through building on the foundation of a template. An empty bot provides only a few capabilities and each template builds on this to offer increasing capabilities.

Our work with the Virtual Assistant solution accelerator has been incorporated directly into Bot Framework Composer and these templates. Key Virtual Assistant capabilities have been infused directly into the Bot Framework SDK and across all templates and packages enabling easier composition to better suit customer needs.



The choice of template purely represents the starting point for your experience; once created, you can then customize and extend as needed. If you choose an Empty bot as your starting point, then capabilities of other templates can be added as your requirements evolve in the future. Conversational capabilities added to your project by the template can and should be customized further to suit your needs; for example, tailoring the LG responses will be highly recommended in support of your experience. Any provided capabilities can be also be removed if not needed.

We have selected the first set of templates based on our learnings with customers and partners to date and will seek to further develop these templates and consider additional templates based on [feedback](#).

## Templates

The templates and the capabilities they provide are covered in the table below. Assistant type experiences are more sophisticated conversational experiences that have many capabilities available to end users and offer a mix of proactive and reactive experiences to end users versus FAQ or single task oriented bots.

TEMPLATE	DESCRIPTION	CONVERSATIONAL CAPABILITIES	REQUIRED INFRASTRUCTURE
----------	-------------	-----------------------------	-------------------------

TEMPLATE	DESCRIPTION	CONVERSATIONAL CAPABILITIES	REQUIRED INFRASTRUCTURE
Empty Bot	A foundational Bot template which provides a basic greeting to new users and unknown intent handling to catch questions the Bot cannot answer.	<ul style="list-style-type: none"> <li>• Greeting new and returning users</li> <li>• Unknown question handling</li> </ul>	None
Core Bot with Language	Building on the Empty Bot capabilities, this template introduces NLU capabilities and provides Help and Cancel utility capabilities.	<ul style="list-style-type: none"> <li>• Greeting new and returning users</li> <li>• Unknown question handling</li> <li>• Asking for Help</li> <li>• Cancelling a dialog</li> </ul>	<ul style="list-style-type: none"> <li>• Language Cognitive Service</li> </ul>
Core Bot with QnA Maker	Building on the Empty Bot capabilities, this template introduces FAQ/Knowledgebase capabilities through QnAMaker. NLU capabilities can easily be enabled as needed.	<ul style="list-style-type: none"> <li>• Greeting new and returning users</li> <li>• Unknown question handling</li> </ul>	<ul style="list-style-type: none"> <li>• QnAMaker Cognitive Service</li> </ul>
Core Assistant Bot	A foundational Assistant template (comparable to the previous Virtual Assistant template) which provides a wide range of core assistant capabilities including chit-chat, disambiguation and user feedback to support a more sophisticated conversational experience.	<ul style="list-style-type: none"> <li>• Greeting new and returning users</li> <li>• Bot Tour to introduce features to end users</li> <li>• Unknown question handling</li> <li>• Asking for Help</li> <li>• Cancelling a dialog</li> <li>• Disambiguation of NLU (intent) results</li> <li>• Submitting feedback about the bot</li> <li>• Error handling in conversations</li> <li>• Chit chat with QnA Maker (professional personality)</li> </ul>	<ul style="list-style-type: none"> <li>• Language Cognitive Service</li> <li>• QnAMaker Cognitive Service</li> </ul>
Enterprise Assistant Bot	Builds on the Core Assistant Bot to provide a set of employee productivity capabilities covering Calendar and People scenarios often used for Enterprise Assistant scenarios.	<ul style="list-style-type: none"> <li>• Everything in Assistant Core</li> <li>• Orchestrator for Skill dispatch</li> <li>• Calendar (skill)</li> <li>• People (skill)</li> </ul>	<ul style="list-style-type: none"> <li>• Language Cognitive Service</li> <li>• QnAMaker Cognitive Service</li> <li>• AD Application configured with required Graph scopes</li> </ul>

TEMPLATE	DESCRIPTION	CONVERSATIONAL CAPABILITIES	REQUIRED INFRASTRUCTURE
<a href="#">Enterprise Calendar Bot</a>	Provides a rich set of calendar productivity capabilities enabling users to manage their calendar as part of a broader conversational experience. Capabilities include the ability, to create, change and retrieve events on your calendar, find when a group of people are available along with implementation of free-flowing, natural dialog design and advanced use of language generation. A set of Graph custom actions are also provided for calendar related scenarios.	Support managing Office 365 calendars using Microsoft Graph. Start from an advanced template including dialogs, language understanding, and language generation	<ul style="list-style-type: none"> <li>● Language Cognitive Service</li> <li>● AD Application configured with required Graph scopes</li> </ul>
<a href="#">Enterprise People Bot</a>	Provides a rich set of people search productivity capabilities enabling users to find people. Capabilities include the ability, to create, change and retrieve events on your calendar along with implementation of free-flowing, natural dialog design and advanced use of language generation. A set of Graph custom actions are also provided for calendar related scenarios.	Support searching for Azure Active Directory users. Start from an advanced template including dialogs, language understanding, and language generation	<ul style="list-style-type: none"> <li>● Language Cognitive Service</li> <li>● AD Application configured with required Graph scopes</li> </ul>

## Empty Bot Template

The Empty Bot template is a foundational template with few conversational capabilities provided out of the box. A bot created using this template will greet new users with a simple welcome message and has an unknown intent handler meaning questions not handled by the Bot will be responded to with a message conveying the Bot doesn't understand.

Developers looking to start from the basics and build up capabilities gradually or needing a simpler bot experience should consider this template. [More information](#)

### Demo

The demo below shows the key capabilities of a bot created using this template and includes an additional scenario demonstrating NLU entity extraction.

Welcome to your bot.



Type your message



## Core Bot with Language Template

The Core Bot with Language template is a foundational template for NLU focused conversational experiences. It builds upon the Empty Bot capabilities by adding greeting for new and returning users, Help and Cancel utility intents along with Error handling. The LUIS recognizer is added automatically to provide the NLU capabilities and you'll be prompted to create or select an existing LUIS resource before starting the Bot for the first time.

Developers looking to build bots that make use of NLU including entity extraction and slot-filling should consider this template. [More information](#)

### Demo

The demo below shows the key capabilities of a bot created using this template and includes an additional scenario demonstrating NLU entity extraction.



Hi! How can I help?



Type your message



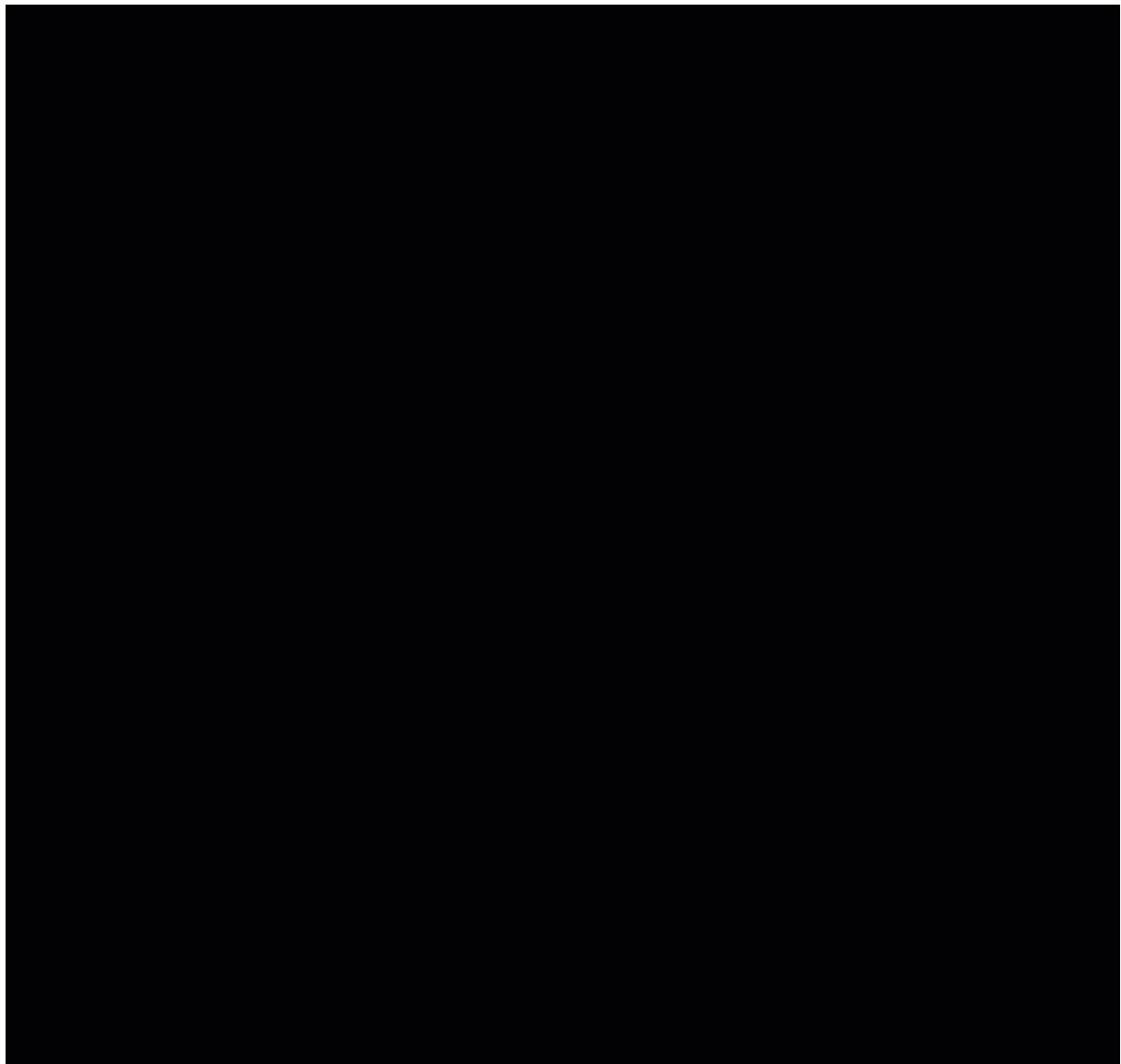
## Core Bot with QnAMaker Template

The Core Bot with QnA template is a foundational template for QnA/FAQ focused conversational experiences. It builds upon the Empty Bot capabilities by adding the QnA recognizer by default and triggering the knowledgebase creation experience. You'll be prompted to create or select an existing QnAMaker resource before starting the Bot for the first time.

Developers looking to build bots that focus on QnA/FAQ types of scenarios should consider this template. [More information](#)

### Demo

The demo below shows the key capabilities of a bot created using this template along with a few example FAQ questions being answered.



## Core Assistant Bot Template

The Core Assistant template provides a foundational assistant-like conversational experience upon which you can build your own conversational experience. An assistant experience is typically more sophisticated with multiple capabilities being offered to end-users spanning both NLU and QNA type experiences. To support these more sophisticated experiences we provide a wider range of conversational capabilities out of the box:

- Greeting new and returning users
- Bot Tour to introduce features to end users
- Unknown question handling
- Asking for Help
- Cancelling a dialog
- Disambiguation of NLU (intent) results
- Submitting feedback about the bot
- Error handling in conversations
- Repeat the previous question
- Chit chat with QnA Maker (professional personality).

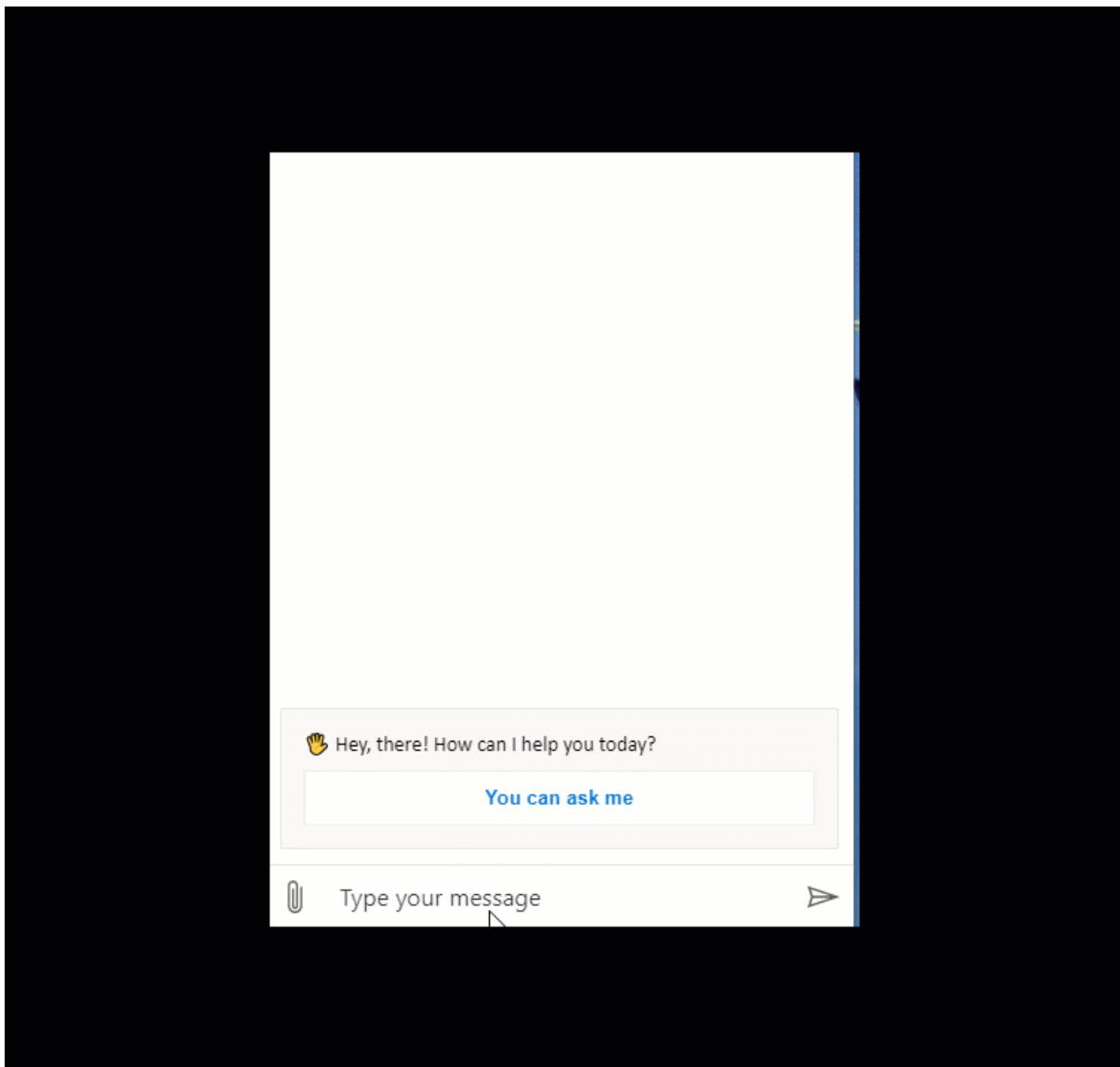
You'll be prompted to create or select an existing LUIS and QnAMaker resource before starting the Bot for the first time.

Developers looking to build more sophisticated conversational experiences with multiple capabilities including

combining NLU and QnA should consider this template. [More information](#)

## Demo

The demo below shows the key capabilities of a bot created using this template along with a few example personality chitchat questions being answered.



## Enterprise Templates

The Enterprise templates provide a starting point for those interested in creating a virtual assistant for common enterprise scenarios. These templates demonstrate a common bot building architecture and high-quality pre-built conversational experiences through a root bot connected to multiple skills.

- [Enterprise Assistant Bot Template](#) documentation for more information.
- [Enterprise Calendar Bot Template](#) documentation for more information.
- [Enterprise People Bot Template](#) documentation for more information.

# Dialogs

6/9/2021 • 5 minutes to read

**APPLIES TO:** Composer v1.x and v2.x

Modern conversational software has many different components, including source code, custom business logic, cloud API, training data for language processing systems, and perhaps most importantly, the actual content used in conversations with the bot's end users. Composer integrates all of these pieces into a single interface for constructing the building blocks of bot functionality called **Dialogs**.

Each dialog represents a portion of the bot's functionality and contains instructions for how the bot will react to the input. Simple bots will have just a few dialogs. Sophisticated bots may have dozens or hundreds of individual dialogs.

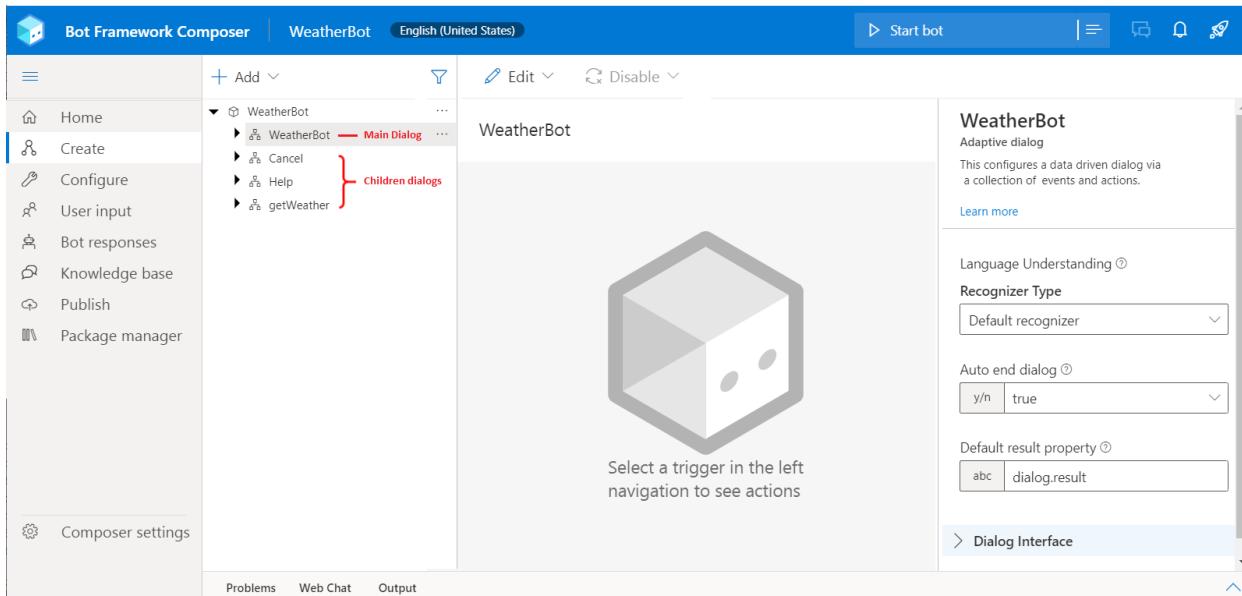
In Composer, dialogs are functional components offered in a visual interface that do not require you to write code. The dialog system supports building an extensible model that integrates all of the building blocks of a bot's functionality. Composer helps you focus on conversation modeling rather than the mechanics of dialog management.

## Types of dialogs

You create a dialog in Composer to manage a conversation objective. There are two types of dialogs in Composer: *main dialog* and *child dialog*. The main dialog is initialized by default when you create a new bot. You can create one or more child dialogs to keep the dialog system organized. Each bot has one main dialog and can have zero or more child dialogs. Refer to the [Create a bot](#) article on how to create a bot and its main dialog in Composer. Refer to the [Add a dialog](#) article on how to create a child dialog and wire it up in the dialog system.

Below is a screenshot of a main dialog named `WeatherBot` and three children dialogs called `Cancel`, `Help` and `getWeather`.

- [Composer v2.x](#)
- [Composer v1.x](#)

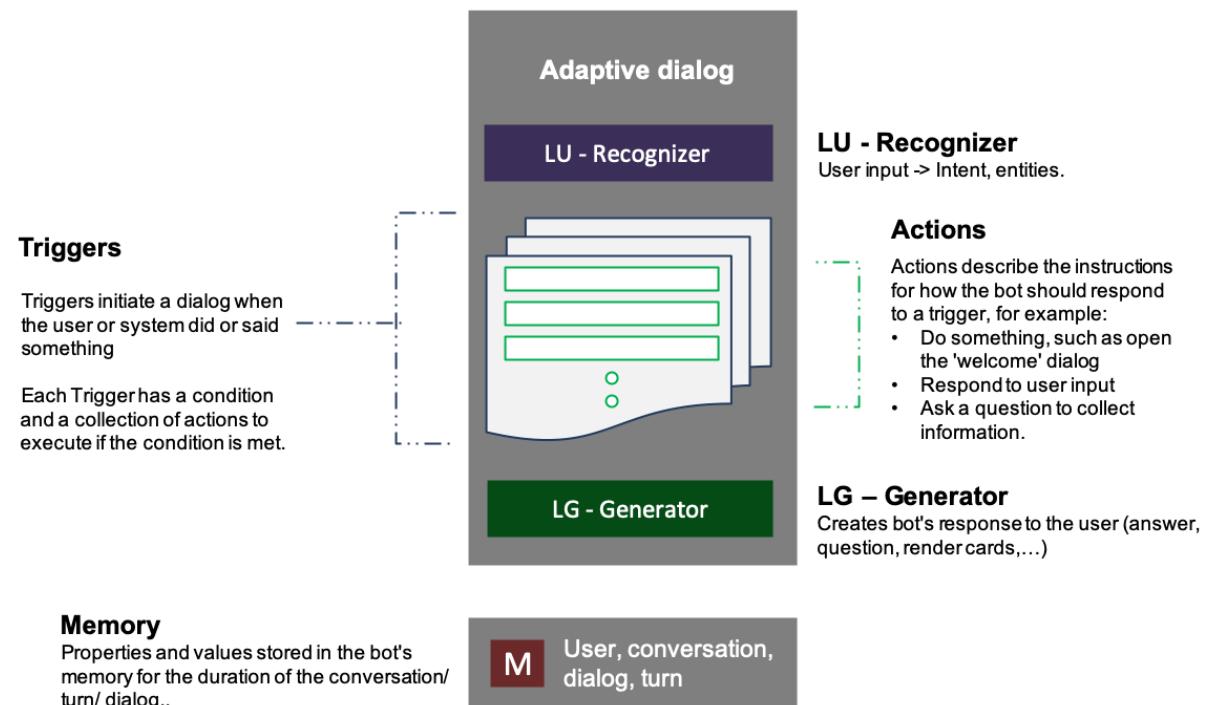


At runtime, the main dialog is called into action and becomes the active dialog, triggering event handlers with the actions you defined during the creation of the bot. As the conversation flows, the main dialog can call a child

dialog, and a child dialog can, in turn, call the main dialog or other children dialogs.

## Anatomy of a dialog

The following diagram shows the anatomy of a dialog in Composer.



### Recognizer

The recognizer interprets what the user wants based on their input. When a dialog is invoked its **recognizer** will

start to process the message and try to extract the primary **intent** and any **entity values** the message includes.

After processing the message, both the **intent** and **entity values** are passed onto the dialog's triggers.

Composer currently supports three recognizers: The LUIS recognizer, which is the default, the Regular expression recognizer and the custom recognizer that allows you to use your own custom recognizer. You can choose only one recognizer per dialog, or you can choose not to have a recognizer at all.

Recognizers give your bot the ability to understand and extract meaningful pieces of information from user input. All recognizers emit events when the recognizer picks up an **intent** (or extracts **entities**) from a given user **utterance**. The **recognizer** of a dialog is not always called into play when a dialog is invoked. It depends on how you design the dialog system.

Below is a screenshot of recognizers in Composer.

#### Recognizer Type

Default recognizer

Default recognizer

Regular expression recognizer

Custom recognizer

- **Default recognizer**: enables you to use the following different recognizers:

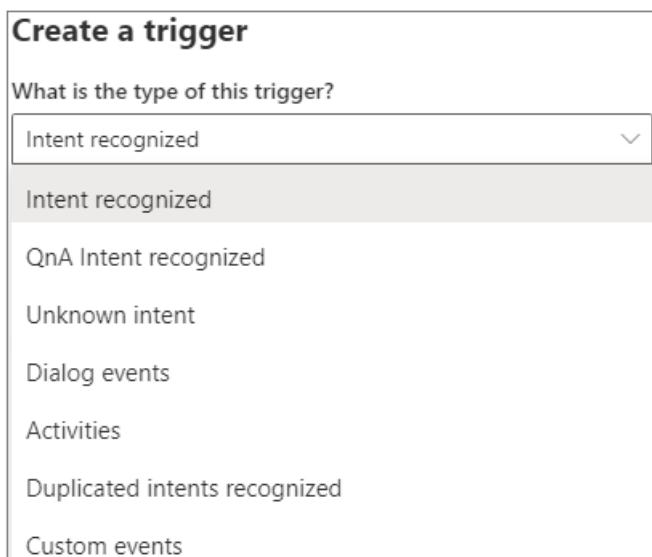
- *None* - do not use recognizer.
- *Luis recognizer* - to extract intents and entities from a user's utterance based on the defined [LUIS](#) application.

- *QnA Maker recognizer* - to extract intents from a user's utterance based on the defined [QnAMaker](#) application.
- *Cross-trained recognizer set* - to compare recognition results from more than one recognizer to decide a winner.
- **Regular expression recognizer:** gives you the ability to extract intent and entity data from an utterance based on regular expression patterns.
- **Custom recognizer:** enables you to customize your own recognizer by editing JSON in the form.

For additional information see the ([Language understanding](#))([#concept-language-understanding.md](#)) concept article.

## Trigger

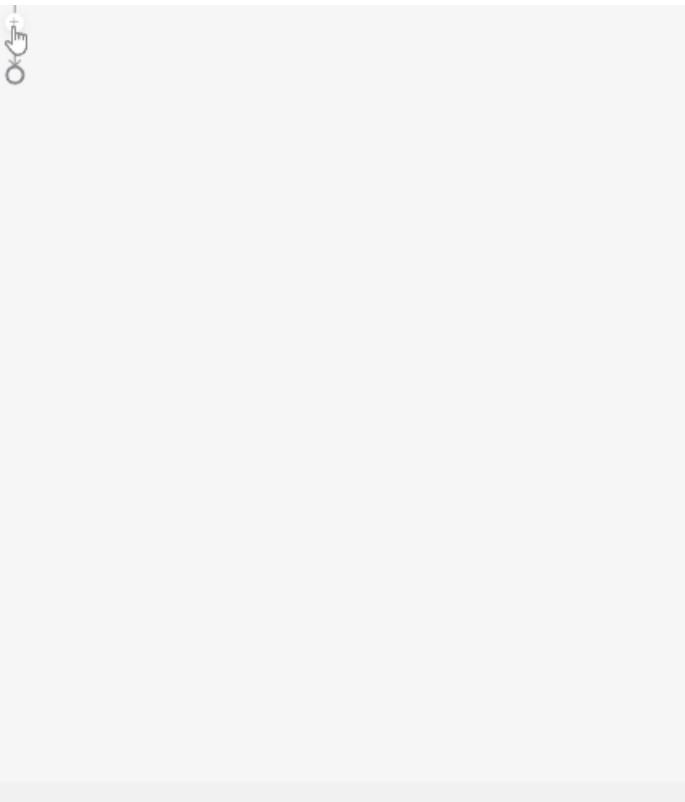
The functionality of a dialog is contained within triggers. Triggers are rules that tell the bot how to process incoming messages and are also used to define a wide variety of bot behaviors, from performing the main fulfillment of the user's request, to handling [interruptions](#) like requests for help, to handling custom, developer-defined events originating from the app itself. Below is a screenshot of the trigger menu in Composer.



For additional information see the ([Triggers](#))([#concept-events-and-triggers.md](#)) concept article.

## Action

Triggers contain a series of actions that the bot will undertake to fulfill a user's request. Actions are things like sending messages, responding to user questions using a [knowledge base](#), making calculations, and performing computational tasks on behalf of the user. The path the bot follows through a dialog can branch and loop. The bot can ask even answer questions, validate input, manipulate and store values in memory, and make decisions. Below is a screenshot of the action menu in Composer. Select the + sign below the trigger you can mouse over the action menu.



## Language Generator

As the bot takes actions and sends messages, the **Language Generator** is used to create those messages from variables and templates. Language generators can create reusable components, variable messages, macros, and dynamic messages that are grammatically correct.

For additional information see the ([Language generation](#))(#concept-language-generation.md) concept article.

## Dialog actions

A bot can have from one to several hundred dialogs, and it can get challenging to manage the dialog system and the conversation with users. In the [Add a dialog](#) section, we covered how to create a child dialog and wire it up to the dialog system using **Begin a new dialog** action. Composer provides more dialog actions to make it easier to manage the dialog system. You can access the different dialog actions by clicking the + node under a trigger and then select **Dialog management**.

Below is a list of the dialog actions available in Composer:

DIALOG ACTION	DESCRIPTION
Begin a new dialog	An action that begins another dialog. When that dialog is completed, it will return to the caller.
End this dialog	A command that ends the current dialog, returning the <code>resultProperty</code> as the result of the dialog.
Cancel all dialogs	A command to cancel all of the current dialogs by emitting an event that must be caught to prevent cancellation from propagating
End this turn	A command to end the current turn without ending the dialog.

DIALOG ACTION	DESCRIPTION
Repeat this Dialog	An action that repeats the current dialog with the same dialog.
Replace this Dialog	An action that replaces the current dialog with the target dialog.

With these dialog actions, you can easily create an extensible dialog system without worrying about the complexities of dialog management.

## Next

- [Events and triggers](#)

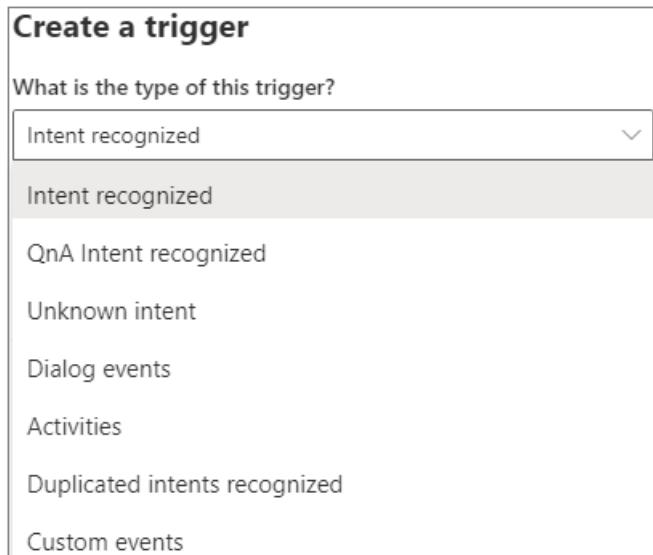
# Triggers

5/26/2021 • 5 minutes to read

**APPLIES TO:** Composer v1.x and v2.x

In Bot Framework Composer, each [dialog](#) includes one or more event handlers called *triggers*. Each trigger contains one or more *actions*. Actions are the instructions that the bot will execute when the dialog receives any event that it has a trigger defined to handle. Once a given event is handled by a trigger, no further action is taken on that event. Some event handlers have a condition specified that must be met before it will handle the event and if that condition is not met, the event is passed to the next event handler. If an event is not handled in a child dialog, it gets passed up to its parent dialog to handle and this continues until it is either handled or reaches the bots main dialog. If no event handler is found, it will be ignored and no action will be taken.

To see the complete trigger menu in Composer, select a dialog, then select the three-dot menu on the right side, choose **+ Add a trigger** from the dropdown menu.



## Anatomy of a trigger

The basic idea behind a trigger (event handler) is "When (*event*) happens, do (*actions*)". The trigger is a conditional test on an incoming event, while the actions are one or more programmatic steps the bot will take to fulfill the user's request.

A trigger contains the following properties:

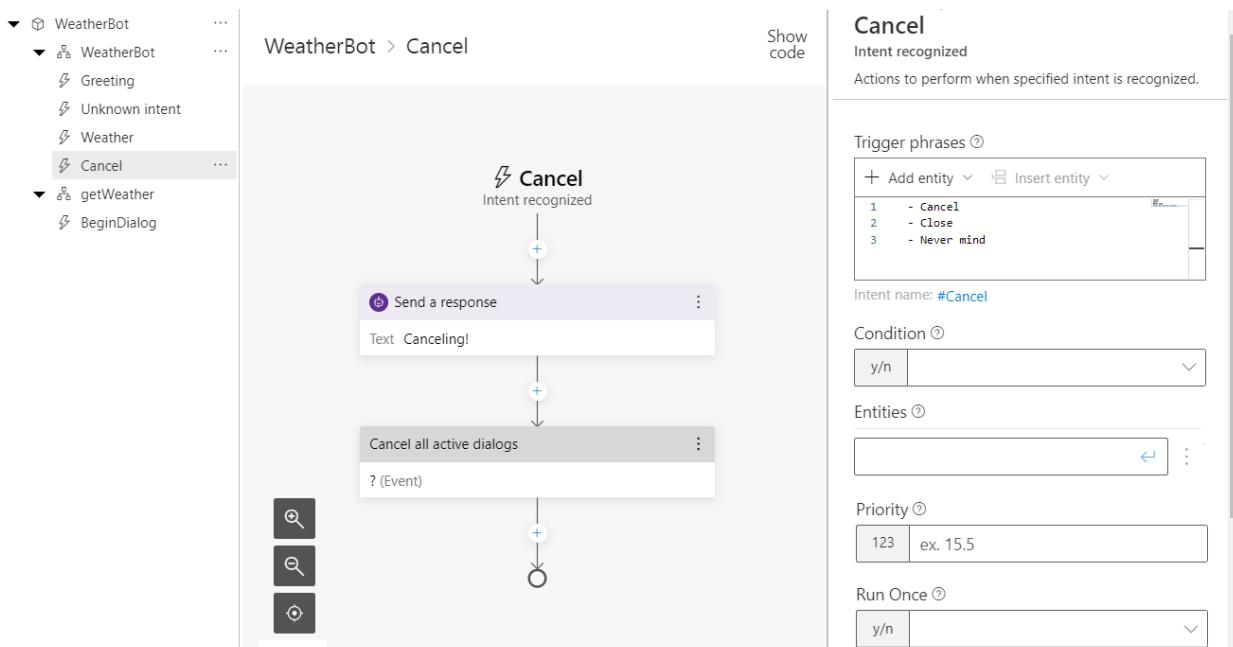
TRIGGER PROPERTY	DESCRIPTION
Name	The trigger name, which can be changed in the properties panel.
Actions	The set of instructions that the bot will execute.

Trigger Property	Description
Condition	The condition to be created or updated in the properties panel and is ignored if left blank, otherwise it must evaluate to <i>true</i> for the event to fire. Conditions must follow the <a href="#">Adaptive expressions</a> syntax. If the condition is ignored or evaluates to false, processing of the event continues with the next trigger.

A dialog can contain multiple triggers. You can view them under the specific dialog in the navigation pane. Each trigger shows as the first node in the authoring canvas. A trigger contains actions defined to be executed. Actions within a trigger occur in the context of the active dialog.

The screenshot below shows the properties of an **Intent recognized** trigger named *Cancel* that is configured to fire whenever the *Cancel/intent* is detected as shown in the properties panel. In this example the **Condition** field is left blank, so no additional conditions are required in order to fire this trigger.

- [Composer v2.x](#)
- [Composer v1.x](#)



If you want to specify a confidence threshold for a specific intent, use the **Condition** field. In the above example, you could set a condition for this specific intent such as: `=#cancel.score >=0.8`.

## Types of triggers

There are different types of triggers that all work in a similar manner, and in some cases can be interchanged. This section will cover the different types of triggers and when you should use them. See the [define triggers](#) article for additional information.

### Intent triggers

Intent triggers handle events emitted by [recognizers](#). After the first round of events is fired, the bot will pass the incoming message through the recognizer. If an intent is detected, it will be passed into the trigger with any **entities** contained in the message. If no intent is detected by the recognizer, an **Unknown intent** trigger will fire, which handles intents not handled by any trigger.

There are four intent triggers in Composer:

- **Intent recognized**

- QnA Intent recognized
- Unknown intent
- Duplicated intents recognized

You should use *intent triggers* when you want to:

- Trigger major features of your bot using natural language.
- Recognize common interruptions like "help" or "cancel" and provide context-specific responses.
- Extract and use entity values as parameters to your dialog.

For additional information see [how to define triggers](#) article.

### **Dialog events triggers**

The base type of triggers are dialog triggers. Almost all events start as dialog events which are related to the "lifecycle" of the dialog. Currently there are four dialog events triggers in Composer:

- Dialog started (Begin dialog event)
- Dialog cancelled (Cancel dialog event)
- Error occurred (Error event)
- Re-prompt for input (Reprompt dialog event)

Most dialogs include a trigger configured to respond to the `BeginDialog` event, which fires when the dialog begins. This allows the bot to respond immediately.

You should use *dialog triggers* to:

- Take actions immediately when the dialog starts, even before the recognizer is called.
- Take actions when a "cancel" signal is detected.
- Take actions on messages received or sent.
- Evaluate the content of the incoming activity.

For additional information, see the [dialog events](#) section of the article on how to define triggers.

### **Activities triggers**

Activities triggers are used to handle activities such as when a new user joins and the bot begins a new conversation. **Greeting (ConversationUpdate activity)** is a trigger of this type and you can use it to send a greeting message. When you create a new bot, the **Greeting (ConversationUpdate activity)** trigger is initialized by default in the main dialog. This specialized option is provided to avoid handling an event with a complex condition attached. **Message events** is a type of Activity trigger to handle message activities.

Here is a list of **Activities triggers** currently provided in Composer:

- [Composer v2.x](#)
- [Composer v1.x](#)
- Activities (Activities received)
- Greeting (ConversationUpdate Activity)
- Conversation ended (EndOfConversation activity)
- Event received (Event activity)
- Handover to human (Handoff activity)
- Conversation invoked (Invoke activity)
- User is typing (Typing activity)
- Message received (Message received activity)
- Command received (Command activity received)

- **Command Result received** (**Command Result activity received**)
- **Message deleted** (**Message deleted activity**)
- **Message reaction** (**Message reaction activity**)
- **Message updated** (**Message updated activity**)

You should use **Activities** triggers when a user begins a new conversation with the bot and when you want to take action on receipt of the following activity types:

- EndOfConversation
- Event
- HandOff
- Invoke
- Typing
- MessageReceived
- Command
- CommandResult
- MessageUpdate
- MessageDelete
- MessageReaction

For additional information, see the [Activities](#) trigger in the [Define triggers](#) article.

### Custom events

You can create and emit your own events by creating an action associated with any trigger, then you can handle that custom event in any dialog in your bot by defining a **Custom event** trigger.

Bots can emit your user-defined events using **Emit a custom event**. If you define an **Emit a custom event** and it fires, any **Custom event** in any dialog will catch it and execute the corresponding actions.

For additional information, see the [Custom events](#) section in the [How to define triggers](#) article.

## Next

- [Conversation flow and memory](#)
- [How to define triggers](#)

# Conversation flow and memory

6/11/2021 • 16 minutes to read

**APPLIES TO:** Composer v1.x and v2.x

All bots built with Bot Framework Composer have a *memory*, a representation of everything that is currently in the bot's active state. Developers can store and retrieve values in the bot's memory, and can use those values to create loops, branches, dynamic messages and behaviors in the bot. Templates and adaptive expressions can also work with properties stored in memory.

The memory system makes it possible for bots built in Composer to do things like:

- Store user profiles and preferences.
- Remember things between sessions, such as the last search query or a list of recently mentioned locations.
- Pass information between dialogs.

## Anatomy of a property in memory

A piece of data in memory is referred to as a *property*. A property is a distinct value identified by a specific address comprised of two parts, the *scope* of the property and the *name* of the property: `scope.name`.

Here are some examples:

- `dialog.index`
- `this.value`
- `turn.activity`
- `user.name`
- `user.profile.age`

The scope of the property determines when the property is available, and how long the value will be retained.

### TIP

It's useful to establish conventions for when to use the `conversation`, `user`, `turn`, and `dialog` scopes. Such conventions help with maintenance and context sharing. Both user and conversation state are scoped by channel. The same person using different channels to access your bot appears as different users, one for each channel, and each with a distinct user state.

When creating a property, also consider how long the property needs to exist. Read more in the [composer best practices](#) article.

- [Composer v2.x](#)
- [Composer v1.x](#)

The Bot Framework defines some longer-term scopes.

- The `settings` scope contains information from the bot configuration file. This is a read-only scope.
- The `user` scope associates properties with the current user. Properties in the user scope do not expire. These properties are in scope while the bot is processing an activity associated with the user.
- The `conversation` scope associates properties with the current conversation. Properties in the conversation scope have a lifetime of the conversation itself. These properties are in scope while the bot is processing an

activity associated with the conversation (for example, multiple users together in a Microsoft Teams channel).

The framework defines some shorter-term scopes. Use these for values only relevant to a specific task.

- The `dialog` scope associates properties with the active dialog. Properties in the dialog scope are retained until the dialog ends.
- The `this` scope associates properties with the current action. This is helpful for input actions since their life time typically lasts beyond a single turn of the conversation. Composer defines some properties for you. For an input action:
  - `this.value` holds the recognized value for the current input action.
  - `this.turnCount` holds the number of times the bot asked the user for this input.
- The `turn` scope associates properties with the current turn. Properties in the turn scope expire at the end of the turn.

The framework also defines scopes for managing the dialog system itself.

- The `dialogcontext` scope contains properties associated with the dialog stack.
- The `dialogclass` scope contains information about the active dialog.
- The `class` scope contains information about the current action.

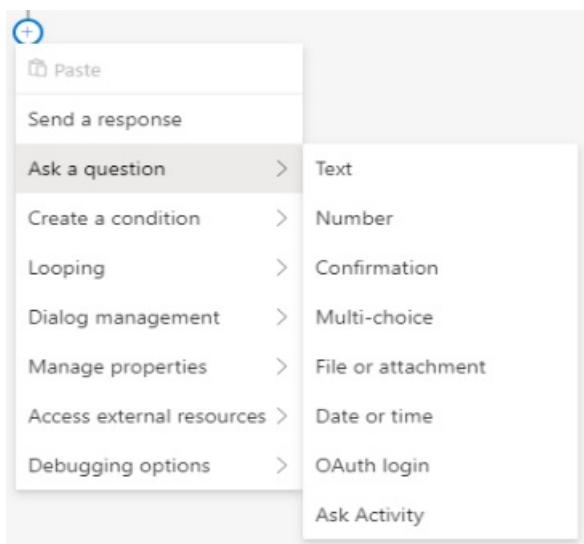
## Set properties with prompts

There are three tabs in the properties panel that relate to setting properties with prompts: asking for input, managing properties, and validating input. You might configure an action that prompts for the user's age to store the result in a `user.age` property.

### Prompt for input

- [Composer v2.x](#)
- [Composer v1.x](#)

Input is collected from users with prompt types provided in the **Ask a question** sub-menu.



Prompts define the questions posed to the user and are set in the **Responses** box under the **Bot response** tab in the properties panel.

## Bot response

## User input

## Other

Ask a question - number 

[Show code](#)

Text 

Responses 

:

What is your age?

[Add alternative](#)

### TIP

When prompting the user for information, you can enter in a plain text string, as the example above demonstrates, but you can also use any of the three insert menus. Each results in a drop-down list containing relevant options to choose from:

 Reference an existing item in your [Bot Responses](#) template.

 Reference a property in memory

 Reference a [prebuilt adaptive expressions function](#).

## Process input

- [Composer v2.x](#)
- [Composer v1.x](#)

Under the **User Input** tab, the **Property** property indicates where the user's response will be stored. You can use the **Output format** property to post-process the user's input.

Bot response

User input

Other

Property 

abc	ex. \$birthday, dialog.\${user.name}
-----	--------------------------------------

Output format 

abc	ex. =toUpperCase(this.value), \${toUpperCase(this.value)}
-----	---

Value 

abc	ex. hello world, Hello \${user.name}, user.lastName)
-----	--

Expected responses

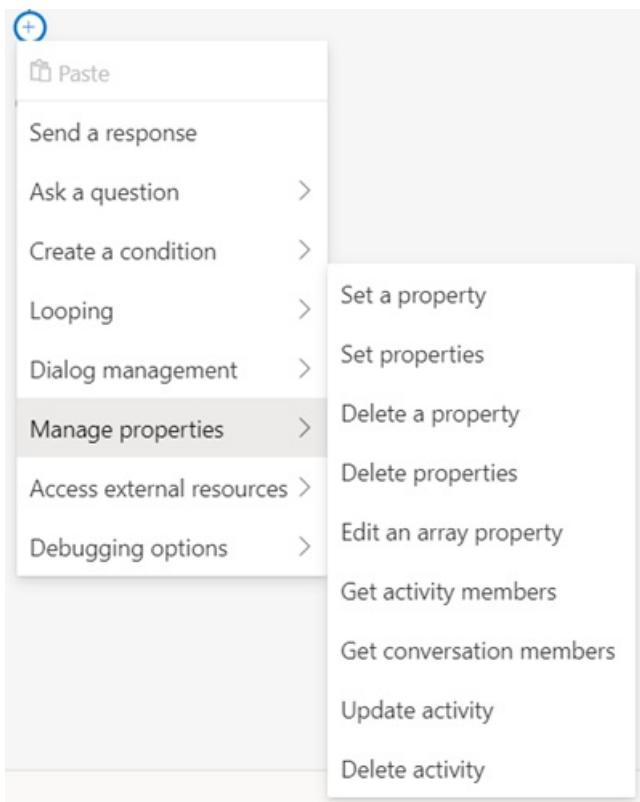
```
+ Add entity ▾ ➔ Insert entity ▾  
1   > add some expected user responses:  
2   > - Please remind me to {itemTitle=buy milk}  
3   > - remind me to {itemTitle}  
4   > - add {itemTitle} to my todo list  
5   >  
6   > entity definitions:  
7   > @ ml itemTitle  
8
```

Intent name: [#TextInput\\_Response\\_MflDMs](#)

For more information about implementing text other prompts see the article [Asking users for input](#).

## Manipulate properties using memory actions

Composer provides a set of memory manipulation actions in the **Manage properties** sub-menu. Use these to create, modify and delete properties in memory. Properties can be created in the editor and during runtime. Composer manages the underlying data for you.



## Set a property

- [Composer v2.x](#)
- [Composer v1.x](#)

Use **Set a property** to set the value of a property. The image below shows the **Value** drop down-list with its available options.

## Set a property

Set Property  
Set property to a value.

[Learn more](#)

Property \* ⑦

Value \* ⑦

abc string

123 number

y/n boolean

[ ] array

{ } object

The value of a property can be set to a literal value, like `true`, `0`, or `fred`, or it can be set to the result of an [adaptive expression](#). It's not necessary to initialize the property when storing simple values.

## Set properties

- [Composer v2.x](#)
- [Composer v1.x](#)

Use **Set properties** to set a group of properties.

## Set properties

Set Properties

Set one or more property values.

[Learn more](#)

Assignments \* 

[Add](#)

The value of each property is assigned individually in the **Properties panel**. You select **Add** to set the next one.

## Delete a property

- [Composer v2.x](#)
- [Composer v1.x](#)

Use **Delete a Property** to remove a property from memory.

### Delete a property

Delete Property

Delete a property and any value it holds.

[Learn more](#)

Property \* 

ex. Hello \${user.name}

## Delete properties

- [Composer v2.x](#)
- [Composer v1.x](#)

Use **Delete properties** to remove properties from memory.

### Delete properties

Delete Properties

Delete multiple properties and any value it holds.

[Learn more](#)

Properties \* 



## Edit an Array Property

- [Composer v2.x](#)
- [Composer v1.x](#)

Use **Edit an Array Property** to add and remove items from an array. Items set in **Value** can be added or removed from the beginning or end of an array in the **Items property** using push, pop, take, remove, and clear in **Type of change**. The result of the edited array is saved to **Result Property**.

## Edit an array property

Edit array

Modify an array in memory

[Learn more](#)

Type of change 

abc		
-----	--	---

Items property \* 

abc	ex. Hello \${user.name}
-----	-------------------------

Result property 

abc	ex. Hello \${user.name}
-----	-------------------------

Value 

abc 	ex. milk
---	----------

Note that it's possible to push the value of an existing property into an array property. For example, push

`turn.choice` onto `dialog.choices`.

## Get activity members

- [Composer v2.x](#)
- [Composer v1.x](#)

Use **Get activity members** when you need to get the members who are participating in an activity.

## Get activity members

`Microsoft.GetActivityMembers`

Get the members who are participating in an activity.  
(BotFrameworkAdapter only)

Property 

abc	ex. user.age
-----	--------------

Activity Id 

abc	ex. turn.lastresult.id
-----	------------------------

## Get conversation members

- [Composer v2.x](#)
- [Composer v1.x](#)

Use **Get conversation members** when you need to get the members who are participating in a conversation.

## Get conversation members

Microsoft.GetConversationMembers

Get the members who are participating in an conversation.  
(BotFrameworkAdapter only)

### Property ②

abc	ex. user.age
-----	--------------

## Update activity

- [Composer v2.x](#)
- [Composer v1.x](#)

Use **Update activity** to update a message or an activity that was previously sent. To do so, you need the ID of the message that was already sent that you want to update. To get the ID of a message, save `turn.lastresult.id` after you call `SendActivities`. You can use a [Set a property](#) action to capture this value and store it for later use.

## Update an activity

Update Activity

Respond with an activity.

### Activity Id ②

abc	ex. =turn.lastresult.id
-----	-------------------------

### Activity ②

[Show code](#)

Text +

### NOTE

The **Update activity** action is supported by channels such as Microsoft Teams and a variety of open-source channel adapters such as Slack and Webex. Before using this action, check whether the channels your bot uses support this activity type. This action is not currently supported in Bot Framework Emulator and Web Chat.

## Delete activity

- [Composer v2.x](#)
- [Composer v1.x](#)

Use **Delete activity** to delete a message or an activity that was previously sent. To do so, you need the ID of the message that was already sent that you want to delete. To get the ID of a message, save `turn.lastresult.id`

after you call `SendActivities`. You can use a [Set a property](#) action to capture this value and store it for later use.

## Delete Activity

`Microsoft.DeleteActivity`

Delete an activity that was previously sent.

ActivityId \* ②

abc

ex. =turn.lastresult.id

### NOTE

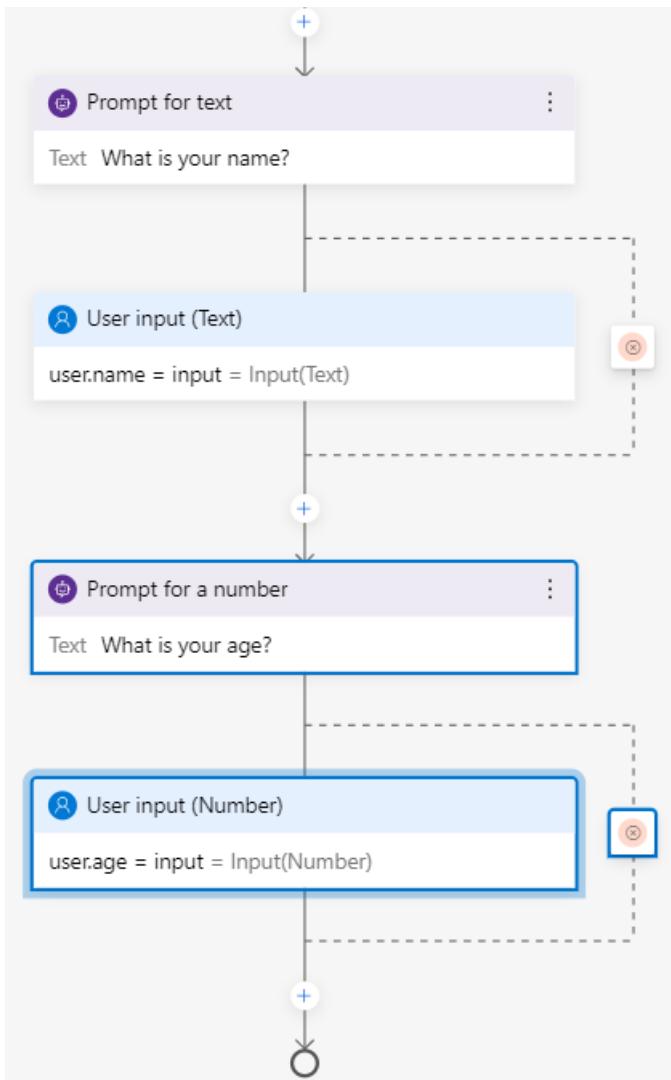
The **Delete activity** action is supported by channels such as Microsoft Teams and a variety of open-source channel adapters such as Slack and Webex. Before using this action, check whether the channels your bot uses support this activity type. This action is not currently supported in Bot Framework Emulator and Web Chat.

## Manipulate properties with dialogs

- [Composer v2.x](#)
- [Composer v1.x](#)

A child dialog can pass properties to its parent dialog. In this way, a child dialog can encapsulate a multi-step interaction, collect and compute multiple values, and then return a single object containing all of the properties to its parent dialog.

For example, a child dialog named **profile** may have two prompts to build a compound property representing a user profile:



When the active dialog returns properties its parent dialog, the return value is specified as the **Default result property**. You can view and modify this in the **Properties** pane when the dialog is selected in the **Navigation** pane:

Default result property ⓘ

abc	dialog.result
-----	---------------

Finally, the parent dialog is configured to capture the return value inside the **Begin a new dialog** action:

## Begin a new dialog

Begin dialog

Begin another dialog.

[Learn more](#)

Dialog name 

profile



Property 

abc

=dialog.profile

Options 

object 

Key \*

Value

Add a new key

Add a new value  



Activity processed 

y/n

true



When executed, the bot will execute the **profile** child dialog, collect the user's name and age in a *temporary* scope, then return it to the parent dialog where it's captured into the `user.profile` property and stored permanently.

## Automatic properties

Some properties are automatically created and managed by the bot. These are available automatically. For example:

PROPERTY	DESCRIPTION
turn.activity	The full incoming <a href="#">Activity</a> object.
turn.intents	If a recognizer is run, the intents found.
turn.entities	If a recognizer is run, the entities found.
turn.dialogEvents.event.name.value	Payload of a custom event fired using the <code>EmitEvent</code> action.

See [Memory scopes and properties](#) for more information.

## Refer to properties in memory

Bots can retrieve values from memory for a variety of purposes. The bot may need to use a value in order to construct an outgoing message, or make a decision based on a value then perform actions based on that decision, or use the value to calculate other values.

Sometimes, you will refer directly to a property by its address in memory: `user.name`. Other times, you will refer to one or more properties as part of an expression: `(dialog.orderTotal + dialog.orderTax) > 50`.

## Expressions

Bot Framework Composer uses the [Adaptive expressions](#) to calculate computed values. This syntax allows developers to create composite values, define complex conditional tests, and transform the content and format of values. For more information see the Adaptive expressions [operators](#) and [pre-built functions](#).

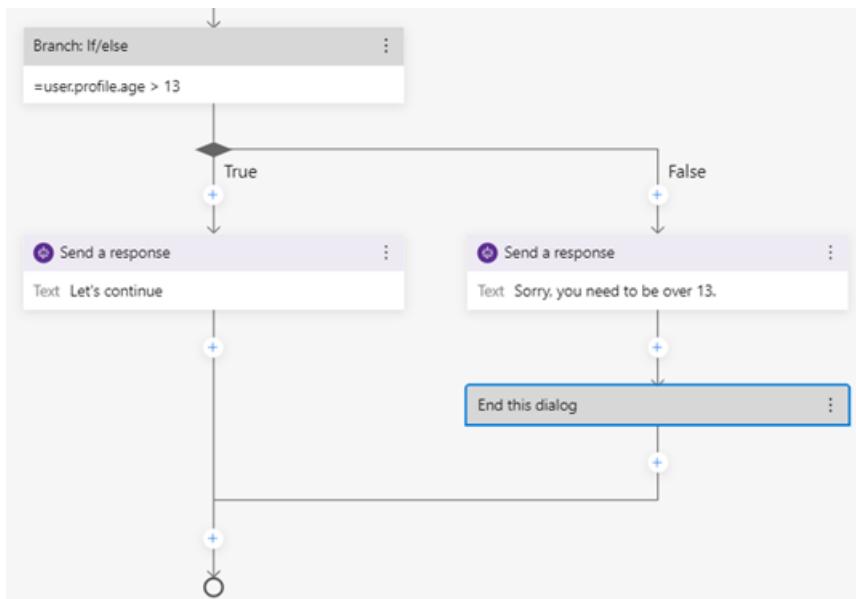
When used in expressions, no special notation is necessary to refer to a property from memory.

## Memory in branching actions

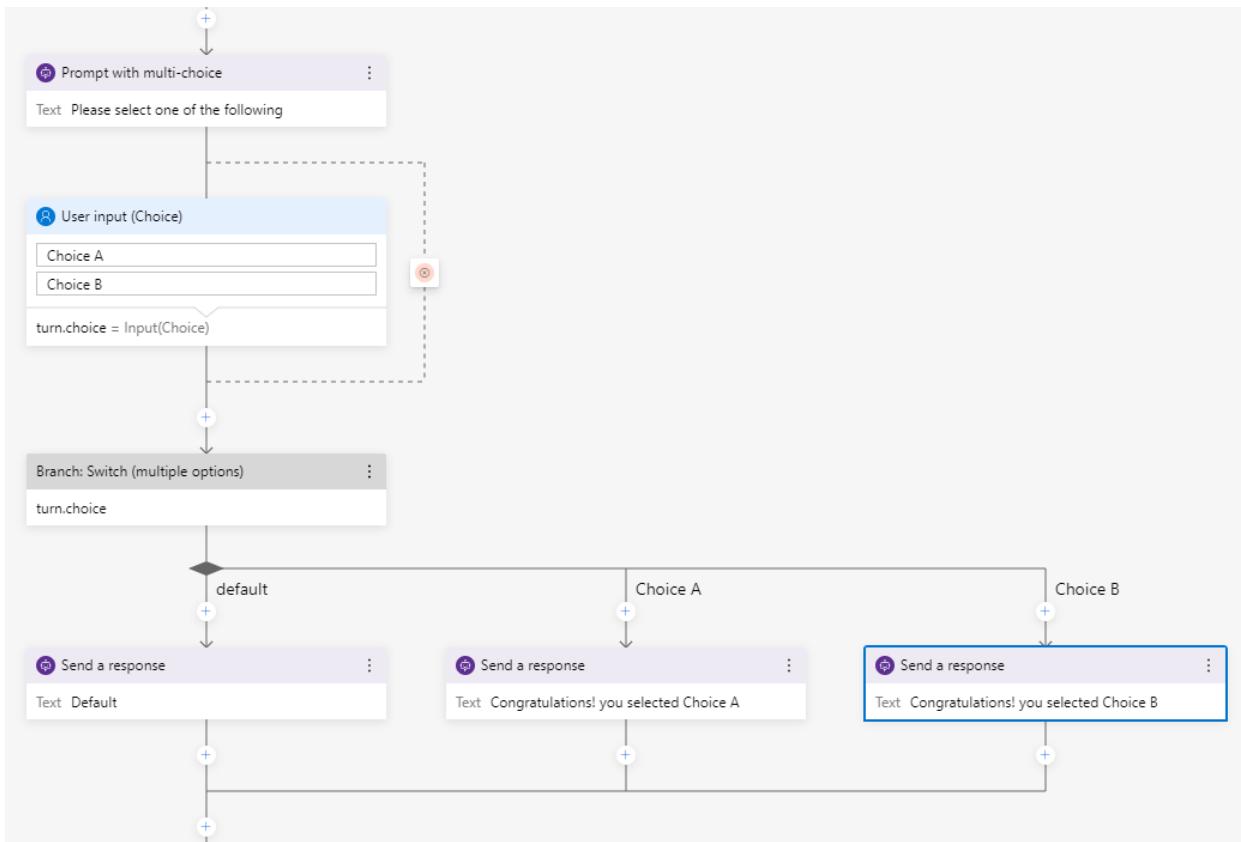
A bot can evaluate values from memory when making decisions inside a [branching action](#) like an **If/Else** or **Switch** branch. The conditional expression that is tested in one of these branching actions is an expression that, when evaluated, drives the decision.

In the example below, the expression `=user.profile.age > 13` will evaluate to either `True` or `False`, and the flow will continue through the appropriate branch.

- [Composer v2.x](#)
- [Composer v1.x](#)



In this example, after prompting the user using the **Prompt with multi-choice** action and storing the results in the `turn.choice` property, the **Branch:Switch** action is used to make a decision based on the users selection.



## Memory in loops

- Composer v2.x
- Composer v1.x

When using **For each** and **For each page** loops, properties also come into play. Both require an **Items** property that holds the array, and **For each page** loops also require a **Page size**, or number of items per page.

### Loop: For each item

For Each

Execute actions on each item in a collection.

[Learn more](#)

Items property \* ②

abc	ex. user.todoList
-----	-------------------

Index property ②

abc	dialog.foreach.index
-----	----------------------

Value property ②

abc	dialog.foreach.value
-----	----------------------

## Memory in bot responses

One of the most powerful features of the Bot Framework system is the bot response language, particularly when used alongside properties pulled from memory.

You can refer to properties in the text of any message, including prompts.

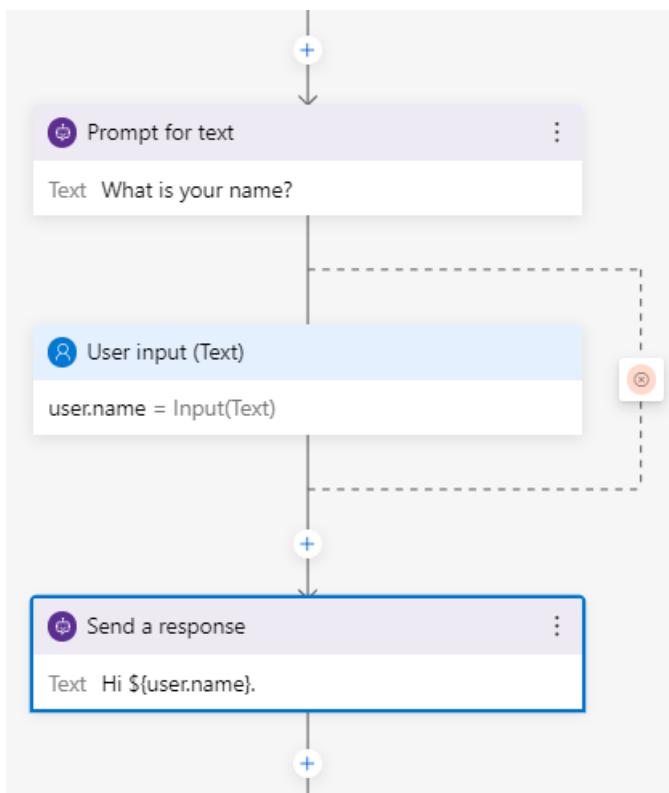
You can also refer to properties in [bot response templates](#). See [Language Generation](#) to learn more about the

bot response system.

To use the value of a property from memory inside a message, wrap the property reference in curly brackets and preface it with a `$` like this:  `${user.profile.name}`.

- [Composer v2.x](#)
- [Composer v1.x](#)

The screenshot below demonstrates how a bot can prompt a user for a value, then immediately use that value in a confirmation message.



In addition to getting property values, it's also possible to embed properties in [expressions](#) used in a message template. Refer to the adaptive expressions page for the full list of [pre-built functions](#).

Properties can also be used within a bot response template to provide conditional variants of a message and can be passed as parameters to both built-in and custom functions.

### Memory shorthand notations

There are a few short-hand notations supported to access specific memory scopes.

SYMBOL	USAGE	EXPANSION	NOTES
\$	<code>\$userName</code>	<code>dialog.userName</code>	Short hand for the dialog scope.
#	<code>#intentName</code>	<code>turn.recognized.intents.intentName</code>	Short hand for a named intent returned by the recognizer.
@	<code>@entityName</code>	<code>turn.recognized.entities.entityName</code>	returns the first and <i>only</i> the first value found for the entity, immaterial of the value's cardinality.

SYMBOL	USAGE	EXPANSION	NOTES
@@	<code>@@entityName</code>	<code>turn.recognized.entities.ent</code> <code>@@entityName</code> returns the actual value of the entity, preserving the value's cardinality.	
%	<code>%propertyName</code>	<code>class.propertyName</code>	Short hand for instance properties, such as <code>MaxTurnCount</code> , <code>DefaultValue</code> , and so on.

## Next

- [Bot responses](#) in Bot Framework Composer.
- See how to [Ask for user for input](#) for more information on using input actions.
- For a list of predefined properties in memory, see [Memory scopes and properties](#).

# About skills

5/26/2021 • 2 minutes to read

**APPLIES TO:** Composer v1.x and v2.x

You can extend a bot using another bot, a *skill*. A skill is available to various other bots, helping with reuse of conversational logic. In this way, you can create a user-facing bot and extend it by consuming your own or third party skills.

- A *skill* is a bot that can perform a set of tasks for another bot. A bot can be both a skill and a user-facing bot.
- A *skill consumer* is a bot that can call one or more skills. A user-facing skill consumer is also called a *root bot*.
- A *skill manifest* is a JSON file that describes the actions the skill can perform, its input and output parameters, and the skill's endpoints.
  - Composer can use the information in the manifest to connect a bot to a skill.
  - Composer can help you create a skill manifest for a bot.

The user interacts directly only with a root bot; however, the root bot can delegate some of its conversational logic to a skill.

You should already be familiar with some of the basics of bot design:

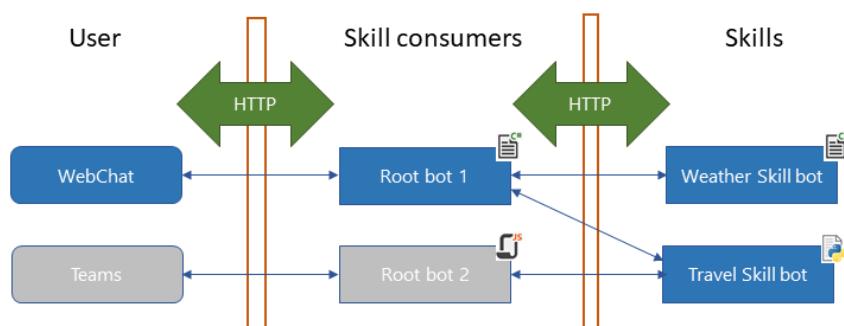
- [Templates](#)
- [Dialogs](#)
- [Triggers](#)
- [Conversation flow and memory](#)

## How it works

Skills are a feature of the Bot Framework SDK and Composer:

- Skills and skill consumers communicate over HTTP using the Bot Framework protocols.
- A skill consumer can consume multiple skills.
- A skill consumer can consume a skill developed in any language. For example, a C# bot can consume a skill implemented using Python.
- A skill can also be a skill consumer and call other skills.
- Skills support user authentication.

This diagram shows some of the possible permutations.



Different skill consumers can access both distinct and common skills. Skills and consumers can use different

programming languages.

## Bot-to-bot communication

It's important to understand certain aspects of this design, independent of which bot you're designing.

- The user-root conversation is different than the root-skill conversation.
  - Each consumer-skill conversation has a unique ID, different from the conversation between the user and the root bot.
  - A consumer-skill conversation ends when the skill completes its task or when the consumer cancels the conversation.
- The skill consumer and skill manage their own state separately.
- Because the root and skill bots communicate over HTTP, the instance of the consumer that receives an activity from a skill may not be the same instance that sent the initiating activity. Different servers may handle these two requests.

## Authentication

Service-level authentication is managed by the Bot Connector service. The framework uses bearer tokens and bot application IDs to verify the identity of each bot.

### IMPORTANT

All deployed bots (the skill consumer and any skills it consumes) must have valid application credentials.

When testing both a consumer and skill locally, you can test both bots without an app ID or password. However, an Azure subscription is still required to deploy your skill to Azure.

Claims are validated based on the bot's configuration file.

- You must update skills to allow specific consumers to call them, though you can also make the skill generally available.
- Composer updates the consumer configuration when you add a skill to the bot.

## Next steps

- To create and test a skill and skill consumer locally on your machine, see how to [Create a and test a local skill](#).
- To publish a bot as a skill, see how to [Publish a skill](#).
- To configure a bot to access an already published skill, see how to [Connect to a remote skill](#).

# Natural language processing in Composer

6/18/2021 • 2 minutes to read

**APPLIES TO:** Composer v1.x and v2.x

Natural language processing (NLP) is a technological process that enables computer applications, such as bots, to derive meaning from a user's input. To do this it attempts to identify valuable information contained in conversations by interpreting the user's needs (intents) and extract valuable information (entities) from a sentence, and respond back in a language the user will understand.

## Why do bots Need Natural Language Processing?

Bots are able to provide little to no value without NLP. It's what enables your bot to understand the messages your users send and respond appropriately. When a user sends a message with "Hello", it's the bot's Natural Language Processing capabilities that enable it to know that the user posted a standard greeting, which in turn allows your bot to leverage its AI capabilities to come up with a proper response. In this case, your bot can respond with a greeting.

Without NLP, your bot can't meaningfully differentiate between when a user enters "Hello" or "Goodbye". To a bot without NLP, "Hello" and "Goodbye" will be no different than any other string of characters grouped together in random order. NLP helps provide context and meaning to text or voice based user inputs so that your bot can come up with the best response.

One of the most significant challenges when it comes to NLP in your bot is the fact that users have a blank slate regarding what they can say to your bot. While you can try to predict what users will and will not say, there are bound to be conversations that you did not anticipate, fortunately Bot Framework Composer makes it easy to continually refine its NLP capabilities.

The two primary components of NLP in Composer are **Language Understanding** (LU) that processes and interprets *user input* and **Language Generation** (LG) that produces *bot responses*.

## Language Understanding

**Language Understanding** (LU) is the subset of NLP that deals with how the bot handles user inputs and converts them into something that it can understand and respond to intelligently.

### Additional information on Language Understanding

- The [Language Understanding](#) concept article.
- The [Advanced intent and entity definition](#) concept article.
- The [Using LUIS for Language Understanding](#) how to article.

## Language Generation

**Language Generation** (LG), is the process of producing meaningful phrases and sentences in the form of natural language. Simply put, it's when your bot responds to a user with human readable language.

### Additional information on Language Generation

- The [Language Generation](#) concept article.
- The [Language Generation](#) how to article.

## Summary

Natural Language Processing is at the core of what most bots do in interpreting users written or verbal inputs and responding to them in a meaningful way using a language they will understand.

While NLP certainly can't work miracles and ensure a bot appropriately responds to every message, it's powerful enough to make-or-break a bot's success. Don't underestimate this critical and often overlooked aspect of bots.

# Language generation

6/11/2021 • 6 minutes to read

**APPLIES TO:** Composer v1.x and v2.x

Language generation (LG) lets you define multiple variations of a phrase, execute simple expressions based on context, and refer to conversational memory. At the core of language generation lies template expansion and entity substitution. You can provide one-off variation for expansion as well as conditionally expand a template. The output from language generation can be a simple text string, multi-line response, or a complex object payload that a layer above language generation will use to construct a complete [activity](#). Bot Framework Composer natively supports language generation to produce output activities using the LG templating system.

You can use Language generation to:

- Achieve a coherent personality, tone of voice for your bot.
- Separate business logic from presentation.
- Include variations and sophisticated composition for any of your bot's replies.
- Construct cards, suggested actions and attachments using a [structured response template](#).

Language generation is achieved through:

- A Markdown based [.lg file](#) that contains the templates and their composition.
- Full access to the current [bot's memory](#) so you can data bind language to the state of memory.
- Parser and runtime libraries that help achieve runtime resolution.

## TIP

You can read the [composer best practices](#) article for some suggestions using LG in Composer.

## Templates

Templates are functions which return one of the variations of the text and fully resolve any other references to templates for composition. You can define one or more text responses in a template. When multiple responses are defined in the template, a single response will be selected at random.

You can also define one or more expressions using [adaptive expressions](#), so when it is a conditional template, those expressions control which particular collection of variations get picked. Templates can be parameterized, meaning that different callers to the template can pass in different values for use in expansion resolution. For additional information see [.lg file format](#).

Composer currently supports three types of templates: [simple response](#), [conditional response](#), and [structured response](#). You can read [define LG templates](#) to learn how to define each of them.

You can split language generation templates into separate files and refer to them from one another. You can use Markdown-style links to import templates defined in another file, like `[description text](file/uri path)`. Make sure your template names are unique across files.

### Anatomy of a template

A template usually consists of the name of the template, denoted with the # character, and one of the following:

- A list of one-off variation text values defined using "-"

- A collection of conditions, each with a:
  - conditional expression, expressed using [adaptive expressions](#) and
  - list of one-off variation text values per condition
- A structure that contains:
  - structure-name
  - properties

Below is an example of a simple LG template with one-off variation text values.

```
> this is a comment
# nameTemplate
- Hello ${user.name}, how are you?
- Good morning ${user.name}. It's nice to see you again.
- Good day ${user.name}. What can I do for you today?
```

## Define LG templates

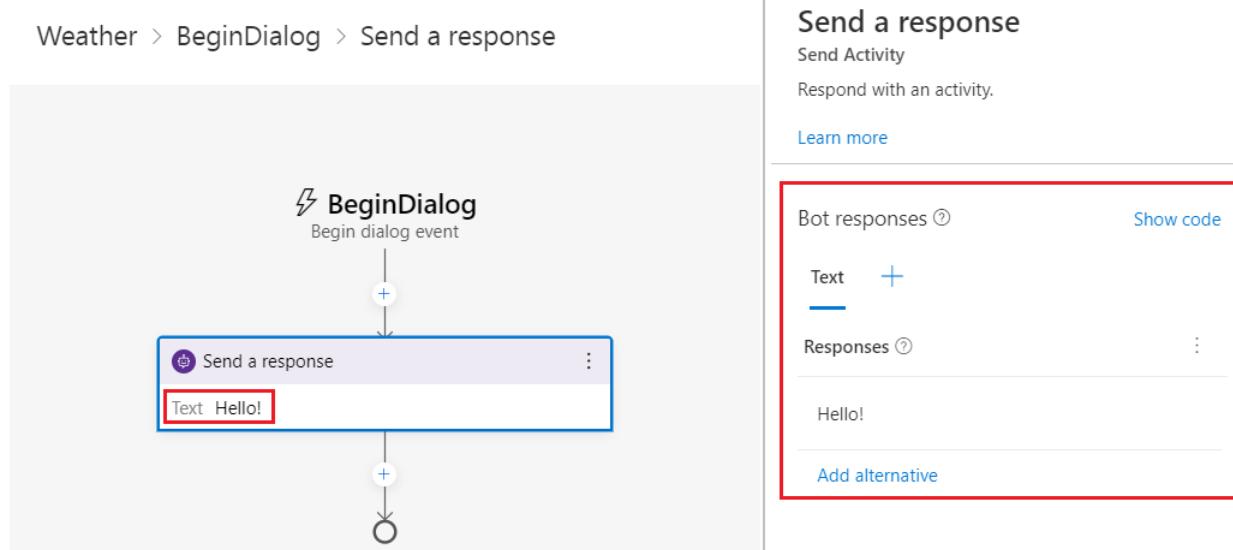
When you want to determine how your bot should respond to user input, you can define LG templates to generate responses. For example, you can define a welcome message to the user in the **Send a response** action. To do this, select the **Send a response** action node. You will see the inline response editor where you can define LG templates.

To define LG templates in Composer, you will need to know:

- the aforementioned LG concepts
- [.lg file format](#)
- [adaptive expressions](#)

You can define LG templates either in the inline response editor in the Properties panel or in the **Bot Responses** view, which lists all templates. Below is a screenshot of the response editor.

- [Composer v2.x](#)
- [Composer v1.x](#)



## TIP

When prompting the user for information, you can enter in a plain text string, seen above, or you can use any of the three insert buttons. Each results in a drop-down list containing relevant options to choose from:

The screenshot shows the 'Bot Responses' page in the Microsoft Bot Framework Composer interface. At the top, there's a 'Text' input field with a blue placeholder 'Hello!'. To the right of the input field are three small icons representing different insertion methods: a bot icon, a '{x}' placeholder, and an 'fx' icon. A red box highlights the top row of these three icons.

Reference an existing item in your [Bot Responses](#) template.

Reference a property in memory

Reference a pre-built adaptive expressions function.

All entries made in the inline response editor in the **Properties** panel will automatically be added to **Bot Responses**. To see them, select the **Bot Responses** icon (or the bot icon when collapsed) in the navigation pane to see all the LG templates defined in the bot categorized by dialog.

You will notice an entry in the list named **Common** that is not associated to a specific dialog. This is the LG template that is created by Composer, and it's designed to be shared by other dialogs in your bot. To enable a dialog to see the templates it contains, select **Show code** on the upper right corner of the **Bot Responses** page to enable editing, then add `[import](common.1g)` in that dialog's template (See screenshot below). This tells that dialog's template to import all the items in the common template. For additional information on importing templates, see [Importing external references](#) in the [.Lg file format](#) article.

- [Composer v2.x](#)
- [Composer v1.x](#)

The screenshot shows the Microsoft Bot Framework Composer interface. On the left, the navigation pane is open, showing various bot components like Home, Create, Configure, User input, Bot responses, Knowledge base, Publish, and Package manager. The 'Bot responses' item is selected. In the main area, the title bar says 'AskingQuestionsSample-0' and 'English (United States)'. The top right has buttons for 'Start bot', '|=', '...', and a gear icon. Below the title bar, there's a 'Bot Responses' section with a 'Hide code' link. Underneath is a tree view of templates: AskingQuestions..., Common, AskingQuestions..., Attachmentinput, choiceinput, confirminput, datetimeinput, numberinput, oauthinput, signout, and textinput. The 'Common' node is expanded. In the bottom right, there's a code editor window with the following code:

```
1 [import](common.1g)
2
3 # SendActivity_068558
4 -${WelcomeUser()}
5
6 # SendActivity_581197
7 -${WelcomeUser()}
```

Composer currently supports definitions of the following three types of templates: [simple](#), [conditional](#), and [structured response](#).

## Simple response template

A simple response template generates a simple text response. A simple response template can be a single line response, a text with memory, or a response of multiline texts. Use the - character before the response text or expression to returns. Here are a few examples of simple response templates from the [Responding with Text sample](#).

Here is an example of a single line text response:

```
- Here is a simple text message.
```

This is an example of a single line response using a variable:

```
- ${user.message}
```

Variables and expressions are enclosed in curly brackets \${}.

Select a property reference using the **Insert property reference** drop-down in the inline response editor:

The screenshot shows the Microsoft Bot Framework's inline response editor interface. At the top, there are two tabs: 'Text' (which is selected, indicated by a blue underline) and 'Code'. Below the tabs, there is a section titled 'Response Variations' with a small question mark icon. Underneath this section is a button labeled 'Add new variation' with a cursor icon pointing to it.

Here is an example of a multi-line response. It includes multiple lines of text enclosed in `````.

```
# multilineText
- ``` you have such alarms
    alarm1: 7:am
    alarm2: 9:pm
````
```

## Conditional response template

For all conditional templates, all conditions are expressed in [Adaptive expressions](#). Condition expressions are enclosed in curly brackets \${}. Here are two [conditional response template examples](#).

If-else

```
> time of day greeting reply template with conditions.
# timeOfDayGreeting
- IF: ${timeOfDay == 'morning'}
  - good morning
- ELSE:
  - good evening
```

## Switch

```
# TestTemplate
- SWITCH: ${condition}
- CASE: ${case-expression-1}
  - output1
- CASE: ${case-expression-2}
  - output2
- DEFAULT:
  - final output
```

## Structured response template

[Structured response templates](#) lets users to define a complex structure that supports all the benefits of LG (templating, composition, substitution) while leaving the interpretation of the structured response up to the bot developer. It provides an easier way to define an outgoing [activity](#) in a simple text format. Composer currently support structured LG templates to define cards and SuggestedActions.

The definition of a structured response template is as follows:

```
# TemplateName
> this is a comment
[Structure-name
  Property1 = <plain text> .or. <plain text with template reference> .or. <expression>
  Property2 = list of values are denoted via '|'. e.g. a | b
> this is a comment about this specific property
  Property3 = Nested structures are achieved through composition
]
```

Below is an example of SuggestedActions from the [Interruption Sample](#):

```
- Hello, I'm the interruption demo bot! \n \[Suggestions=Get started | Reset profile]
```

Below is an example of a Thumbnail card from the [Responding With Cards Sample](#):

```
# ThumbnailCard
[ThumbnailCard
  title = BotFramework Thumbnail Card
  subtitle = Microsoft Bot Framework
  text = Build and connect intelligent bots to interact with your users naturally wherever
  they are, from text/sms to Skype, Slack, Office 365 mail and other popular services.
  image = https://sec.ch9.ms/ch9/7fff/e07cfef0-aa3b-40bb-9baa-
7c9ef8ff7fff/buildreactionbotframework_960.jpg
  buttons = Get Started
]
```

## References

- [.lg file format](#)
- [Structured response template](#)

- Adaptive expressions

## Next

- Language understanding

# Language understanding

6/11/2021 • 6 minutes to read

**APPLIES TO:** Composer v1.x and v2.x

Language understanding (LU) is used by a bot to understand language naturally and contextually to determine what to do next in a conversation flow. In Bot Framework Composer, the process is achieved through setting up recognizers and providing training data in the dialog so that the *intents* and *entities* contained in the message can be captured. These values will then be passed on to triggers which define how the bot responds using the appropriate actions.

LU has the following characteristics when used in Bot Framework Composer:

- LU is training data for the LUIS recognizer.
- LU is authored in the inline editor or in **User Input** using the [.lu file format](#).
- Composer currently supports LU technologies such as LUIS.

## Core LU concepts in Composer

### Intents

Intents are categories or classifications of user intentions. An intent represents an action the user wants to perform. It is a purpose or goal expressed in the user's input, such as booking a flight, paying a bill, or finding a news article. You define and name intents that correspond to these actions. A travel app may define an intent named "BookFlight."

Here's a sample .lu file that demonstrates a simple **Greeting** intent with a list of example user utterances that capture different ways they might express this intent. You can use - or + or \* to denote lists. Numbered lists are not supported.

```
# Greeting
- Hi
- Hello
- How are you?
```

`#<intent-name>` defines a new intent. Each line after the intent definition contains an example utterance that describes that intent. You can stitch together multiple intent definitions in a language understanding editor in Composer. Each section is identified by `#<intent-name>` notation. Blank lines are skipped when parsing the file.

### Utterances

Utterances are inputs from users and may have a lot of variations. Since utterances are not always well-formed, we need to provide example utterances for specific intents to train bots to recognize intents from different utterances. By doing so, your bots will have some "intelligence" to understand human languages.

In Composer, utterances are always captured in a markdown list following the intent they refer to. For example, the **Greeting** intent with some example utterances are shown in the [Intents](#) section above.

## NOTE

You may have noticed that the LU format is very similar to the LG format, but they are different. LU is for bots to understand user's inputs (primarily capture **intent** and optionally **entities**) and it is associated with recognizers, while LG is for bots to respond to users as output, and it is associated with a language generator.

## Entities

Entities are a collection of objects, each consisting of data extracted from an utterance such as places, times, and people. Entities and intents are both important data extracted from utterances. An utterance may include zero or more entities, while an utterance usually represents one intent. In Composer, all entities are defined and managed inline. Entities in the [.lu file format](#) are denoted using `{\<entityName\>=\<labelled value\>}` notation. For example:

```
# BookFlight
- book a flight to {toCity=seattle}
- book a flight from {fromCity=new york} to {toCity=seattle}
```

The example above shows the definition of a `BookFlight` intent with two example utterances and two entity definitions: `toCity` and `fromCity`. When triggered, if LUIS is able to identify a destination city, the city name will be made available as `@toCity` within the triggered actions or a departure city with `@fromCity` as available entity values. The entity values can be used directly in expressions and LG templates, or stored into a property in [memory](#) for later use. For additional information on entities see the article [advanced intents and entities](#).

## Example

The table below shows an example of an intent with its corresponding utterances and entities. All three utterances share the same intent `BookFlight` each with a different entity. There are different types of entities, you can find more information in [.lu file format](#).

| INTENT     | UTTERANCES                                    | ENTITIES                |
|------------|-----------------------------------------------|-------------------------|
| BookFlight | "Book me a flight to London"                  | "London"                |
|            | "Fly me to London on the 31st"                | "London", "31st"        |
|            | "I need a plane ticket next Sunday to London" | "next Sunday", "London" |

Below is a similar definition of a `BookFlight` intent with entity specification `{city=name}` and a set of example utterances. We use this example to show how they are manifested in Composer. Extracted entities are passed along to any triggered actions or child dialogs using the syntax `@city`.

```
# BookFlight
- book a flight to {city=austin}
- travel to {city=new york}
- I want to go to {city=los angeles}
```

After publishing the model to LUIS, it will be able to identify a city as an entity and the city name will be made available as `@city` within the triggered actions. The entity value can be used directly in expressions and LG templates, or stored into a property in [memory](#) for later use. See the [Define intents with entities](#) article for examples of how these are implemented in Composer.

## Author .lu files in Composer

You author .lu files as training data for recognizers in Composer. You need to know:

- [Language Understanding concepts](#)
- [.lu file format](#)
- [Adaptive expressions](#)

To create .lu files in Composer, follow these steps:

### Set up a Recognizer Type

Select a dialog in the **Navigation** pane and then select **Default recognizer** from the **Recognizer Type** drop-down list in the **Properties** pane on the right side of the Composer screen.

Recognizer Type

Default recognizer

### Create an *Intent recognized* trigger

In the same dialog you selected the **Default recognizer** recognizer, select the three-dot icon then select **Add a trigger** from the drop-down menu.

Select **Intent recognized** from the trigger type menu. Fill in the **What is the name of this trigger** field with an [intent](#) name and add example [utterances](#) in the **Trigger phrases** field.

An **Intent recognized** trigger with an intent named **weather** and a few examples utterances can look like the following:

- [Composer v2.x](#)
- [Composer v1.x](#)

### Create a trigger

What is the type of this trigger?

Intent recognized

What is the name of this trigger?

weather

Trigger phrases

+ Add entity ▾   ▾ Insert entity ▾

|   |                                 |
|---|---------------------------------|
| 1 | - How is the weather in {city}? |
| 2 | - Tell me about the weather     |
| 3 | - Is it raining tomorrow?       |

Cancel

Submit

After selecting **Submit** you will see an **Intent recognized** trigger named **weather** in the **Navigation** pane

and the trigger node in the authoring canvas. You can edit the .lu file on the right side of the Composer screen in the **Properties** pane as shows in the image below:

- [Composer v2.x](#)
- [Composer v1.x](#)

The screenshot shows the Microsoft Bot Framework Composer interface. On the left, the navigation bar indicates 'WeatherBot > weather'. The main workspace contains a trigger node labeled '#weather Intent recognized' with a plus sign (+) icon. To the right, the properties pane is open for the '#weather' intent, titled 'weather Intent recognized'. It describes the intent as 'Actions to perform when specified intent is recognized.' A red box highlights the 'Trigger phrases' section, which lists sample phrases: 1. - How is the weather in {city}?, 2. - Tell me about the weather, and 3. - Is it raining tomorrow?. Below this is the 'Intent name: #weather' field.

Select **User Input** from the Composer menu to view all the LU templates created. Select a dialog from the **Navigation** pane then select **Show code** to edit the LU template.

- [Composer v2.x](#)
- [Composer v1.x](#)

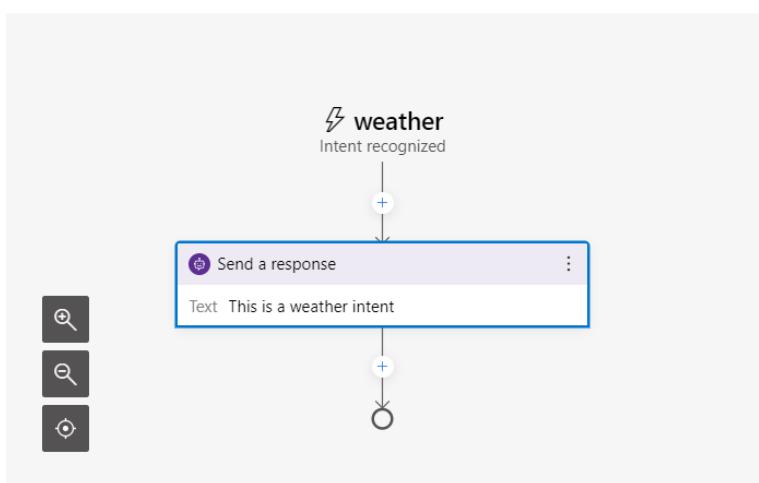
The screenshot shows the 'User Input' list in the Microsoft Bot Framework Composer interface. The left sidebar shows a tree structure with 'WeatherBot' expanded, showing 'WeatherBot', 'Weather', 'Help', and 'getWeather'. The main table lists the user input entries. One entry is highlighted with a red border: '#weather' under 'Intent', with sample phrases '- How is the weather?', '- Tell me about the weather', and '- Is it raining tomorrow?'. The 'Defined in:' column shows 'WeatherBot' and the 'Activity' column shows 'Not yet published'. A 'Show code' button is located at the top right of the table.

| User Input | Intent   | Sample Phrases                                                                    | Defined in: | Activity          |
|------------|----------|-----------------------------------------------------------------------------------|-------------|-------------------|
| #weather   | #weather | - How is the weather?<br>- Tell me about the weather<br>- Is it raining tomorrow? | WeatherBot  | Not yet published |

### Add action(s) to the Intent recognized trigger

Select **+** under the **Intent recognized** trigger and add any action(s) you want your bot to execute when the **weather** trigger is fired.

- [Composer v2.x](#)
- [Composer v1.x](#)



## Send a response

### Send Activity

Respond with an activity.

[Learn more](#)

Bot Responses [?](#)

[Switch to code editor](#)

Text [+](#)

Response Variations [?](#)

This is a weather intent

[Add new variation](#)

## Publish LUIS to LUIS

The last step is to publish your .lu files to LUIS.

Select **Project Settings** from the Composer menu. On the **Bot management and configurations** page, enter values into the **Luis authoring key** and **Luis region** fields.

- [Composer v2.x](#)
- [Composer v1.x](#)

The screenshot shows the 'Bot management and configurations' page for 'WeatherBot'. On the left is a sidebar with links: Home, Design, Bot Responses, User Input, QnA, Diagnostics, Publish, and Project Settings (which is selected). The main area has a heading 'Bot management and configurations' and a note about security settings. It contains two input fields: 'LUIS authoring key \*' with a placeholder 'Enter LUIS authoring key' and a red error message 'LUIS key is required with the current recognizer setting to start your bot locally, and publish'; and 'LUIS region \*' with a placeholder 'Enter LUIS region'. A blue button 'Get LUIS keys' is at the bottom.

If you have not created a LUIS authoring key, or do not know how to find these values, you can do either by selecting the **Get LUIS keys** button

## Select LUIS keys



Choose from existing LUIS keys, create a new LUIS resource, or generate a request to handoff to your Azure admin. [Learn more](#)

- Choose from existing
- Create a new LUIS resource
- Handoff to admin

[Next](#)

[Cancel](#)

If you need to handoff this task to your Azure admin, select **Handoff to admin** for instructions:

## Share resource request

X

Copy and share this information with your Azure admin to provision resources on your behalf.

### Instructions for your Azure admin



I am working on a Microsoft Bot Framework project, and I now require some Azure resources to be created. Please follow the instructions below to create these resources and provide them to me.

1. Using the Azure portal, please create a Language Understanding resource on my behalf.
2. Once provisioned, securely share the resulting credentials with me as described in the link below.

Detailed instructions:

<https://aka.ms/bfcomposerhandoffluis>

Back

Okay

Any time you start or restart your bot project, Composer will evaluate if your LU content has changed. If so Composer will automatically make the required updates to your LUIS applications then train and publish them. If you go to your LUIS app website, you will find the newly published LU model.

## References

- [What is LUIS](#)
- [Language Understanding](#)
- [.lu file format](#)
- [Adaptive expressions](#)
- [Using LUIS for language understanding](#)
- [Extract data from utterance text with intents and entities](#)

## Next

- Learn how to [send messages to users](#).

# Best practices for building bots using Composer

6/11/2021 • 13 minutes to read

**APPLIES TO:** Composer v1.x and v2.x

Bot Framework Composer is a visual authoring tool for building conversational AI software. By learning the concepts described in this section, you'll become equipped to design and build a bot using Composer that aligns with the best practices. Before reading this article, you should read the [introduction to Bot Framework Composer](#) article for an overview of what you can do with Composer.

Use the basic authoring process to build your bots:

- [Create a bot](#)
- Create primary conversation flows by
  - adding [triggers](#) to dialogs
  - adding [actions](#) to triggers
  - authoring [language understanding](#) for user input
  - authoring [language generation](#) for bot responses
  - manipulating [memory](#)
- Integrate with APIs
- Add greater natural language complexity using entity binding and interruption support

The following list includes the recommended best practices and things to avoid for building bots with Composer:

| RECOMMENDED                                                     | NOT RECOMMENDED                                      |
|-----------------------------------------------------------------|------------------------------------------------------|
| <a href="#">Plan your bot</a>                                   | -----                                                |
| <a href="#">Give your bot a bit of personality</a>              | <a href="#">Make your bot talk too much</a>          |
| <a href="#">Consider when to use dialogs</a>                    | <a href="#">Nest more than two deep conditionals</a> |
| <a href="#">Consider the non-happy path</a>                     | -----                                                |
| <a href="#">Name dialogs clearly</a>                            | -----                                                |
| <a href="#">Keep dialogs short</a>                              | -----                                                |
| <a href="#">Create a dialog for help/cancel</a>                 | -----                                                |
| <a href="#">Keep a root menu</a>                                | -----                                                |
| <a href="#">Keep state properties consistent</a>                | -----                                                |
| <a href="#">Define LG templates and reuse them consistently</a> | -----                                                |
| <a href="#">Parameterize reusable LG templates</a>              | -----                                                |

| RECOMMENDED                                                    | NOT RECOMMENDED |
|----------------------------------------------------------------|-----------------|
| Add variations for bot responses                               | -----           |
| Make your prompt text clear                                    | -----           |
| Prepare for ambiguity in the responses                         | -----           |
| Add prompt properties                                          | -----           |
| Use LUIS prebuilt entities                                     | -----           |
| Use LUIS entities to support synonyms in a multi-choice prompt | -----           |

## Design bots

### Plan your bot

Before building your bot application, make a plan of the bot you want to create. Ensuring a great **Conversational User Experience (CUX)** should be your number one priority when designing a bot. Consider the following questions:

- **What your bot is used for?** Determine the kind of bot you plan to build. This will help to define the functionalities you want to implement in the bot.
- **What problems does your bot intend to solve?** Be clear about the problems your bot intends to solve. Solving problems for customers is the top factor you should consider when building bots. You should also consider things such as how to solve the user's problem better/easier/faster than any of the alternative experiences.
- **Who will use your bot?** If you are designing a bot, it's safe to assume that you are expecting users to use it. Different customers will expect different user experiences. This will also determine the complexity you should consider in your bot design. Consider what languages to implement for the bot.
- **Where will your bot run?** You should decide the platforms your bot will run on. For example, a bot designed to run on a mobile device will have more features like sending SMS to implement.

#### TIP

Most successful bots have at least one thing in common: great CUX. The [CUX guide](#), contains guidance on designing a bot. This guidance aligns with best practices and capitalizes on lessons learned.

### Give your bot a bit of personality

Adding personality to your bot makes it more conversational and engaging. Here are some tips to give your bot a bit of personality:

- Use [language generation](#) to [create multiple variations](#) of messages. A little bit of personality goes a long way, don't over use it.
- Consider the context where your bot will be used. Bots used in private scenarios can turn out to be more conversational than bots in public. A bot will be more descriptive to a new user than to an returning user.
- Define [language generation templates](#) and [reuse them](#) across the bot consistently. This will make your bot's personality consistent.
- Use [cards](#) to give your bots visual branding and personality if your platform supports UI cards.

### Users care when the bot solves their query

A great conversational bot does not require users to type too much, talk too much, repeat themselves several times, or explain things that the bot should automatically know and remember. Being concise and clear in messages is highly recommended in your design. Make the messages your bot sends relevant and information-dense. Don't say less or more than the conversation requires.

## Design dialogs

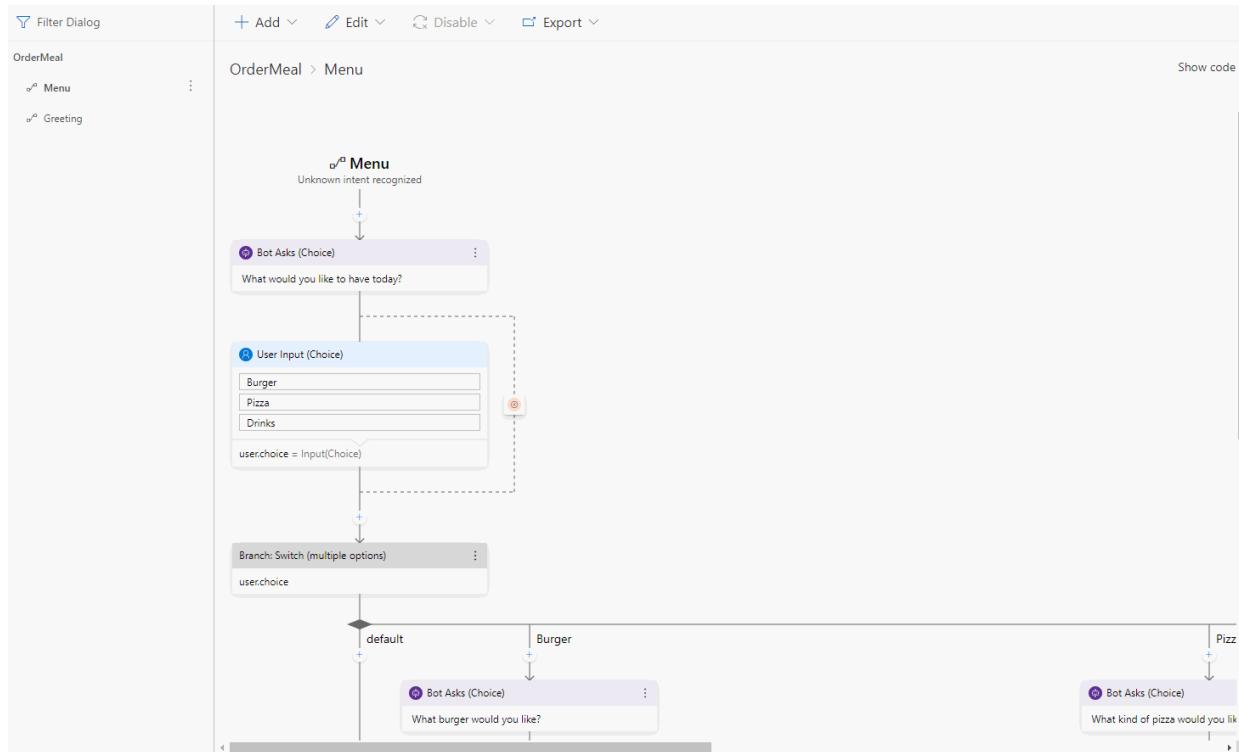
### Consider when to use dialogs

Think of [dialogs](#) as modular pieces with specific functionalities. Each dialog contains instructions for how the bot will react to the input. Dialogs allow you more granular control of where you start and restart, and they allow you to hide details of the "building blocks" that people do not need to know.

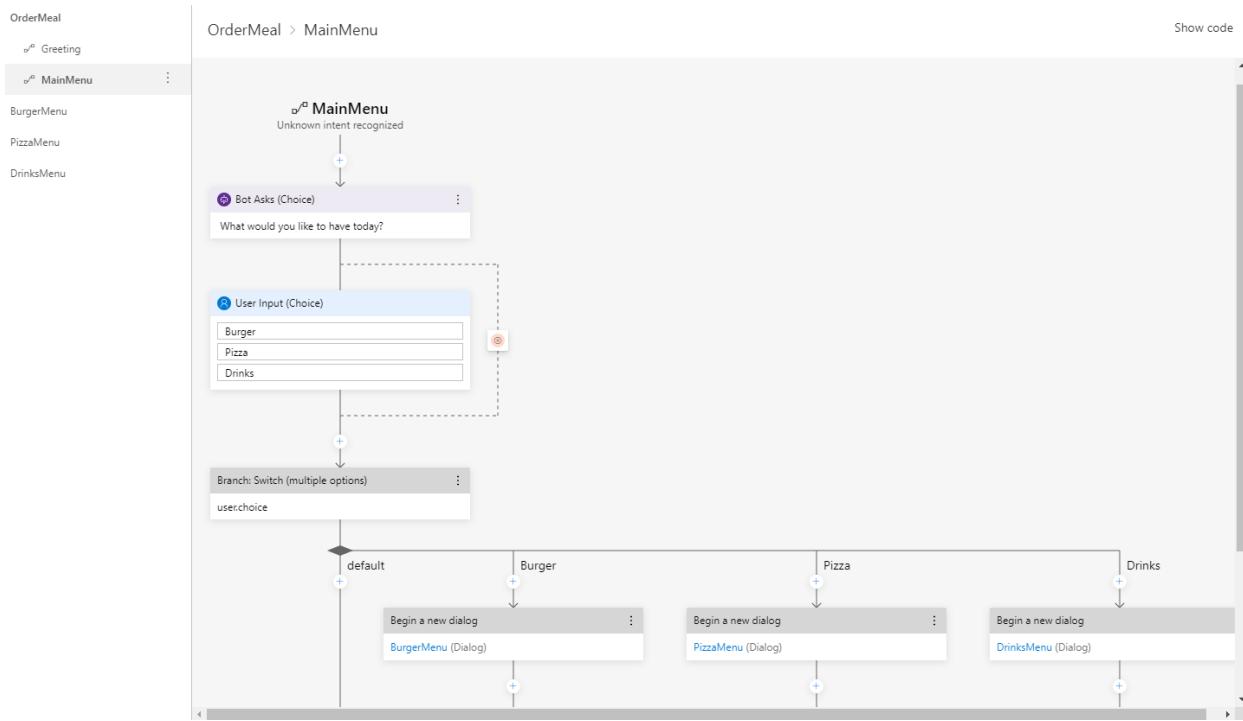
Consider using dialogs when you want to:

- Reuse things.
- Have interruptions that are local to that flow (for example, contextual help inside a date collection flow).
- Have a place in your conversation that you need to jump to easily from other places.
- [Nest more than two deep conditionals](#) within a dialog.

The following example shows a bot which nests two switch statements. This is inefficient and hard to read.



Instead of using nested switch statements, you can use dialogs to encapsulate the functionalities.



## Consider the non-happy path

When designing dialogs, you should consider asking follow-up questions, clarifications, and whether you want to use local interruption which only happens within the context of a child dialog.

- **Prepare for unexpected responses.** When your bot asks questions such as "What's your name?", get your bot prepared for answers such as "Why?" and "No". You can make use of prompt capabilities such as using *Unrecognized prompt* and *Invalid prompt* properties. Read how to make use of them in the [add prompt properties](#) section.
- **Use interruptions.** Consider using the *Allow Interruptions* property to either handle a global interruption or a local interruption within the context of the dialog.

### TIP

The *Allow Interruptions* property is located in the **Properties** panel of the **Other** tab of any prompt actions. You can set the value to be `true` or `false`.

## Name dialogs clearly

You should name your dialogs clearly when you create them as you cannot change the name of your dialog once created. You should use a naming scheme early as you might end up with a lot of dialogs.

Some commonly used naming schemes include but are not limited to the following:

- Camel case: `myDialog`
- Pascal case: `MyDialog`
- Snake case: `my_dialog`
- Kebab Case: `my-dialog`

### NOTE

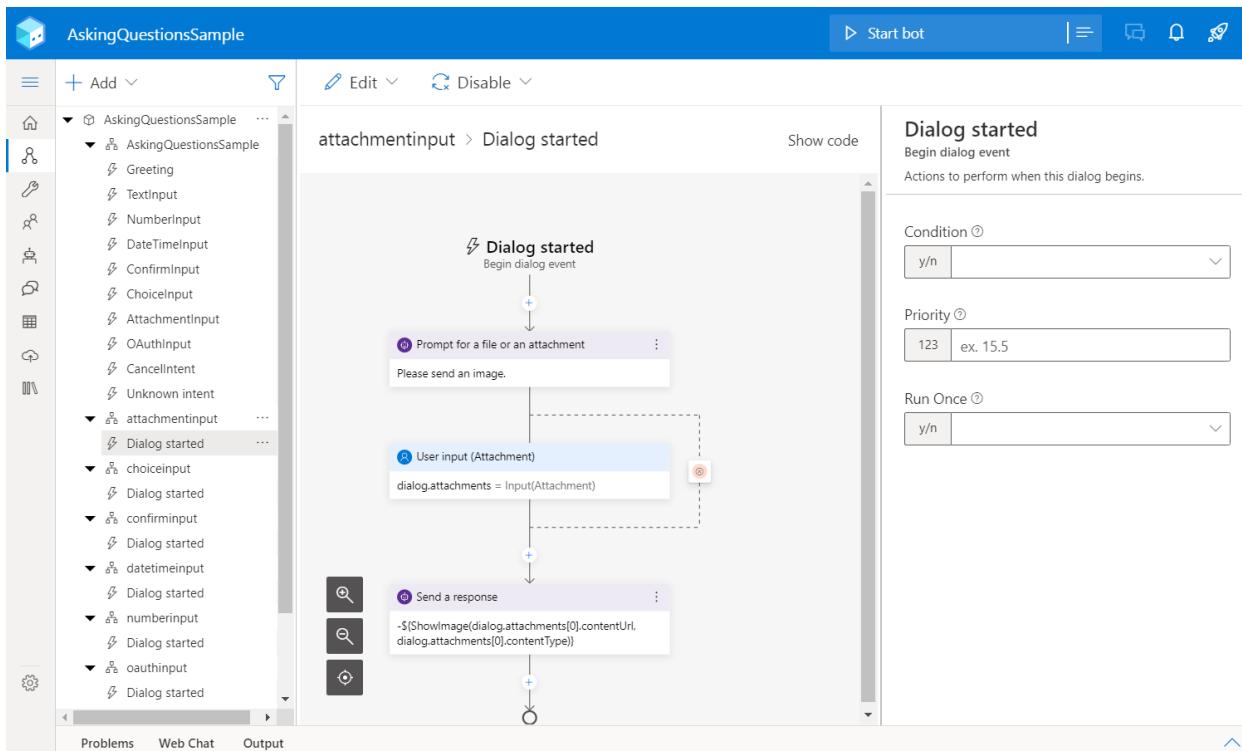
Don't use spaces and special characters in dialog names.

## Keep dialogs short

It's easy for you to view shorter dialogs in the authoring canvas and see all the "hidden details" in a dialog when

you want to. You should also try to limit the complexity of your dialog system. See some examples in the [AskingQuestionsSample](#). You can click through and see the dialogs.

- [Composer v2.x](#)
- [Composer v1.x](#)



## Create a dialog for help/cancel

It is a good practice to create a dialog for help/cancel, either locally or globally. By creating global help/cancel dialog, you can let users exit out of any process at any time and get back to the main dialog flow. Local help/cancel dialog will be very helpful to handle interruptions within the context of a dialog.

You can read more in the [weather bot tutorial- adding help and cancel functionality to your bot](#) article to see how to create a dialog for help/cancel functionalities.

## Design conversation flows

### Keep a root menu

It is a good practice to present a root menu with some common choices like at the beginning of the interaction. That menu should come back into play when the user completes a task. Think of this root menu is like an app's home screen. You begin your interaction with the app from the home screen and end with the home screen. Then you can take your next action.

### Keep state properties consistent

It's useful to establish conventions for your state properties across `conversation`, `user`, and `dialog` state for consistency and to prepare for context sharing scenarios.

When creating a property, think about the lifetime of that property. Do not hesitate to use `turn` memory scope to capture volatile information. There is no point using `dialog.confirmation.outcome` when you are asking a confirmation prompt because the lifetime of that confirmation is only for that particular turn of the conversation. You should not feel bad about using that `turn` memory scope. The [memory](#) concept article contains basic concepts of memory scopes used in Composer.

### Don't nest more than two deep conditionals

When you nest more than two deep conditionals (loops or switch statements) in a single dialog, you should

consider building child dialogs to encapsulate them. Making use of child dialogs to encapsulate bots functionalities is highly recommended for maintaining conversation flows.

## Design Language Generation

### Define LG templates and reuse them consistently

Reusing LG templates helps to keep consistency across the bot. You can define LG templates in advance in the `common.1g` file and use them in different dialogs when necessary. In Composer, you can select **Bot Responses** and then select **All** in the navigation to see the templates defined and shared by all the dialogs.

To import the common templates into another LG file, add the line `[import](common.1g)` at the top.

It is a good practice to build templates for reusable components and use them consistently. For example, things like acknowledgement phrases and apology phrases will be used by a bot in several different places, like every time you have an invalid prompt or unrecognized intent.

Here is an example of an `acknowledgePhrase` template with three variations:

```
#acknowledgePhrase
- I'm sorry you are having this problem. Let's see if there is anything we can do.
- I know it is frustrating - let's see how we can help...
- I completely understand your situation. Let me try my best to help.
```

### Parameterize reusable LG templates

For things like prompts, for texts in send activity, composition is one of the key things in LG, which enables reusability but also enables parameterization. You can have some lowest unit of composition.

When defining reusable templates, parameterize them so they can be used in different scenarios by passing in the appropriate options for expansion or evaluation.

- You can set up templates with parameters that take properties as parameters, so you can do things like date formatters or card builders.
- When you set up a template, instead of referring directly to a specific property `Dialog.foo`, use a parameter name instead. For example, in the code below, specify `name` as a parameter instead of referring to a property.
- Set up cards in parts and compose them into bigger cards.

Example:

```
# welcomeUser(name)
- ${greeting()}, ${ personalize(name)}

# greeting
- Hello
- Howdy
- Hi

# personalize(name)
- IF: ${name != ''}
- ${ name }
- ELSE:
- HUMAN
```

Having set up these templates, you can now use them in a variety of situations. For example:

```
> Greet a user whose name is stored in `user.name`  
- ${ welcomeUser(user.name) }  
  
> Greet a user whose name you don't know:  
- ${ welcomeUser() }  
  
> Use personalization in another message:  
- That's ok, ${ personalize(user.name) } , we can try again!
```

#### TIP

Read more in the [.lg file format](#) and [structured response template](#) articles.

## Add variations for bot responses

Language generation lets you define multiple variations of a phrase to make bots replies less robotic. For example, in the [weather bot tutorial - adding language generation](#) article, you can define multiple variations of a greeting message to make the conversation more natural. However, this does not mean you should add as many variations as possible, it depends on how you want your bot to respond and in what tone and voice.

## Design inputs

### Make your prompt text clear

Make sure your prompt texts are clear and unambiguous. Ambiguity is a problem in languages and it is something we should avoid when we phrase the text of a prompt.

Consider giving your user input hints including using suggested responses. This will help make your prompt clear and avoid ambiguity. For example, instead of saying "What is your birthday", you can say "What is your birthday? Please include the day month and year in the form of DD/MM/YYYY".

### Prepare for ambiguity in the responses

While ambiguity is something you try to avoid in outgoing messages, you should also be prepared for ambiguity in the incoming responses from users. This helps to make your bot perform better but also prepares you for platforms like voice where users more commonly add words.

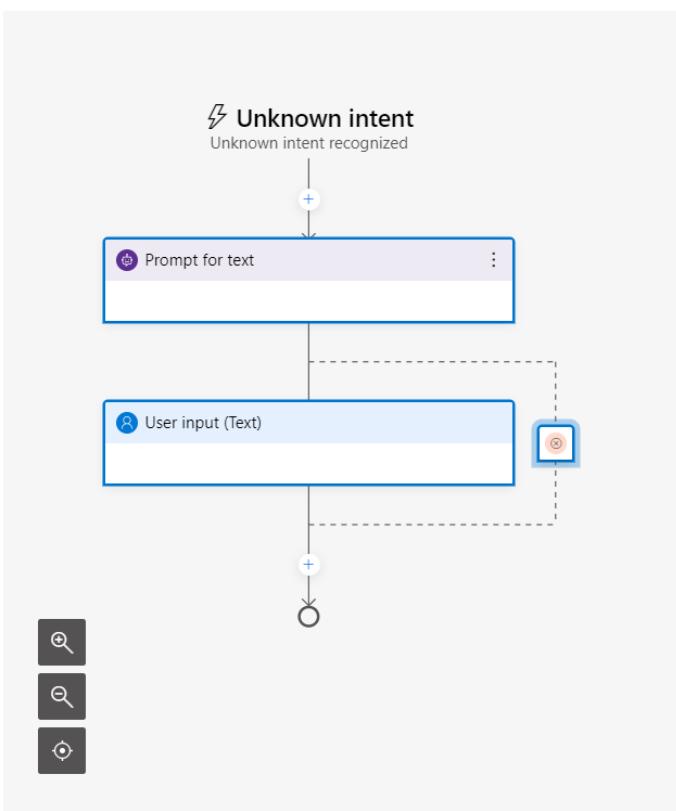
When people are talking out loud, they tend to add words in their responses than when they are typing into a text box. For example, in a text box they will just say "my birthday is 1/25/78" while the spoken input can be something like "my birthday is in January, it's the 25th".

Sometimes when people make their bots personality rich they introduce language ambiguity. For example, be cautious when you use greeting messages such as "What's up?", which is a question that users will try to answer. If you don't prepare your bot to responses like "Nothing", it will end up confusion.

### Add prompt properties

Make use of the prompt features such as *Unrecognized prompt* and *Invalid prompt*. These are powerful properties that give you a lot of control over how your bot responds to unrecognized and invalid answers. Access these properties under the **Other** tab of any type of input (**Ask a question**) action.

- [Composer v2.x](#)
- [Composer v1.x](#)



## Prompt for text

Text Input

Collection information - Ask for a word or sentence.

[Learn more](#)[Bot response](#) [User input](#) [Other](#)> [Recognizers](#)> [Validation](#)▽ [Prompt Configurations](#)Default value response [Show code](#)Text [+](#)Responses [?](#)[Add alternative](#)Max turn count [?](#)

123 | 3

Default value [?](#)

abc | ex. hello world, Hello \${user.name}, user.lastN...

Allow Interruptions [?](#)

... | ...

Add guidance along the prompts for re-prompts, otherwise the bot will keep asking the same question without telling the users why it is asking again.

Use validations when possible. The *Invalid prompt* fires when the input does not pass the defined validation rules. Here are two examples of how to phrase in the *Unrecognized prompt* and *Invalid prompt* fields.

- **Unrecognized prompt**

- Sorry, I do not understand '\${this.value}'. Please enter a zip code in the form of 12345.

- **Invalid prompt**

- Sorry, '\${this.value}' is not valid. I'm looking for a 5 digit number as zip code. Please specify a zip code in the form 12345.

## Design recognizers

### Use LUIS prebuilt entities

Luis provides a list of prebuilt entities which are very handy to use. When you think of defining entities, check the list of [LUIS prebuilt entities](#) first instead of reinventing your own wheels. Some commonly-used prebuilt entities include: `time`, `date` and `Number`.

For the best practices of building LUIS models, you should read the [best practices for building a language understanding \(LUIS\) app](#) article.

### Use LUIS entities to support synonyms in a multi-choice prompt

It is possible to use LUIS entities as part of a multi-choice prompt so that the bot can understand a wider set of replies. This can be achieved using the [LUIS list entity](#), which you can define **Expected responses** in the **User Input** section of a multi-choice prompt's **properties panel**. For example:

```
- American food  
- Mexican food  
- Asian food
```

```
@ list foodType =
```

```
- American food:  
- burger  
- pizza  
- hot dog
```

```
- Mexican food:
```

```
- taco  
- torta  
- posole
```

```
- Asian food:
```

```
- pho  
- noodle  
- dim sum
```

The following image demonstrates how this appears in the Authoring canvas and Properties pane in Composer:

The screenshot shows the Microsoft Bot Composer interface. On the left is the Authoring canvas, displaying a sequence of actions:

- A User input (Text) action with the code: `dialog.itemTitle = Input(Text)`.
- An AskForFoodType action with the question: "- What type of food are you interested in?".
- A User input (Choice) action with three options: "American food", "Mexican food", and "Asian food". The code for this action is: `dialog.foodType = Input(Choice)`.
- An Edit an Array property action with the code: `push user.lists[dialog.listType]`.

On the right is the Properties pane for the AskForFoodType action, titled "AskForFoodType Choice input". It includes the following sections:

- Bot response:** abc dialog.foodType
- User input:** abc value
- Value:** abc =@foodType
- Expected responses:** A list of 16 items corresponding to the food types defined in the code.
- Intent name:** #ChoiceInput\_Response\_878594

In the example above, `burger`, `pizza`, and `hot dog` will be recognized as the same entity as `American food`; `taco`, `torta`, and `posole` will be recognized as the same entity as `Mexican food`; and `pho`, `noodles`, and `dim sum` will be recognized as the same entity of `Asian food`.

- [Composer v2.x](#)
- [Composer v1.x](#)

You can also see a good implementation of this in the `ToDoWithLuisSample`. Do the following to get the To Do with LUIS example running in Composer:

1. Clone the [Bot Builder samples github repo](#) onto your machine.
2. Within the `composer-samples` folder you'll find C# and JavaScript projects, choose a language and navigate into the `projects` subfolder.
3. In this folder you'll find a `ToDoWithLuisSample` project which you can open in Composer.

Now that you have it loaded in Composer, you can select the **additem** dialog from the navigation pane. Then, in the **BeginDialog** trigger select the multi-choice prompt as follows:

The screenshot shows the Microsoft Bot Framework Composer interface. On the left, a navigation pane lists triggers: WelcomeUser, Add, Delete, View, UserProfile, whatCanYouDo, cancel, Unknown intent, additem, deleteitem, help, userprofile, and viewitem. The main area is titled "ToDoBotWithLuisSample-6" and contains a large gray hexagonal icon with two small circles inside. Below the icon is the text "Select a trigger on the left navigation to see actions". On the right, a configuration panel is open for the "ToDoBotWithLuisSample-6" dialog. It includes sections for "Language Understanding" (set to "Adaptive dialog"), "Recognizer Type" (set to "Default recognizer"), "Auto end dialog" (set to "boolean" with value "true"), and "Default result property" (set to "dialog.result"). A "Show code" button is at the top right of the configuration panel.

## Additional information

- [Best practices for building a language understanding \(LUIS\) app](#)
- [Best practices for creating a QnA Maker knowledge base](#)

# Conversational user experience

6/18/2021 • 3 minutes to read

**APPLIES TO:** Composer v1.x and v2.x

Bot Framework Composer enables creators to design conversational bots, virtual agents, digital assistants, and all other dialog interfaces—offering flexible, accessible, and powerful ways to connect with customers, employees, and one another. Whether you’re a novice designer or a veteran developer, the concepts described in this section will offer insights to help you craft an effective, responsible, inclusive, and, we hope, delightful experience that tackles a variety of business scenarios.

We define **Conversational User Experience (CUX)** as a modality of interaction that’s based on natural language. When interacting with each other, human beings use conversation to communicate ideas, concepts, data, and emotional information. CUX allows us to interact with our devices, apps, and digital services the way we communicate with each other, using phrasing and syntax via voice and text or chat that come naturally.



Other modalities can burden users with the task of learning interaction behaviors that are meaningful to the system: the syntax of a command line, the information architecture of a graphical user interface, or the touch affordances of a device. CUX turns the tables. Instead of users having to learn the system, it's the system that learns. It learns what we teach it about human language – patterns of speech, colloquialisms, chit-chat, even abusive words – so that it can respond appropriately.

## A great conversational bot

Most successful bots have at least one thing in common: a great conversational user experience. CUX can be multi-modal – employing text or voice, with or without visual, auditory, or touch enabled components. But fundamentally, CUX is human language.

### TIP

Regardless of the type of bot you're creating, make CUX a top priority.

If you are designing a bot, assume that users will prefer the bot experience over alternative experiences like apps, websites, phone calls with live agents, and other means of addressing their particular queries. Therefore, ensuring a great conversational user experience should be your number one priority when designing a bot. Some key considerations include:

- Does the bot easily solve the user's problem with minimal back and forth turns?
- Does the bot solve the user's problem better/easier/faster than any of the alternative experiences?
- Does the bot run on the devices and platforms the user cares about?
- Is the bot discoverable and easy to invoke?
- Does the bot guide the user when they are stuck; either via handover to a live agent or by providing relevant help?

Users care when the bot solves their query. A great conversational bot does not require users to type too much,

talk too much, repeat themselves several times, or explain things that the bot should automatically know and remember.

## The CUX guide

The CUX guide, contains guidance on designing a bot. This guidance aligns with best practices and capitalizes on lessons learned. The authors and designers of this guidance are drawing from combined decades of experience building and deploying conversational UX for a variety of bots, virtual agents, and other conversational experience projects including Cortana, Bot Framework Templates, Microsoft Virtual Assistant, Personality Chat, and others.

### TIP

Download the [CUX Guide Microsoft.pdf](#)

This CUX guide is divided loosely into a few different sections:

- An introduction to CUX, ethics and inclusive design.
- A brainstorming worksheet and guidelines for planning and designing.
- Practical development tips for building CUX experiences.

Read the topics in order, or jump to the area that addresses your needs.

### NOTE

A note on terminology: the guide explores several kinds of conversational experiences, including bots, virtual agents, and digital assistants. We use those terms relatively interchangeably because the principles of CUX design in this guidance apply to all, but we recognize there are distinctions in the industry. Our intention is to offer guidance that will help with most text-based conversational experiences, regardless of their intent.

## Additional information

- [Best practices for building bots using Composer](#)
- [Language generation in Composer](#)
- [Respond to users with cards in Composer](#)
- [Speech in Composer](#)

# Composer extensions

5/20/2021 • 13 minutes to read

**APPLIES TO:** Composer v1.x and v2.x

It's possible to extend and customize the behavior of Composer by installing extensions. Extensions can hook into the internal mechanisms of Composer and change they way they operate. Extensions can also *listen to* the activity inside Composer and react to it.

## What is a Composer extension?

Composer extensions are JavaScript modules. When loaded into Composer, the module is given access to a set of Composer APIs which can then be used by the extension to provide new functionality to the application. Extensions do not have access to the entire Composer application—they are granted limited access to specific areas of the application, and must adhere to a set of interfaces and protocols.

## Extension endpoints

Extensions currently have access to the following functional areas:

- [Authentication and identity](#) - extensions can provide a mechanism to gate access to the application, as well as mechanisms used to provide user identity.
- [Storage](#) - extensions can override the built in filesystem storage with a new way to read, write and access bot projects.
- [Web server](#) - extensions can add additional web routes to Composer's web server instance.
- [User interface](#) - extensions can add custom user interfaces to different parts of the application
- [Publishing](#) - extensions can add publishing mechanisms.
- [Runtime templates](#) - extensions can provide a runtime template used when *ejecting* from Composer.

Combining these endpoints, it is possible to achieve scenarios such as:

- Store content in a database
- Require login via AAD or any other oauth provider
- Create a custom login screen
- Require login via GitHub, and use GitHub credentials to store content in a Git repo automatically
- Use AAD roles to gate access to content
- Publish content to external services such as remote runtimes, content repositories, and testing systems.
- Add custom authoring experiences inside Composer

## How to build a extension

Extension modules must come in one of the following forms:

- Default export is a function that accepts the Composer extension API
- Default export is an object that includes an `initialize` function that accepts the Composer extension API
- A function called `initialize` is exported from the module

Extensions can be loaded into Composer with the following method:

- The extension is placed in the `/extensions/` folder, and contains a `package.json` file with a configured

`composer` definition.

The simplest form of a extension module is below:

```
export default async (composer: any): Promise<void> => {

    // call methods (see below) on the composer API
    // composer.useStorage(...);
    // composer.usePassportStrategy(...);
    // composer.addWebRoute(...)

}
```

## Authentication and identity

To provide auth and identity services, Composer has in large part adopted [PassportJS](#) instead of implementing a custom solution. Extensions can use one of the [many existing Passport strategies](#), or provide a custom strategy.

### `composer.usePassportStrategy(strategy)`

Configure a Passport strategy to be used by Composer. This is the equivalent of calling `app.use(passportStrategy)` on an Express app. [See PassportJS docs](#).

In addition to configuring the strategy, extensions will also need to use `composer.addWebRoute` to expose login, logout and other related routes to the browser.

Calling this method also enables a basic auth middleware that is responsible for gating access to URLs, as well as a simple user serializer/deserializer. Developers may choose to override these components using the methods below.

### `composer.useAuthMiddleware(middleware)`

Provide a custom middleware for testing the authentication status of a user. This will override the built-in auth middleware that is enabled by default when calling `usePassportStrategy()`.

Developers may choose to override this middleware for various reasons, such as:

- Apply different access rules based on URL
- Do something more than check `req.isAuthenticated`, such as validate or refresh tokens, make database calls, and provide telemetry.

### `composer.useUserSerializers(serialize, deserialize)`

Provide custom serialize and deserialize functions for storing and retrieving the user profile and identity information in the Composer session.

By default, the entire user profile is serialized to JSON and stored in the session. If this is not desirable, extensions should override these methods and provide alternate methods.

For example, the below code demonstrates storing only the user ID in the session during serialization, and the use of a database to load the full profile out of a database using that id during deserialization.

```
const serializeUser = function(user, done) {
    done(null, user.id);
};

const deserializeUser = function(id, done) {
    User.findById(id, function(err, user) {
        done(err, user);
    });
};

composer.useUserSerializers(serializeUser, deserializeUser);
```

### **composer.addAllowedUrl(url)**

Allow access to a URL, without requiring authentication. The `url` parameter can be an Express-style route with wildcards, such as `/auth/:stuff` or `/auth(.*)`.

This is primarily for use with authentication-related URLs. While the `/login` route is allowed by default, any other URL involved in authentication needs to be added to the allowlist.

For example, when using oauth, there is a secondary URL for receiving the auth callback. This has to be in the allowlist, otherwise access will be denied to the callback URL and it will fail.

```
// define a callback url
composer.addWebRoute('get','/oauth/callback', someFunction);

// Add the callback to the allow list.
composer.addAllowedUrl('/oauth/callback');
```

### **plugLoader.loginUri**

This value is used by the built-in authentication middleware to redirect the user to the login page. By default, it is set to `'/login'` but it can be reset by changing this member value.

Note that if you specify an alternate URI for the login page, you must use `addAllowedUrl` to add it to the list of allowed callback URLs.

### **pluginLoader.getUserFromRequest(req)**

This is a static method on the `PluginLoader` class that extracts the user identity information provided by Passport. This is for use in the web route implementations to get user and provide it to other components of Composer.

For example:

```
const RequestHandlerX = async (req, res) => {

  const user = await PluginLoader.getUserFromRequest(req);

  // ... do some stuff

};
```

## **Storage**

By default, Composer reads and writes assets to the local filesystem. Extensions may override this behavior by providing a custom implementation of the `IFileStorage` interface. [See interface definition here](#)

Though this interface is modeled after a filesystem interaction, the implementation of these methods does not require using the filesystem, or a direct implementation of folder and path structure. However, the implementation must respect that structure and respond in the expected ways -- ie, the `glob` method must treat path patterns the same way the filesystem glob would.

### **composer.useStorage(customStorageClass)**

Provide an iFileStorage-compatible class to Composer.

The constructor of the class will receive 2 parameters: a `StorageConnection` configuration, pulled from Composer's global configuration (currently `data.json`), and a user identity object, as provided by any configured authentication extension.

The current behavior of Composer is to instantiate a new instance of the storage accessor class each time it is used. As a result, caution must be taken not to undertake expensive operations each time. For example, if a database connection is required, the connection might be implemented as a static member of the class, inside

the extension's init code and made accessible within the extension module's scope.

The user identity provided by a configured authentication extension can be used for purposes such as:

- provide a personalized view of the content
- gate access to content based on identity
- create an audit log of changes

If an authentication extension is not configured, or the user is not logged in, the user identity will be `undefined`.

The class is expected to be in the form:

```
class CustomStorage implements IFileStorage {  
    constructor(conn: StorageConnection, user?: UserIdentity) {  
        ...  
    }  
  
    ...  
}
```

## Web server

Extensions can add routes and middlewares to the Express instance.

These routes are responsible for providing all necessary dependent assets such as browser JavaScript, CSS, etc.

Custom routes are not rendered inside the front-end React application, and currently have no access to that application. They are independent pages -- though nothing prevents them from making calls to the Composer server APIs.

**composer.addWebRoute(method, url, callbackOrMiddleware, callback)**

This is equivalent to using `app.get()` or `app.post()`. A simple route definition receives 3 parameters - the method, URL and handler callback.

If a route-specific middleware is necessary, it should be specified as the 3rd parameter, making the handler callback the 4th.

Signature for callbacks is `(req, res) => {}`

Signature for middleware is `(req, res, next) => {}`

For example:

```
// simple route  
composer.addWebRoute('get', '/hello', (req, res) => {  
    res.send('HELLO WORLD!');  
});  
  
// route with custom middleware  
composer.addWebRoute('get', '/logout', (req, res, next) => {  
    console.warn('user is logging out!');  
    next();  
},(req, res) => {  
    req.logout();  
    res.redirect('/login');  
});
```

**composer.addWebMiddleware(middleware)**

Bind an additional custom middleware to the web server. Middleware applied this way will be applied to all routes.

Signature for middleware is `(req, res, next) => {}`

For middleware dealing with authentication, extensions must use `useAuthMiddleware()` as otherwise the built-in auth middleware will still be in place.

## User interface

Extensions can host and serve custom UI in the form of a bundled React application at select entries called *contribution points*.

A detailed set of explanation, documentation, and examples are [here](#).

## Publishing

`composer.addPublishMethod(publishMechanism, schema, instructions)`

By default, the publish method will use the name and description from the package.json file. However, you may provide a customized name:

```
composer.addPublishMethod(publishMechanism, schema, instructions, customDisplayName,  
customDisplayDescription);
```

Provide a new mechanism by which a bot project is transferred from Composer to some external service. The mechanisms can use whatever method necessary to process and transmit the bot project to the desired external service, though it must use a standard signature for the methods.

In most cases, the extension itself does NOT include the configuration information required to communicate with the external service. Configuration is provided by the Composer application at invocation time.

Once registered as an available method, users can configure specific target instances of that method on a per-bot basis. For example, a user may install a *Publish to PVA* extension, which implements the necessary protocols for publishing to PVA. Then, in order to actually perform a publish, they would configure an instance of this mechanism, "Publish to HR Bot Production Slot", that includes the necessary configuration information.

Publishing extensions support the following features:

- publish - given a bot project, publish it. Required.
- getStatus - get the status of the most recent publish. Optional.
- getHistory - get a list of historical publish actions. Optional.
- rollback - roll back to a previous publish (as provided by getHistory). Optional.

`publish(config, project, metadata, user)`

This method is responsible for publishing the `project` using the provided `config` using whatever method the extension is implementing - for example, publish to Azure. This method is *required* for all publishing extensions.

In order to publish a project, this method must perform any necessary actions such as:

- The LUIS lbuild process
- Calling the appropriate runtime `buildDeploy` method
- Doing the actual deploy operation

### Parameters:

| PARAMETER            | DESCRIPTION                                                                                              |
|----------------------|----------------------------------------------------------------------------------------------------------|
| <code>config</code>  | an object containing information from the publishing profile, as well as the bot's settings -- see below |
| <code>project</code> | an object representing the bot project                                                                   |

| PARAMETER | DESCRIPTION                                                           |
|-----------|-----------------------------------------------------------------------|
| metadata  | any comment passed by the user during publishing                      |
| user      | a user object if one has been provided by an authentication extension |

Config will include:

```
{
  templatePath: '/path/to/runtime/code',
  fullSettings: {
    // all of the bot's settings from project.settings, but also including sensitive keys managed in-app.
    // this should be used instead of project.settings which may be incomplete
  },
  profileName: 'name of publishing profile',
  ... // All fields from the publishing profile
}
```

The project will include:

```
{
  id: 'bot id',
  dataDir: '/path/to/bot/project',
  files: // A map of files including the name, path and content
  settings: {
    // content of settings/appsettings.json
  }
}
```

Below is an simplified implementation of this process:

```
const publish = async(config, project, metadata, user) => {

  const { fullSettings, profileName } = config;

  // Prepare a copy of the project to build

  // Run the lubuild process

  // Run the runtime.buildDeploy process

  // Now do the final actual deploy somehow...

}
```

`getStatus(config, project, user)`

This method is used to check for the status of the most recent publish of `project` to a given publishing profile defined by the `config` field. This method is *required* for all publishing extensions.

This endpoint uses a subset of HTTP status codes to report the status of the deploy:

| STATUS | MEANING                        |
|--------|--------------------------------|
| 200    | Publish completed successfully |
| 202    | Publish is underway            |

| STATUS | MEANING          |
|--------|------------------|
| 404    | No publish found |
| 500    | Publish failed   |

`config` will be in the form below. `config.profileNames` can be used to identify the publishing profile being queried.

```
{
  profileName: `name of the publishing profile`,
  ... // all fields from the publishing profile
}
```

Should return an object in the form:

```
{
  status: [200|202|404|500],
  result: {
    message: 'Status message to be displayed in publishing UI',
    log: 'any log output from the process so far',
    comment: 'the user specified comment associated with the publish',
    endpointURL: 'URL to running bot for use with Emulator as appropriate',
    id: 'a unique identifier of this published version',
  }
}
```

`getHistory(config, project, user)`

This method is used to request a history of publish actions from a given `project` to a given publishing profile defined by the `config` field. This is an *optional*/feature - publishing extensions may exclude this functionality if it is not supported.

`config` will be in the form below. `config.profileNames` can be used to identify the publishing profile being queried.

```
{
  profileName: `name of the publishing profile`,
  ... // all fields from the publishing profile
}
```

Should return in array containing recent publish actions along with their status and log output.

```
[{
  status: [200|202|404|500],
  result: {
    message: 'Status message to be displayed in publishing UI',
    log: 'any log output from the process so far',
    comment: 'the user specified comment associated with the publish',
    id: 'a unique identifier of this published version',
  }
}]
```

`rollback(config, project, rollbackToVersion, user)`

This method is used to request a rollback *in the deployed environment* to a previously published version. This DOES NOT affect the local version of the project. This is an *optional*/feature - publishing extensions may exclude this functionality if it is not supported.

`config` will be in the form below. `config.profileNames` can be used to identify the publishing profile being queried.

```
{  
  profileName: `name of the publishing profile`,  
  ... // all fields from the publishing profile  
}
```

`rollbackToVersion` will contain a version ID as found in the results from `getHistory`.

Rollback should respond using the same format as `publish` or `getStatus` and should result in a new publishing task:

```
{  
  status: [200|202|404|500],  
  result: {  
    message: 'Status message to be displayed in publishing UI',  
    log: 'any log output from the process so far',  
    comment: 'the user specified comment associated with the publish',  
    endpointURL: 'URL to running bot for use with Emulator as appropriate',  
    id: 'a unique identifier of this published version',  
  }  
}
```

## Runtime templates

`composer.addRuntimeTemplate(templateInfo)`

Expose a runtime template to the Composer UI. Registered templates will become available in the **Runtime settings** tab. When selected, the full content of the `path` will be copied into the project's `runtime` folder. Then, when a user clicks `Start Bot`, the `startCommand` will be executed. The expected result is that a bot application launches and is made available to communicate with the Bot Framework Emulator.

```
await composer.addRuntimeTemplate({  
  key: 'myUniqueKey',  
  name: 'My Runtime',  
  path: __dirname + '/path/to/runtime/template/code',  
  startCommand: 'dotnet run',  
  build: async(runtimePath, project) => {  
    // implement necessary actions that must happen before project can be run  
  },  
  buildDeploy: async(runtimePath, project, settings, publishProfileName) => {  
    // implement necessary actions that must happen before project can be deployed to azure  
  
    return pathToBuildArtifacts;  
  },  
});
```

`build(runtimePath, project)`

Perform any necessary steps required before the runtime can be executed from inside Composer when a user clicks the `Start Bot` button. Note this method *should not* actually start the runtime directly - only perform the build steps.

For example, this would be used to call `dotnet build` in the runtime folder in order to build the application.

`buildDeploy (runtimePath, project, settings, publishProfileName)`

| PARAMETER                | DESCRIPTION                                    |
|--------------------------|------------------------------------------------|
| <code>runtimePath</code> | the path to the runtime that needs to be built |

| PARAMETER          | DESCRIPTION                                                         |
|--------------------|---------------------------------------------------------------------|
| project            | a bot project record                                                |
| settings           | a full set of settings to be used by the built runtime              |
| publishProfileName | the name of the publishing profile that is the target of this build |

Perform any necessary steps required to prepare the runtime code to be deployed. This method should return a path to the build artifacts with the expectation that the publisher can perform a deploy of those artifacts *as is* and have them run successfully. To do this it should:

- Perform any necessary build steps
- Install dependencies
- Write `settings` to the appropriate location and format

#### **composer.getRuntimeByProject(project)**

Returns a reference to the appropriate runtime template based on the project's settings.

```
// load the appropriate runtime config
const runtime = composer.getRuntimeByProject(project);

// run the build step from the runtime, passing in the project as a parameter
await runtime.build(project.dataDir, project);
```

#### **composer.getRuntime(type)**

Get a runtime template by its key.

```
const dotnetRuntime = composer.getRuntime('csharp-azurewebapp');
```

#### **Accessors**

`composer.passport`

`composer.name`

## Extension roadmap

These features are not currently implemented, but are planned for the near future:

- Eventing - extensions will be able to emit events as well as respond to events emitted by other extensions and by Composer core.
- Schema extensions - Extensions will be able to amend or update the component schema and uischema.

## Next

- Learn how to [extend Composer with extensions](#).

# Connections

5/20/2021 • 3 minutes to read

**APPLIES TO:** Composer v2.x

Composer makes it easier than ever to connect your bot to different messaging platforms. Using **Connections**, developers can make it possible for users to interact with their bots on a variety of messaging services.

Developers can connect their bots to [Azure connections](#), including [Web Chat](#), [Direct Line Speech](#), and [Microsoft Teams](#), as well as [external connections](#). Azure connections are created by Microsoft, and external connections are created by community developers. Both offer the ability to connect bots to messaging services, but external connections may be more flexible.

To use Connections, you need:

- A bot in Composer that has been published.
- An Azure account, or Azure resources provisioned by an administrator.

## Access Connections

The following steps show how to access both types of connections in Composer.

1. Open a bot project in Composer.
2. Select the **Configure** button on the left.
3. Select **Connections** on the top.

Overview    Development resources    **Connections**    Skill configuration    Localization

Add connections to make your bot available in Webchat, Direct Line Speech, Microsoft Teams and more. [Learn more](#).

### Azure connections

Connect your bot to Microsoft Teams and WebChat, or enable DirectLine Speech.

Select publishing profile



### External connections

Find and install more external services to your bot project in [package manager](#). For further guidance, see documentation for [adding external connections](#).

| Name                                     | Enabled | Configuration |
|------------------------------------------|---------|---------------|
| <a href="#">Add from package manager</a> |         |               |

See the sections on [Azure connections](#) and [External connections](#) for more information about different types of connections.

# Azure connections

There are three available Azure connections:

- [Web Chat](#) (enabled by default)
- [Direct Line Speech](#)
- [Microsoft Teams](#)

To add Azure connections, do the following:

1. Select a publishing profile. You will be prompted to sign into Azure.
2. Select the Azure connection you want to add, shown in the screenshot below, by toggling it on. Note that Web Chat is enabled by default

**Azure connections**

Connect your bot to Microsoft Teams and WebChat, or enable DirectLine Speech.

myemptybot

| Name     | Enabled                             | Documentation              |
|----------|-------------------------------------|----------------------------|
| MS Teams | <input type="checkbox"/>            | <a href="#">Learn more</a> |
| Web Chat | <input checked="" type="checkbox"/> | <a href="#">Learn more</a> |
| Speech   | <input type="checkbox"/>            | <a href="#">Learn more</a> |

## Web Chat

Web Chat is enabled by default, therefore you don't need to take any steps to connect your bot to Web Chat. See the [enabling a connection to Web Chat](#) page for more information.

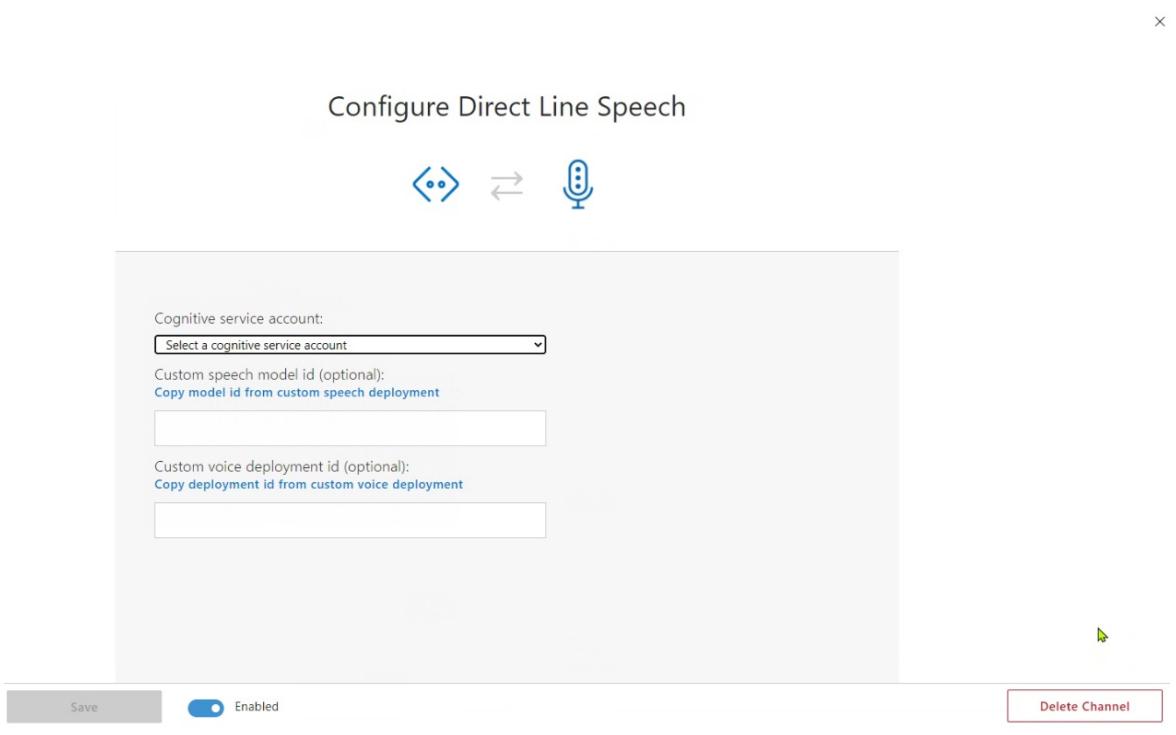
## Direct Line Speech

1. Under **Azure connections**, toggle on **Direct Line Speech**.
2. Select the **Learn more** link for information about [enabling a connection to Direct Line Speech](#). Make sure you have all the prerequisites before attempting to enable a connection to Direct Line Speech.
3. If you want to delete your connection to Direct Line Speech, toggle off Direct Line Speech on the **Connections** page and republish your bot.
4. If you want to configure your Direct Line Speech connection, go to the Azure portal and select **Channels** from the right. You will notice that Direct Line Speech is included in the list of connected channels:

### Connect to channels

| Name                                | Health  | Published |                      |
|-------------------------------------|---------|-----------|----------------------|
| Direct Line Speech                  | Running | --        | <a href="#">Edit</a> |
| Web Chat                            | Running | --        | <a href="#">Edit</a> |
| <a href="#">Get bot embed codes</a> |         |           |                      |

5. Select **Edit**. Here you can configure the connection or delete it, as shown below:



#### NOTE

Your Speech key may be only associated with one Speech app at a time.

## Microsoft Teams

- Under **Azure connections**, toggle on **Microsoft Teams**.
- Select the **Learn more** link for information about [enabling a connection to Microsoft Teams](#). Make sure you have all the prerequisites before attempting to enable a connection to Microsoft Teams. Also select **See instructions** that pops up when you toggle on Microsoft Teams, as shown below, for more information:

#### Azure connections

Connect your bot to Microsoft Teams and WebChat, or enable DirectLine Speech.

| Name     | Enabled                             | Documentation              |
|----------|-------------------------------------|----------------------------|
| MS Teams | <input checked="" type="checkbox"/> | <a href="#">Learn more</a> |
| Web Chat | <input checked="" type="checkbox"/> | <a href="#">Learn more</a> |
| Speech   | <input type="checkbox"/>            | <a href="#">Learn more</a> |

**Almost there!**

Teams requires a few more steps to get your connection up and running. Follow the instructions on our documentation page to learn how.

[See instructions](#)

- Once Microsoft Teams is configured, you will be able to add it to your Teams manifest. Copy the manifest and select **Open Teams** on the bottom.

## Teams Manifest

X

Your Teams adapter is configured for your published bot. Copy the manifest, open App Studio in Teams and add the manifest so you can test your bot in Teams

Teams manifest for your bot:



```
{  
  "$schema": "https://developer.microsoft.com/en-us/json-schemas/teams/v1.9/MicrosoftTeams.schema.json",  
  "manifestVersion": "1.9",  
  "version": "1.0.0",  
  "id": "c00849b3-7450-491e-a421-09e281324db8",  
  "packageName": "myemptybot",  
  "developer": {  
    "name": "contoso",  
    "websiteUrl": "https://contoso.com",  
    "privacyUrl": "https://contoso.com/privacy",  
    "termsOfUseUrl": "https://contoso.com/terms"  
  },  
  "icons": {  
    "color": "",  
    "outline": ""  
  },  
  "name": {  
    "short": "myemptybot",  
    "full": "myemptybot"  
  },  
  "description": {  
    "short": "short description for myemptybot",  
    "full": "full description for myemptybot"  
  },  
  "accentColor": "#FFFFFF",  
}
```

Close

Open Teams

4. If you want to delete your connection to Microsoft Teams, toggle off Microsoft Teams on the **Connections** page and republish your bot.
5. If you want to configure your Microsoft Teams connection, go to the Azure portal and select **Channels** from the right. You will notice that Microsoft Teams is included in the list of connected channels.

### Connect to channels

| Name            | Health  | Published |      |
|-----------------|---------|-----------|------|
| Microsoft Teams | Running | --        | Edit |
| Web Chat        | Running | --        | Edit |

[Get bot embed codes](#)

6. Select **Edit**. Here you can configure the connection or delete it, as shown below:

### Configure Microsoft Teams



[Messaging](#) [Calling](#) [Publish](#)

#### Messaging [Learn more](#)

Messaging is available by default for your bot.

- Microsoft Teams Commercial (most common).  
 Microsoft Teams for Government. [Learn more](#)

Delete channel to change the selection.

Save

Delete Channel

# External connections

To access external connections, complete the following steps:

## 1. Open the Package Manager.

The screenshot shows the 'Package Manager' section of the Microsoft Bot Framework interface. On the left is a sidebar with icons for Home, Create, Configure, User input, Bot responses, Knowledge base, Publish, and Package manager. The 'Create' option is currently selected. At the top right are buttons for 'Add a package' and 'Edit feeds'. Below the sidebar is a search bar with the word 'nuget' typed into it. To the right of the search bar is a 'Search' button. A list of packages is displayed, each with a small icon, the package name, and a brief description. The packages listed are: Microsoft.Bot.Builder.Adapters.Twilio, Microsoft.Bot.Builder.Adapters.Slack, Microsoft.Bot.Builder.Adapters.Facebook, Microsoft.Bot.Builder.Adapters.Webex, and Microsoft.Bot.Builder.AI.Orchestrator. A tooltip on the right side of the screen says 'Select an item from the list to view a detailed description.'

2. Browse or search for your extension.

3. Select the package you want to install and select **Install**.

4. Read the documentation in the package's README to learn about more about it and how to use it in your bot.

Alternatively, you can select **+ Add a package** at the top and enter the exact name and version of the connection's package you want to add.

## Add a package

X

Install a specific package by pasting the name and version number below. The text must be an exact match in order to find the correct package. You can also add and browse feeds to find packages to add.

**Package Name \***

super-dialog-bundle

**Version (optional)**

1.0.0

Add

Cancel

# Authoring Speech experiences

5/20/2021 • 6 minutes to read

**APPLIES TO:** Composer v1.x and v2.x

Bots are able to communicate over speech based channels, including Telephony, enabling a bot to work in contact center/IVR scenarios, and Direct Line Speech, enabling speech experiences in Web Chat, or via embedded devices.

Bots can use text-to-speech (also known as speech synthesis, and referred to as *speech* in this article) to convert text to human-like synthesized speech. Text is converted to a phonemic representation (the individual components of speech sounds) which are converted to waveforms that are output as speech. Composer uses Speech Synthesis Markup Language (SSML), an XML-based markup language that lets developers specify how input text is converted into synthesized speech. SSML gives developers the ability to customize different aspects of speech, like pitch, pronunciation, rate of speech, and more.

This modality lets developers create bots in Composer that can not only respond visually with text, but also audibly with speech. Bot developers and designers can create bots with a variety of voices in different languages with the speech middleware and using SSML tags.

## Add Speech components to your bot responses

It's important to ensure that bot responses are optimized for the channels that they will be available on. For example, a welcome message written in text along with an Adaptive Card attachment will not be suitable when sent via a speech capable channel. For this reason, bot responses can contain both text and speech responses, with the speech response used by the channel when required.

- [Composer v2.x](#)
- [Composer v1.x](#)

Using the **response editor**, bot developers can easily add speech components to bots and customize them with SSML tags.

To add speech to a bot, complete the following steps:

1. Open a bot project and add a **Send a response** action to one of your dialogs. Enter text in the **Text** box for a fallback text response.
2. Now click the **+** next to **Text**. You will see three options: **Speech**, **Attachments**, and **Suggested Actions**. Select **Speech**.

Response Variations [?](#)

This is text

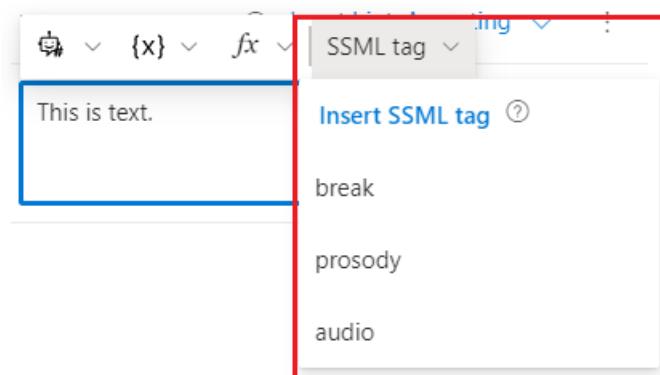
[Add new variation](#)

3. When speech is added you will see **Input hint: accepting** next to **Response variations**. Select **Input hint: accepting** to see all of the available [input hints](#):

- **Accepting:** Indicates that your bot is passively ready for input but is not awaiting a response from the user (this is the default value if no specific InputHint value is set).
- **Ignoring:** Indicates that your bot isn't ready to receive input from the user.
- **Excepting:** Indicates that your bot is actively awaiting a response from the user

For more information, see the Bot Framework SDK article [Add input hints to messages with the Bot Connector API](#).

4. You can add SSML tags to your speech component to customize your speech output. Select **SSML tag** in the command bar to see the SSML tag options.



Composer supports the following SSML tags:

- **break:** Inserts pauses (or `breaks`) between words, or prevent pauses automatically added by the text-to-speech service.
- **prosody:** Specifies changes to pitch, contour, range, rate, duration, and volume for the text-to-speech output.
- **audio:** Allows you to insert MP3 audio into an SSML document.

For more information, see the [Improve synthesis with Speech Synthesis Markup Language \(SSML\)](#) article.

## Voice font and speech related bot settings

In order for speech responses to work correctly on some channels, including Telephony and Direct Line Speech channels, there are some [required SSML tags](#) that must be present.

- **Speak** - required to enable use of SSML tags.
- **Voice** - defines the voice font that will be used when responses are read out by the Telephony or Direct Line

Speech channels.

**TIP**

Visit the [language and voice support for the Speech Service documentation](#) to see a list of supported voice fonts. It's recommended that you use neural voice fonts, where available, as these sound particularly human-like.

Composer makes it as easy as possible for bot builders to develop speech applications, automatically including these SSML tags on all outgoing responses, with the ability to modify related properties in the Composer runtime settings.

To access the speech related settings, complete the following steps:

1. Open a Composer bot project and select the **Project Settings** on the **Navigation** pane on the left
2. Select **Advanced Settings View (json)** to show the JSON view of the project settings. There are two relevant speech sections, shown below.
  - [Composer v2.x](#)
  - [Composer v1.x](#)

 Advanced Settings View (json)



```
57     "path": "...",
58 },
59   "runtimeSettings": {
60     "adapters": [],
61     "features": {
62       "removeRecipientMentions": false,
63       "showTyping": false,
64       "traceTranscript": false,
65       "useInspection": false,
66       "setSpeak": {
67         "voiceFontName": "en-US-AriaNeural",
68         "fallbackToTextForSpeechIfEmpty": true
69       }
70     },
71     "components": [
72       {
73         "name": "Speech"
74       }
75     ]
76   }
77 }
```

- `"voiceFontName": "en-US-AriaNeural"` : Determines the `voiceFontName` your bot will use to speak, and the default is `en-US-AriaNeural`. You can customize this using any of the available [voices and locales](#) appropriate for your bot.
- `"fallbackToTextForSpeechIfEmpty": true` : Determines whether text will be used if speech is empty, and the default is true. If you don't add SSML tags to your speech, there will be silence and instead the text will be displayed as a fallback message. To turn this off, set this to `false`.

**NOTE**

If you need to disable the speak / voice SSML tags being applied to all responses, you can do this by removing the `setSpeak` element from your bot settings completely. This will disable the related middleware within the runtime.

## Connect to channels

Speech is supported by the Telephony and Direct Line Speech channels within Azure Bot Service. For information about connecting a bot to channels that support voice, see the appropriate document below:

- Connect to [Telephony](#)
- Connect to [Direct Line Speech](#)

## Test speech

To test speech capabilities in your bot, connect your bot to one of the aforementioned [channels](#) and use the channel's native communication method to test your bot. For example, for the Telephony channel you would call the bot's configured phone number.

To inspect the responses being sent by your bot, including speech specific responses containing the automatically added SSML tags, plus any that you have manually added, do the following.

1. Go to a bot project and [create a few activities that have text and speech](#).
2. Start your bot and test it in the Emulator. You will see SSML elements appear in the inspect activity JSON.

# Orchestrator

6/11/2021 • 8 minutes to read

**APPLIES TO:** Composer v2.x

Developers can craft many diverse conversational experiences in the Bot Framework but unify the onboarding to these experiences via one entry point for end-users; this composition model is often referred to as skills.

Within an Enterprise, creating one parent bot bringing together multiple child bots (skills) owned by different teams, or more broadly leveraging common capabilities provided by other developers is an effective design pattern for horizontally scaling automation across many different concerns in the organization.

There are two component model approaches available today within Bot Framework, [Bot Components](#) and [skills](#).

- Bot Components enable you to share any combination of declarative conversational assets (dialogs, language models and language generation responses) and code with other developers either in source-code form or through a package and associated package feed. This is then imported into your project and can be modified as needed to suit your scenario. This is directly analogous to a shared code library.
- Skills enable a Bot to be surfaced to other conversational experiences sharing it with other bots who then pass utterances across for remote processing. This is directly analogous to how you might construct a lattice of web services.

It's important to select the most appropriate mechanism for your scenario. Bot components provide a simple way to share common capabilities across projects which can then be modified and skills provide central ownership of a given experience which is then used as-is by other conversational experiences.

Any bot can be leveraged as a skill, additional configuration is needed including creation of a skill manifest that Composer helps automate.

For skill scenarios, the parent bot is responsible for dispatching utterances from a user to the skill best suited to process the utterances. This dispatcher needs to be trained with example utterances from each skill to build a dispatching model enabling the parent bot to identify the right skill and route this and subsequent utterances to it.

Orchestrator is a replacement for the now deprecated [Bot Framework Dispatcher](#) and provides a straightforward to use, robust skill dispatching solution for Bot Framework customers. Bots built using Composer or ones created directly using the SDK can use it, enabling existing Dispatch users to switch to Orchestrator without friction.

Orchestrator utilizes state-of-the-art natural language understanding methods while at the same time simplifying the process of language modeling. Using it does not require expertise in Deep Neural Networks (DNN), [Transformers](#), or [Natural Language Processing \(NLP\)](#). This work is co-authored with the industry experts in the field and includes some of the top methods used in the [General Language Understanding Evaluation \(GLUE\)](#) leaderboard. Orchestrator will continue to evolve and adapt to the latest advancements in science and the industry.

## Using Orchestrator

Orchestrator can easily be added to any Composer based bot through the packager manager capability by searching for the Orchestrator package via the appropriate package repository [[NuGet](#)] or [[npm](#)]. Once the package is installed, Orchestrator automatically manages the download of the latest model ready for use by your bot.

## Package Manager

Discover and use components that can be installed into your bot. [Learn more](#)

The screenshot shows the Microsoft NuGet Package Manager interface. At the top, there are tabs for 'Browse' and 'Installed'. Below the tabs is a search bar with the word 'nuget' typed into it. To the right of the search bar is a 'Search' button. On the far right, there is a Microsoft logo and a blue button labeled 'Install 4.13.2 | v'. The main area displays a list of packages. The first five packages listed are: 'Microsoft.Bot.Builder.Adapters.Twilio' (Library for connecting bots with Twilio SMS API), 'Microsoft.Bot.Builder.Adapters.Slack' (Library for connecting bots with Slack), 'Microsoft.Bot.Builder.Adapters.Facebook' (Library for connecting bots with Facebook), 'Microsoft.Bot.Builder.Adapters.Webex' (Library for connecting bots with Webex Teams API), and 'Microsoft.Bot.Builder.AI.Orchestrator' (This library implements .NET support for Orchestrator). A 'View documentation' link is also visible next to the Orchestrator package.

Once installed, you can select the Orchestrator recognizer within your root bot's recognizer configuration.

The screenshot shows the 'Adaptive dialog' configuration page. The title 'Adaptive dialog' is at the top. Below it is a section titled 'Adaptive dialog' with a description: 'This configures a data driven dialog via a collection of events and actions.' A 'Learn more' link is present. The main configuration area is titled 'Language Understanding' with a help icon. Below it is a 'Recognizer Type' dropdown menu. The menu items are: 'Orchestrator recognizer' (selected), 'Default recognizer', 'Regular expression recognizer', 'Custom recognizer', and 'Orchestrator recognizer' again. At the bottom of the dropdown menu, there are two buttons: 'done' and 'dialog.result'.

Alternatively, when connecting a Skill for the first time a developer, the step of adding Orchestrator can be automated as part of the skill connection flow.

## Enable Orchestrator

X

A bot that consists of multiple bots or connects to skills (multi-bot project) needs Orchestrator to detect and route user input to the appropriate bot or skill.

A bot that consists of multiple bots or connects to skills (multi-bot project) needs Orchestrator to detect and route user input to the appropriate bot or skill.

[Learn more about Orchestrator](#)

Use Orchestrator for multi-bot projects (bots that consist of multiple bots or connect to skills).

[Back](#)

[Skip](#)

[Continue](#)

Alternatively, you can add Orchestrator to a SDK-first bot following [instructions on GitHub](#). A sample SDK-first bot is also available [here](#).

## Orchestrator benefits

### No ML or NLP expertise required

Our previous Dispatch based approach required significant expertise and time to train a robust dispatch model especially when many diverse skills were connected to a parent bot. For example, the chatbot author would be concerned with proper data distributions, data imbalance, feature-level concerns including the generation of various synonym lists, and more. These aspects, if not addressed, could lead to poor skill Dispatch results.

With Orchestrator, our goals are to ensure these aspects and related expertise are not needed to create a robust dispatching model.

### State-of-the-art classification with few training examples

Developers often face an issue of very few training examples available to properly define the language model. With the powerful pre-trained state-of-the-art models used by Orchestrator, this is less of a concern. Just one example for an intent or skill can often go a long way in making quite accurate predictions. For example, a "Greeting" intent defined with just one example, "hello", can be successfully predicted for examples like "how are you today" or "good morning to you".

The power of the pre-trained models and their generalization capabilities using a very few simple and short examples is powerful. This ability is often called a *few-shot learning* including [one-shot learning](#) that Orchestrator also supports. This ability is made possible thanks to the pre-trained models that were trained on large data sets and then optimized for conversation also on large data.

### Multi-lingual

Orchestrator provides a multi-lingual model alongside English which provides the ability for a model trained with for example English-only data to process utterances in other languages.

In this example, using the CLI for ease of demonstration, we pull down the multi-lingual model rather than the default English model. You can retrieve a list of available models through the command

```
bf orchestrator:basemodel:list .
```

```
bf orchestrator:basemodel:get --versionId=pretrained.20210205.microsoft.dte.00.06.unicoder_multilingual.onnx  
--out=model
```

Then create a snapshot using a .lu file with solely English utterances.

```
bf orchestrator:create --in test.lu --model model --out generated
```

And then test using a German utterance ("book a meeting with Darren") which correctly classifies the intent as the `BookMeeting` intent.

```
bf orchestrator:query -i="generated\test.blu" -m=model -q="Buchen Sie einen Termin mit Darren"
```

```
[  
 {  
   "label": {  
     "name": "BookMeeting",  
     "label_type": 1,  
     "span": {  
       "offset": 0,  
       "length": 34  
     }  
   },  
   "score": 0.24771601762059242,  
   "closest_text": "book a meeting with darren"  
 }  
]
```

## Multi-intents

Orchestrator also supports multi-intent detection, whereby if an utterance for a user contains two instructions (for example `book a meeting with Darren and add a task to pickup milk`) these can both be identified and provided to the bot for subsequent processing.

The example below, using the CLI for ease of demonstration, shows two intents being extracted from a given utterance.

```
bf orchestrator:query -i="generated\test.blu" -m=model -q="book a meeting with darren and add a task to pickup milk"
```

```
[  
 {  
   "closest_text": "add task to pickup milk",  
   "score": 0.7430192247281288,  
   "label": {  
     "name": "AddTask",  
     "label_type": 1,  
     "span": {  
       "length": 56,  
       "offset": 0  
     }  
   }  
 },  
 {  
   "closest_text": "book a meeting with darren",  
   "score": 0.6492044311828292,  
   "label": {  
     "name": "BookMeeting",  
     "label_type": 1,  
     "span": {  
       "length": 56,  
       "offset": 0  
     }  
   }  
 }  
 ]
```

## Ability to classify the unknown intent without additional examples

Another common challenge that developers face in handling intent classification decisions is determining whether the top scoring intent should be triggered or not. Orchestrator provides a solution for this. Its scores can be interpreted as probabilities calibrated in such way that the score of 0.5 is defined as the maximum score for an *unknown* intent selected in a way to balance the precision and recall. If the top intent's score is 0.5 or lower the query or request should be considered an unknown intent and should probably trigger a follow-up question by the bot. On the other hand, if the score of two intents is above 0.5 then both intents (skills) could be triggered. If the bot is designed to handle only one intent at a time, then the application rules or other priorities could pick the one that gets triggered in this case.

The classification of the unknown intent is done without the need for any examples that define the unknown (often referred to as ["zero-shot learning"] [10]) which would be challenging to accomplish. It would be hard to accomplish this without the heavily pre-trained language model, especially since the bot application may be extended in the future with additional skills that were unknown so far.

## Local, fast library

The Orchestrator core is written in C++ and is currently available as a library in C# and Node.js. The library can be used directly by the bot code (a preferred approach) or can be hosted out-of-proc or on a remote server. Running locally eliminates additional service round-trip latency. This is especially helpful when using Orchestrator to dispatch across disparate LU and QnA services.

As an example, the English pre-trained language model (`pretrained.20200924.microsoft.dte.00.06.en.onnx`) is roughly 260 MB. Classification of a new example with this initial model takes about 10 milliseconds (depending on the text length). These numbers are for illustration only to give a sense of performance. As we improve the models or include additional languages, these numbers will likely change.

## Reports

Orchestrator also provides a test mechanism for evaluating the performance of an Orchestrator model, which in turn generates a report.

In order to achieve high quality natural language processing (like intent detection), it's necessary to assess and refine the quality of the model. This is much simplified in Orchestrator because of its use of pre-trained models. The optimization cycle is required in order to account for human language variations.

More information on how to generate and analyze the report can be found [on GitHub](#).

## Minimal or no model training required

Building a language model requires multiple iterations of adding or removing training examples followed by training the model and evaluation which can take significant effort depending on the complexity. Also, when using a [transformer](#) model for classification tasks, a classification layer (or layers) are added and trained making this process expensive, time consuming and often requiring GPU.

To address these concerns, we chose an example-based approach where the language model is defined as a set of labeled examples. In Orchestrator a model example is represented as a vector of numbers (an embedding) obtained from the transformer model for a given text that the corresponding skills is capable of handling.

During runtime a similarity of the new example is calculated comparing it to the existing model examples per skill. The weighted average of  $K$  closest examples ([KNN algorithm](#)) is taken to determine the classification result. This approach does not require an explicit training step, only calculation of the embeddings for the model examples is done. The operation is performed locally without GPU and without remote server roundtrips.

## Additional Information

- See the [school navigator sample](#) for an example of Orchestrator in Composer.

# Understanding packages

5/20/2021 • 2 minutes to read

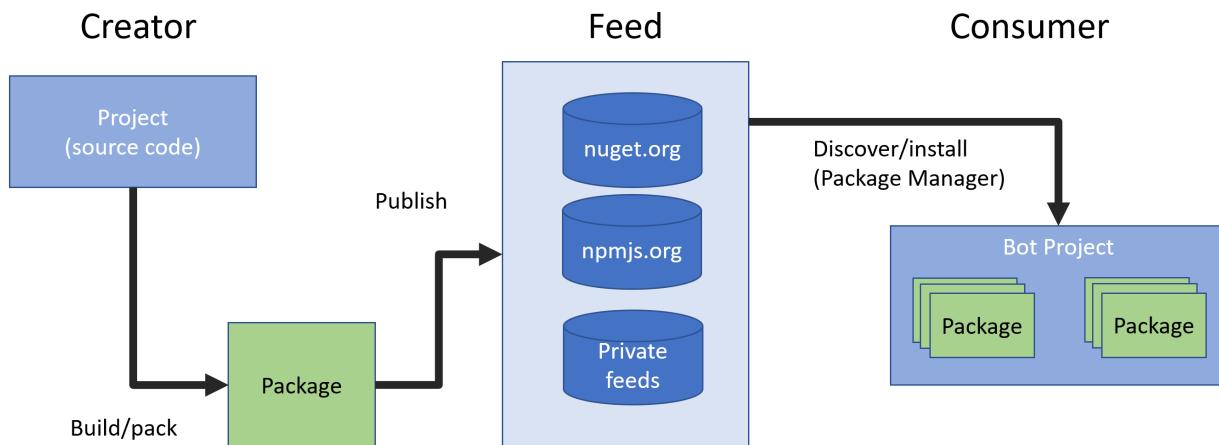
APPLIES TO: Composer v2.x

Packages provide a mechanism for bot developers to create, share, and consume useful bundles of bot functionality like custom actions and triggers, dialogs, and language understand and generation files. They are the preferred method for adding (or sharing) context-specific features to a bot - for example when creating bots for Microsoft Teams there is a package to add custom actions and triggers for working with Teams-specific concepts like messaging extensions and task modules.

The packages themselves are compressed (zipped) folders described by a manifest file with information about the package like the publisher, version, and any dependencies the package has. Package Manager in Composer will help discover, install, and manage packages for bot projects.

## The package lifecycle

Composer bots support working with two types of packages - [NuGet](#) for C#/dotnet bots and [npm](#) for Node/JavaScript bots. Regardless of type, package *creators* package up the functionality they want to share, then publish them to a *package feed*. Package feeds can be public, like the [Nuget feed](#), private like those offered through a service like [MyGet](#), or simply a folder on a local drive or server share. *Consumers* then use a tool like Package Manager in Composer to discover packages, and install them into their bot projects.



## Packages in bot projects

Packages for Composer bots require some additional steps beyond those typically required for npm and NuGet packages in order to work seamlessly with a bot project. Package Manager will perform the necessary additional steps when working with packages, but it is important to understand the results of those steps.

Declarative files (.dialog, .lu, .lg, and .qna files) in a package will be copied into an *exported* folder at the root of the bot project. The bot will then reference the copied files rather than the versions in the package itself. This is to support editing of those files directly - typically a bot developer will need to customize them to meet their exact scenario.

Schema files (.schema and .uischema files) in a package will be merged into the schema files for the bot project, so that Composer can consume them appropriately.

Finally, references will be added to the *components* array in the *appsettings.json* file for any components (coded

extensions like custom actions, middleware, adapters etc.) in the package - this allows the runtime to inject the component at initialization time appropriately.

For additional technical details (like how to perform these additional steps from outside of Composer), see the articles in the [Next steps](#) section.

## Next steps

- [Managing packages with Package Manager](#)
- [Create your own packages](#)

# Hand off to human or virtual agent

5/20/2021 • 3 minutes to read

APPLIES TO: Composer v2.x

Even though a bot can handle many different use cases in a fully automated way, there may still be times when it needs to hand off the conversation to another agent (often a human being). In such cases the bot should recognize when it needs to hand off and provide the user with a smooth transition.

There are two elements that need to be considered as part of implementing handoff within a Composer bot.

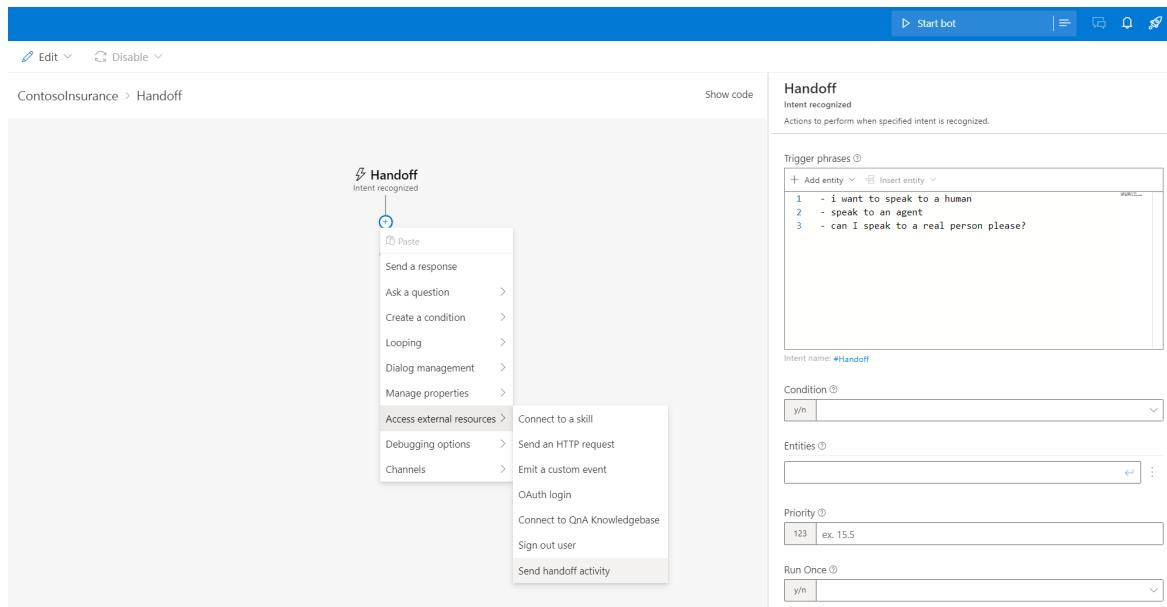
- Triggering the handoff at the appropriate point within a conversation
- Adding the appropriate package which handles the escalation and communication with the chosen platform

## Triggering handoff

The appropriate time to trigger handoff to an agent within your project can occur at many points in a given conversation. For example, this could be in response to a user explicitly asking to speak to a human.

To trigger handoff, complete the following steps:

1. Add a **Send a handoff request** action to one of your dialogs or triggers.



2. When adding a **Send a handoff request** action, you can provide Context information to pass to the target handoff platform, relaying information about the user or conversation to the agent being handed off to. To populate this Contextual information, refer to the instructions for the handoff package you are using.

If successfully triggered, all messages from the user will be routed to the target platform (with responses from the target platform subsequently being forwarded to the user), until target platform indicates that handoff has completed. After the handoff is completed, all messages from the user will once again be sent to, and handled by, the bot.

## Installing a handoff package

You need to install an appropriate handoff package for the platform that you wish to have conversations handed

off to when you use a **Send a handoff request** activity.

1. Select the **Package Manager** on the **Navigation** pane on the left.
2. Search for **handoff** using the search field to find a list of available packages.
3. Select the package for the platform you wish to handoff to and click the **Install** button in the right hand pane to install the package.
4. Follow the **View documentation** link in the right hand pane for the package to complete the specific configuration steps for the package.

## Available handoff packages

The following hanoff packages, from the [Bot Framework Community](#) are currently available to be used within Composer.

### ServiceNow

This package provides integration with the ServiceNow platform, enabling handoff of a conversation to the ServiceNow Virtual Agent or if preferred a human agent using the ServiceNow Bot-to-Bot integration API.

Welcome to the Enterprise Bot Example which has the new Orchestrator integrated and demonstrates handoff of ServiceNow related questions to the ServiceNow Virtual Agent

 Type your message 



It can be found within the **Package Manager** with the name **Bot.Builder.Community.Components.Handoff.ServiceNow**.

In order to use this package, a ServiceNow account with sufficient permissions to login to create / manage apps within your ServiceNow instance. If you do not have this you can create a trial account for free at <https://developer.servicenow.com/>.

When using the **Send a handoff activity** action, to trigger handoff, with the ServiceNow package, you can optionally provide a JSON object within the **Context** property input box in the right hand pane. Currently you can pass a timezone, user Id and / or an email address, using the following format. All properties within this JSON are optional.

```
{  
  "timeZone": "Europe",  
  "userId": "user@domain.com",  
  "emailId": "myuserId"  
}
```

## LivePerson

This package provides integration with the LivePerson platform, enabling handoff to the [LivePerson Conversational Cloud](#). How a conversation is handled/routed when it is handed off to LivePerson is defined by the [routing rules](#) configured within the LivePerson platform.

The package can be found within the [Package Manager](#) with the name `Bot.Builder.Community.Components.Handoff.LivePerson`.

In order to use this package, A LivePerson account with sufficient permissions to login to create / manage apps at <https://connector-api.dev.liveperson.net/>. If you do not have this you can [create a trial account for free](#).

When using the **Send a handoff activity** action, to trigger handoff, with the LivePerson package, you can optionally provide a JSON object within the **Context** property input box in the right hand pane. Currently you can pass a **Skill** name, which can be used as part of the routing rules within LivePerson, as well as customer information Engagement Attributes, which are made available to virtual and human agents within LivePerson.

```
{  
  "Skill": "Credit Cards",  
  "EngagementAttributes": [  
    {  
      "Type": "ctmrinfo",  
      "CustomerType": "vip",  
      "SocialId": "123456789"  
    }  
  ]  
}
```

# Extending your bot with code

5/20/2021 • 2 minutes to read

**APPLIES TO:** Composer v2.x

Composer provides a powerful platform for creating complex conversation flows, and comes with a robust set of pre-built triggers and actions to use. However, sometimes there will be a need to perform an action, respond to a trigger, work with an external API or do something for which there is no pre-built functionality. Or there may be a need to do something more complex like author middleware, create a custom storage provider, or connect to a client using a custom adapter. To accomplish these tasks (and more) create coded extensions called **bot components**.

Bot creation in Composer creates the files necessary files to build and run your bot, including a `<mybot>.sln` or `index.js` file you can use to open your bot in your favorite code IDE. The bot project has a dependency on the **adaptive runtime** package, which provides access to the breadth of the Bot Framework SDK and extension points for registering components.

## Adaptive runtime

The adaptive runtime is responsible for registering and injecting components. The `BotComponent` interface in the adaptive runtime describes a single method that accepts Service Collection and Configuration instances. The Service Collection provides components access to the common set of Bot Framework things that comprise the runtime. The Configuration instance provides access to a scoped set of user-provided configurations (like connection strings or secrets).

This creates a flexible and safe method to extend a bot with code. Through the runtime, components have access to the power of the Bot Framework SDK, while ensuring that any created component can be easily [packaged](#) and reused across multiple bots.

## Bot components

A **component** is simply a group of coded extensions for a bot that uses the `BotComponent` class to register itself with the adaptive runtime. Components can contain things like:

- Actions
- Triggers
- Middleware
- Adapters
- Storage providers
- Authentication providers
- Controllers/routes

Typically, a bot component consists of these parts:

- The coded extension, like custom actions or adapters.
- An implementation of the `BotComponent` class to register the objects and services of the component at start time.
- One or more `.schema` files that are used to register services in the component with the adaptive dialog stack.
- A `.uischema` file to tell Composer where and how to show the component in application UI.

## Next

- [Create custom actions](#)
- [Create custom triggers](#)

# Enterprise Assistant Bot Template

6/11/2021 • 5 minutes to read

APPLIES TO: Composer v2.x

## Overview

Many organizations are looking to provide a centralized conversational experience across canvases for their employees. The Enterprise Assistant template in [Bot Framework Composer](#) provides a starting point for those interested in creating a virtual assistant for common enterprise scenarios. This template demonstrates a common bot building architecture and high-quality pre-built conversational experiences through a root bot connected to multiple skills. This pattern allows for the consolidation of multiple bots across the organization into a centralized solution where a root bot finds the correct bot to handle a query, accelerating user productivity. With Composer, you have the flexibility to personalize the assistant to reflect the values, brand, and tone of your company.

The template is designed to be ready out-of-the-box with support for common employee channels such as Microsoft Teams and Web Chat. It includes features from [Core Assistant Bot Template](#) and two pre-built skills ([Enterprise Calendar Bot](#) and [Enterprise People Bot](#)), with more to come.

## Conversational experience

### Design principles

This template aims to deliver an experience that lets bots communicate like humans, rather than teaching humans how to talk to bots. It leverages principles of natural conversation to let users accomplish common enterprise tasks, such as managing calendars and searching for colleagues. Employees in your organization will save time and have a more accurate understanding of their schedules and the people in your organization.

The end-to-end interactions with the Enterprise Assistant Bot are based on the following principles:

- **UX Pattern:** the chat-optimized flows follow a consistent pattern of displaying visual information using [Adaptive Cards](#) (i.e., from complex elements in a list to a detailed view card). When appropriate, follow up actions are included to promote smooth transitions between relevant tasks.
- **Natural conversation:** every reply from the assistant (excluding card responses) has at least 3 variant [Language Generation](#) responses.
- **Hero channels:** the template is optimized for a great experience in [Microsoft Teams](#) and [Web Chat](#). It is also designed so that support for [additional channels](#) can be added.
- **Chit-chat style:** professional chit-chat is included in the template, can be replaced with other out-of-the-box personality offerings. Learn more about [Personality Chat](#).

Learn more about these design principles in the [Conversational User Experience Guide](#)

### Scenarios

Here is an overview of the scenarios included in the Enterprise Assistant Bot template:



# Enterprise Assistant Bot

## Bot

### Core Assistant

- Greeting new and returning users
- Bot tour
- Asking for help
- Cancelling a dialog
- Submitting feedback
- Error handling
- Repeat

## Bot

### Enterprise Calendar

- Overview of the day
- Find event(s)
- Create event
- Update event
- Respond to event
- First event of the day
- Last event of the day
- Breaks

## Bot

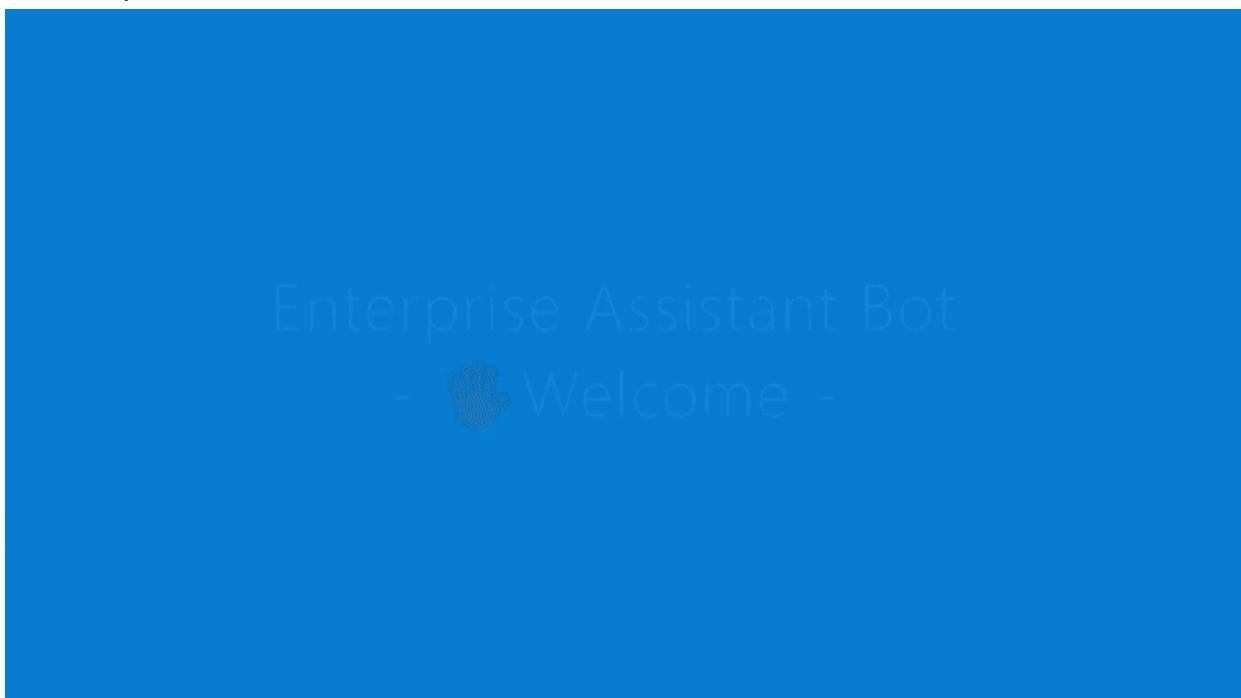
### Enterprise People

- Contact details of a colleague
- Their manager
- Works with
- Peers of
- Collaborators
- Directly call/email/message

## Example interactions

Find example interactions from the Enterprise Assistant Bot and connected skills below:

### Welcome experience



### Switching between skills

Enterprise Assistant Bot  
- Context switching -

**Calling the Enterprise Calendar Bot to view and manage events**

Enterprise Assistant Bot  
- Calendar -

**Calling the Enterprise People Bot to answer questions about the people in an organization**

# Enterprise Assistant Bot

## - People -

## Developer experience

The Enterprise Assistant Bot builds upon the features in the [Core Assistant Bot Template](#) to include the following:

- Orchestrator recognition
- Connected skills

### Orchestrator

The Enterprise Assistant Bot uses Orchestrator as the primary tool for language understanding to route incoming activities to the right intent handler or skill. Learn more about [Orchestrator](#).

### Connected skills

The template includes two connected skills out-of-the-box, the Enterprise Calendar Bot and Enterprise People Bot. These skills are hosted separately to demonstrate a pattern that is common in enterprise scenarios. Learn more about [Skills](#).

Refer to the following documentation for more information about the pre-built skills:

- [Enterprise Calendar Bot Template](#)
- [Enterprise People Bot Template](#)

## Data usage and storage

The Enterprise Assistant project includes two enterprise skills, Calendar and People, which require access to users' [Microsoft Graph](#) data. The following sections outline what data is accessed, where it is stored, and the default data retention policy for each service. We recommend reviewing this information and adjusting the default configuration of your services to meet your organization's compliance standards.

### Storage locations

This section outlines the different data stores used in a typical bot lifecycle, their purpose, and our recommendations for configuring them.

#### Azure Cosmos DB (Required)

[Cosmos DB](#) is the default and recommended provider for storing Bot State. Bot State is split into 4 different scopes, each with a different default lifetime. The default scopes are as follows:

- **Conversation State:** For storing values that should persist over the lifetime of a conversation with a user on a given channel. In a channel with persistent conversations such as Microsoft Teams or Facebook Messenger, this could be stored indefinitely unless otherwise configured.
- **User State:** For storing values that should persist over the lifetime of a user on a given channel. In a channel with persistent conversations such as Microsoft Teams or Facebook Messenger, this could be stored indefinitely unless otherwise configured.
- **Dialog State:** For storing values that should persist for the duration of a dialog. If a dialog is left in a waiting state in a channel with persistent conversations such as Microsoft Teams or Facebook Messenger, this could be stored indefinitely unless otherwise configured.
- **Turn State:** For storing values that persist only for the current turn. A turn indicates the time between the user's response. Every time a new activity from a user is received, Turn State is reset. Turn State properties are never written to a persistent storage location.

For more information about the data accessed and stored in each skill, refer to the following documentation:

- [Enterprise Calendar Bot Template](#)
- [Enterprise People Bot Template](#)

Due to the potentially sensitive data stored by the skills in bot state, we recommend the following configuration for your Cosmos DB resource:

- Configure a default Time to Live (TTL) for your Cosmos DB container. This value should be adjusted according to your organization's standards. [Learn more](#).

#### **Application Insights (Recommended)**

If you have [Application Insights](#) provisioned and configured in your Project Settings, each incoming and outgoing activity will be logged including varying properties depending on your settings. You can configure your telemetry settings in your Project Settings in Composer by toggling the following properties:

```
"runtimeSettings": {
  "telemetry": {
    "logActivities": true,
    "logPersonalInformation": false
  }
}
```

The `logActivities` property controls whether activity will be logged in Application Insights, and the `logPersonalInformation` property controls whether the logged activities will include the `Activity.Text`, `Activity.Speak`, and `Activity.From.Name` properties.

By default, any telemetry stored in Application Insights will follow the default retention duration. You may want to change this behavior depending on your organization's standards. [Learn more](#).

#### **Azure Blob Storage (Optional)**

If you have transcript storage configured in your Project Settings each incoming and outgoing activity will be logged in Azure Blob Storage. This is useful for analysis of your bot's conversations but should be evaluated against your organization's compliance standards. You can configure transcript storage in your Project Settings in Composer by toggling the following property:

```
"runtimeSettings": {
  "features": {
    "traceTranscript": false
  }
}
```

## Next steps

To set up the assistant created using Enterprise Assistant Bot template in Bot Framework Composer, follow the steps in [Enterprise Assistant Bot Tutorial](#).

[Submit a feature request](#) with your detailed enterprise scenario for future consideration.

# Tutorial: Set up your Enterprise Assistant Bot

5/20/2021 • 2 minutes to read

APPLIES TO: Composer v2.x

## Create your project in Composer

The Enterprise Assistant Bot template is included in the Bot Framework Composer by default. Follow these steps to create a project:

1. Open Bot Framework Composer (version 2.0.0 or higher)
2. Select **Create new**
3. Select **Enterprise Assistant Bot** from the list
4. Enter your desired **Name** and **Location**
5. Select Azure Web App for **Runtime type**
6. Select **Create**

### NOTE

This template does not support the Azure Functions runtime due to its project dependencies.

## Provision Azure resources

The Enterprise Assistant Bot requires Azure resources in order to run locally. Follow these steps to provision the required resources:

1. Create a publishing profile and provision resources according to the steps in [Publish a bot to Azure](#). The following resources are required for local development:
  - Microsoft Application Registration
  - Azure Hosting
  - Microsoft Bot Channels Registration
  - Microsoft Language Understanding Authoring Account
  - Microsoft QnA Maker
2. Fill in the following settings in **Configure > Development Resources** with the keys you provisioned:
  - Language Understanding authoring key
  - QnA Maker Subscription key
  - Microsoft App Id
  - Microsoft App Password

## Provision skills and configure authentication

Complete the setup guides for each of your skills:

- [Enterprise Calendar Bot](#)
- [Enterprise People Bot](#)

# Configure allowed callers

For each of your bots, follow these steps to register allowed callers:

1. Open **Configure > Skill configuration**
2. Add the App ID(s) that should be allowed to call your bot. For your root bot, this should be the IDs of the skills. For the skill bots, this should be the ID of the root bot.

## Run and test your bot locally

1. Running your bot in Composer will automatically create add the following configuration in your root bot's settings, visible in the **Advanced Settings View**, enabling to your communicate with your local skills:

```
"skill": {  
  "Calendar": {  
    "endpointUrl": "<local skill messaging endpoint>",  
    "msAppId": "<skill MS App ID>"  
  },  
  "People": {  
    "endpointUrl": "<local skill messaging endpoint>",  
    "msAppId": "<skill MS App ID>"  
  }  
}
```

## Next steps

After you have completed the preceding steps to run and test locally, you can follow these steps to publish your projects to Azure:

### Create skill manifests & publish skills

Once your resources are provisioned and your authentication settings have been configured, follow these steps to create a skill manifest and publish each of your skills:

1. In the **Create** tab, select the **More Options** button beside your skill project.
2. Select **Export as skill** to launch the manifest creation flow
3. Fill in the properties, dialogs, and triggers you would like to include. Learn more about [skill manifests](#).
4. Fill in the allowed callers field with the ID of your root bot
5. Select the publishing profile to use for your skill endpoints
6. Finally, select **Generate and Publish** to publish your skill to your endpoint with the generated manifest.

### Publish to Azure

To publish your Enterprise Assistant Bot, follow these steps:

1. Create a publishing profile and provision resources according to the steps in [Publish a bot to Azure](#). The following resources are required for production environments:
  - Microsoft Application Registration
  - Azure Hosting
  - Microsoft Bot Channels Registration
  - Azure Cosmos DB
  - Application Insights

- Microsoft Language Understanding Authoring Account
  - Microsoft Language Understanding Prediction Account
  - Microsoft QnA Maker
2. In **Configure > Skill configuration > Call skills**, update your **Skill host endpoint URL** to your production bot skills endpoint (i.e. "http://<bot-name>.azurewebsites.net/api/skills")
3. Update your skill endpoints to your published endpoints in **Configure > Advanced Settings View**:

```
"skill": {  
    "Calendar": {  
        "endpointUrl": "<published skill messaging endpoint>",  
        "msAppId": "<skill MS App ID>"  
    },  
    "People": {  
        "endpointUrl": "<published skill messaging endpoint>",  
        "msAppId": "<skill MS App ID>"  
    }  
}
```

4. Publish your bot via the **Publish** tab

# Enterprise Calendar Bot Template

5/20/2021 • 9 minutes to read

APPLIES TO: Composer v2.x

## Overview

The Enterprise Calendar Bot helps to manage user's schedule by displaying, creating, and rescheduling calendar events.

## Conversational experience

The following are the applicable core scenarios:

### Enterprise Calendar Bot

- Overview of the day
- Find event(s)
- Create event
- Update event
- Respond to event
- First event of the day
- Last event of the day
- Breaks

### Showcase

- "Set up a meeting with Anna for tomorrow at 9 AM"
- "When do I have breaks today"
- "What is scheduled for tomorrow", and much more!

[Learn more](#) about the design principles used in the Enterprise Assistant Bot which includes the Enterprise Calendar Bot as a skill.

## Developer experience

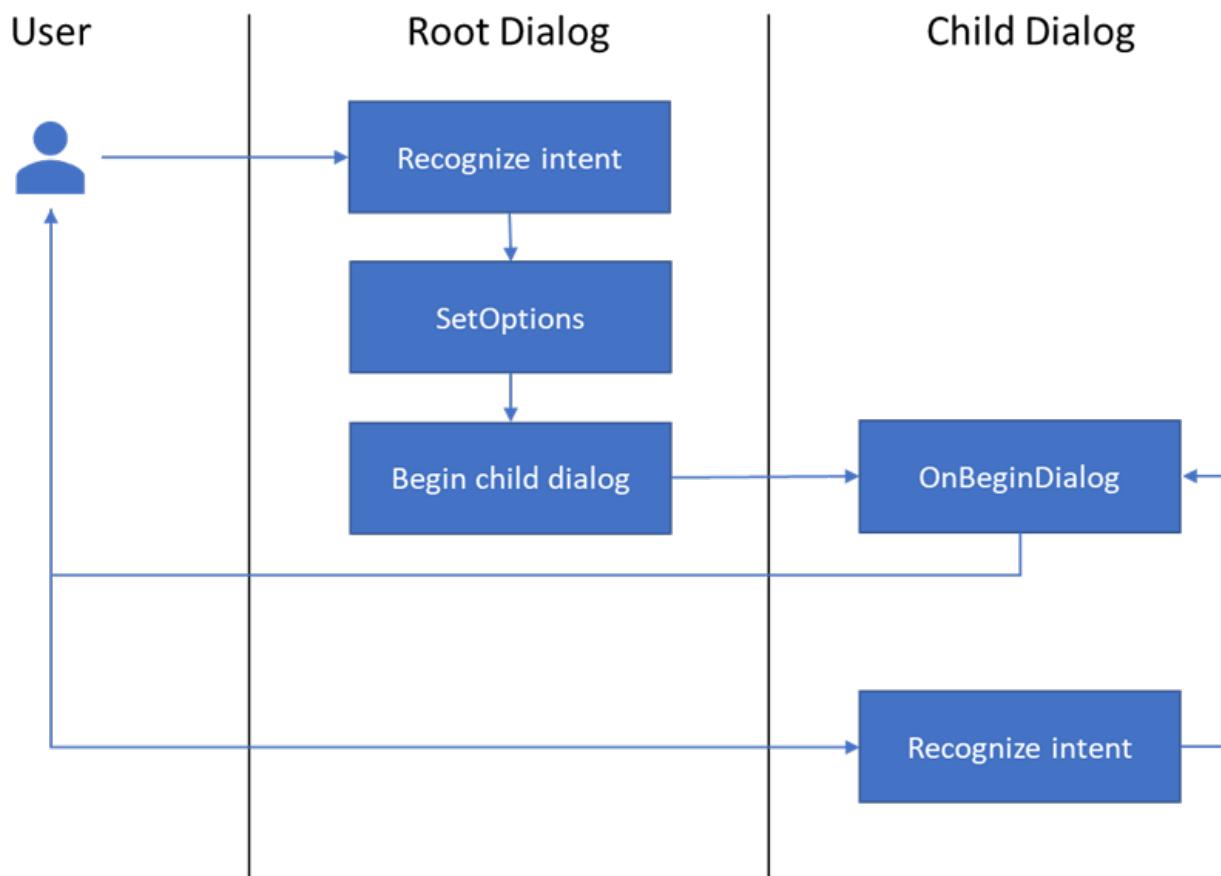
The Enterprise Calendar Bot template highlights the recommended best practices for complex conversational experiences. The bot demonstrates this through the following features:

- Adaptive Dialogs with Interruptions
- Language Understanding with LUIS
- Language Generation and Adaptive Cards
- Custom actions for Microsoft Graph

### Adaptive Dialogs with Interruptions

Adaptive Dialogs enable more flexibility and sophistication in dialog flows. Learn more about [Adaptive Dialogs](#). The Enterprise Calendar Bot uses these features to build flows that can be interrupted by configuring different rules that allow prompt responses to be evaluated against the full stack of language recognizers to find the best flow for the user's utterance. Each intent handled in the Enterprise Calendar Bot implements the following design pattern:

1. Recognize intent at Root Dialog level
2. Emit `SetOptions` custom event
  - a. Extract all entities and save them in bot state
  - b. Authenticate user
  - c. Get user profile from Microsoft Graph
3. Begin the child dialog and pass in entities as dialog options
4. Execute dialog logic in `OnBeginDialog` trigger of child dialog
5. For any activities received by during the execution of the child dialog, recognize intent and evaluate interruption rules



The Enterprise Calendar Bot contains the following dialogs:

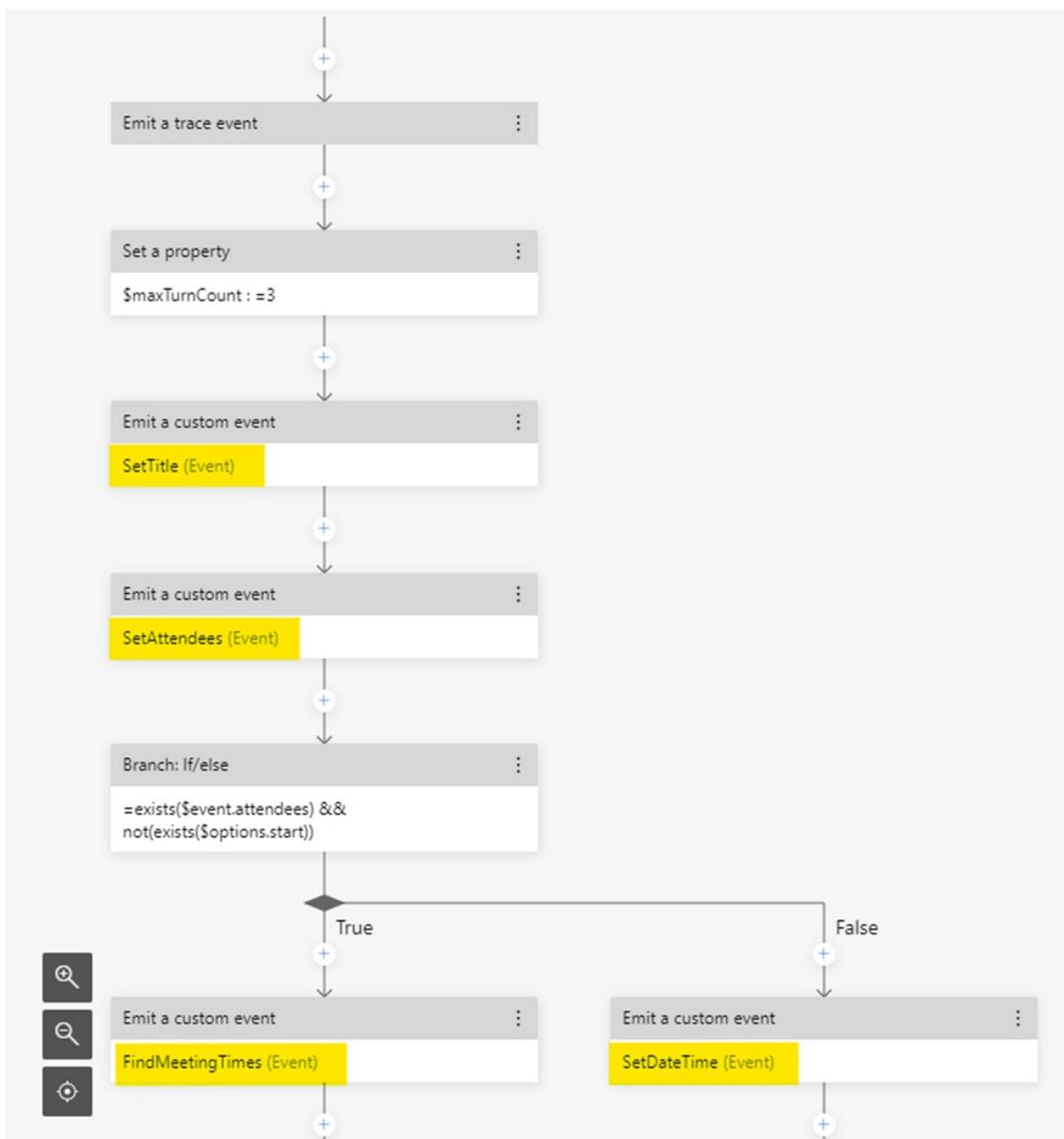
| DIALOG          | DESCRIPTION                                                                 |
|-----------------|-----------------------------------------------------------------------------|
| Authentication  | Signs in user and gets basic user information                               |
| CancelEvent     | Flow for cancelling existing calendar item                                  |
| CreateEvent     | Flow for creating a new calendar item                                       |
| GetContacts     | Flow for looking up contacts                                                |
| GetEvents       | Flow for looking up events                                                  |
| List            | Flow for displaying, navigating, and choosing from a list                   |
| ResolveDateTime | Flow for recognizing and disambiguating datetimes from the user's utterance |

| DIALOG         | DESCRIPTION                                        |
|----------------|----------------------------------------------------|
| RespondToEvent | Flow for accepting and declining event invitations |
| ShowEvents     | Flow for displaying calendar events                |
| UpdateEvent    | Flow for updating calendar events                  |

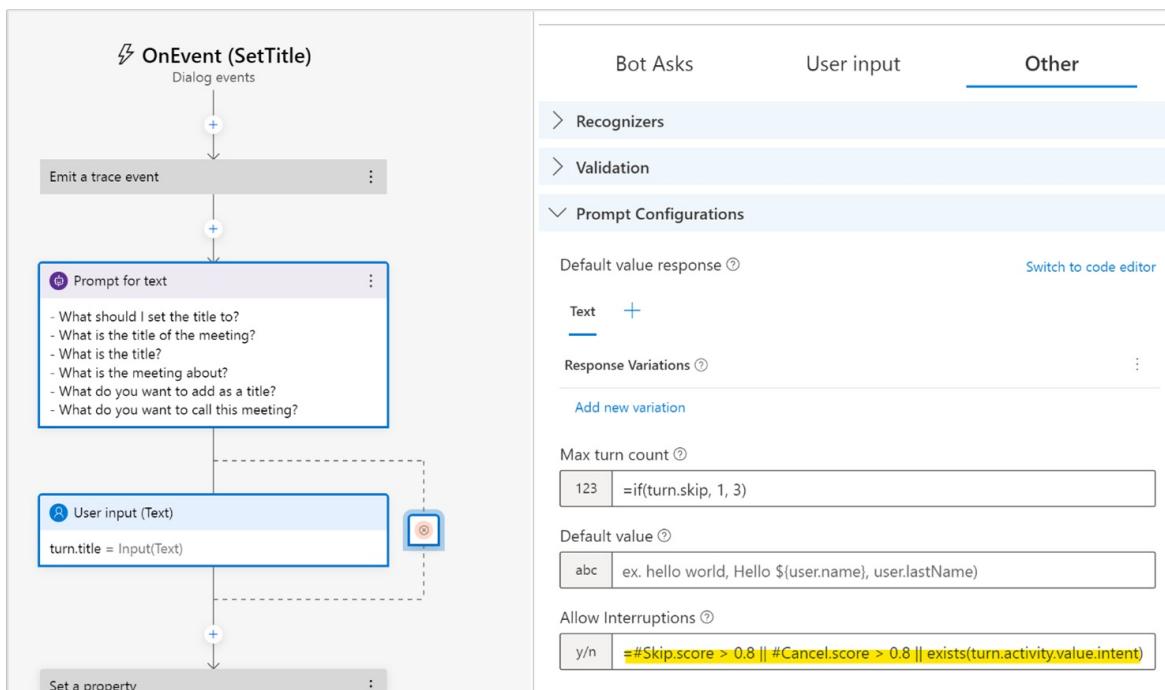
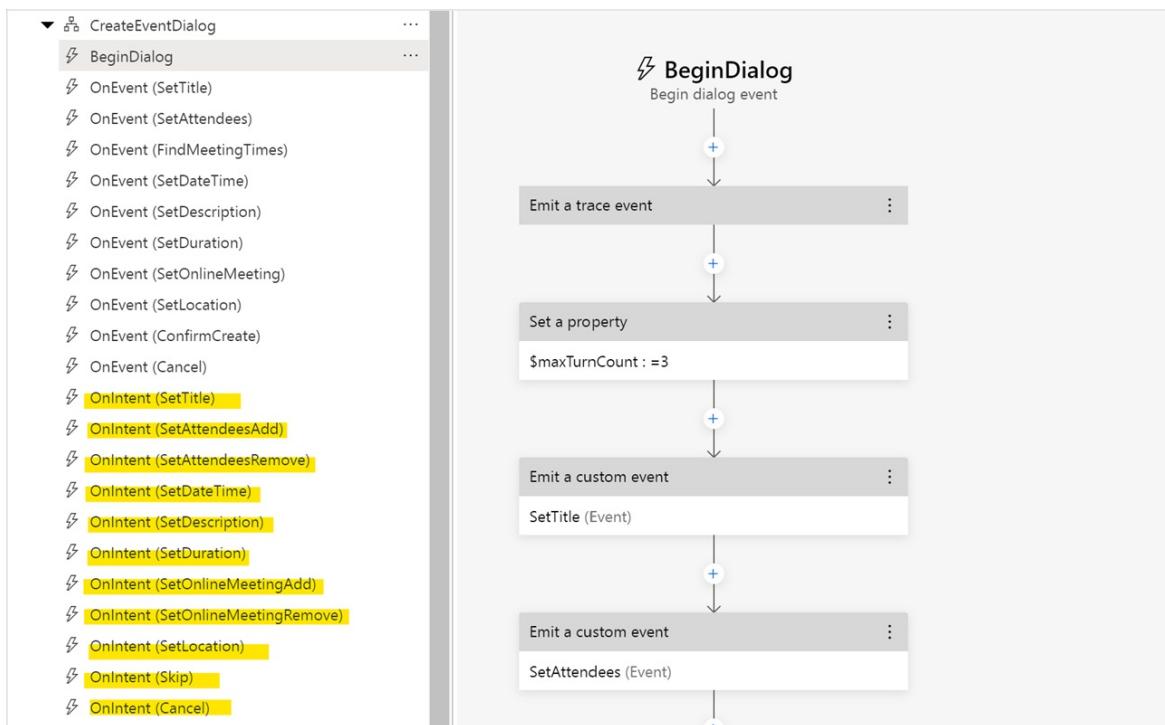
Each dialog follows these patterns:

- Each dialog contains a series of `OnEvent` trigger steps that are called from the `OnBeginDialog` trigger. This makes the flow easier to follow and allows the steps to be executed in a different order without duplication.

CreateEventDialog > BeginDialog



- `OnIntent` triggers have been added for each dialog where interruptions are allowed. Each Input action uses the `allowInterruptions` property to configure the rules that should apply to that specific prompt.



- Reusable flows have been factored out into separate dialogs so that the logic is consistent throughout. Some examples in this project are **GetEventsDialog**, **GetContactsDialog** **ListDialog**, and **ResolveDateTimeDialog**.

## Language understanding with LUIS

Luis provides the ability to recognize intents and extract entities from user utterances. In this project, each dialog that includes language understanding has its own Luis model containing the specific intents that should be recognized in the context of that dialog flow. If the dialog is configured to allow interruptions, but does not recognize an intent, the utterance may be evaluated against the other recognizers in the dialog stack. Learn more about [language understanding](#).

The language model included in this project recognizes the following intents:

| INTENT | EXAMPLE UTTERANCES |
|--------|--------------------|
|--------|--------------------|

| INTENT                    | EXAMPLE UTTERANCES                                                                                                                                                                             |
|---------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| CancelEvent               | <p><i>Cancel my meeting</i><br/> <i>I want to cancel the meeting this afternoon</i><br/> <i>Cancel my meeting with Thomas tomorrow</i></p>                                                     |
| CreateEvent               | <p><i>Schedule a meeting</i><br/> <i>Book a meeting with Sharon tomorrow</i><br/> <i>New meeting on Wednesday with Diego and Alex titled Project Sync</i></p>                                  |
| GetAvailabilityBreaks     | <p><i>When am I free today?</i><br/> <i>When do I have breaks on Friday?</i><br/> <i>Am I free tomorrow?</i><br/> <i>When do I start on Wednesday?</i><br/> <i>When does my day start?</i></p> |
| GetAvailabilityLast       | <p><i>When am I done on Friday?</i><br/> <i>When is my last meeting?</i><br/> <i>When am I done for the day?</i></p>                                                                           |
| GetEventAttendees         | <p><i>Who is attending the next meeting?</i><br/> <i>Who is coming to my meeting tomorrow morning?</i><br/> <i>Will Megan be at my next event?</i></p>                                         |
| GetEventDateTime          | <p><i>When are my meetings with Chris?</i><br/> <i>What time is my next meeting?</i><br/> <i>When am I meeting with Rebecca this week?</i></p>                                                 |
| GetEventLocation          | <p><i>Where is my meeting with Michael?</i><br/> <i>Where is the next meeting?</i><br/> <i>What room is my meeting in today?</i></p>                                                           |
| GetEvents                 | <p><i>What is on my schedule today?</i><br/> <i>What meetings do I have tomorrow?</i><br/> <i>What is up next?</i></p>                                                                         |
| RespondAccept             | <p><i>Accept the meeting on February 4th</i><br/> <i>I want to accept the next meeting</i><br/> <i>Accept the invite from Monica</i></p>                                                       |
| RespondDecline            | <p><i>Decline the meeting at 1pm</i><br/> <i>Decline the invite from Frances</i><br/> <i>Tell George I can't make the meeting tomorrow</i></p>                                                 |
| ResponseTentativelyAccept | <p><i>Tentatively accept the meeting with next week with Ryan</i><br/> <i>Mark me as tentative for the meeting this afternoon</i><br/> <i>Tell Louis I'm tentative today</i></p>               |
| SetAttendeesAdd           | <p><i>Add Donovan to the Project Review tomorrow</i><br/> <i>Forward the Project Review to Martin</i><br/> <i>Invite attendees to my meeting on Friday</i></p>                                 |
| SetAttendeesRemove        | <p><i>Remove attendees from a meeting</i><br/> <i>Remove Ron from the meeting tomorrow at 10am</i><br/> <i>Uninvite Tara to the Project Review today</i></p>                                   |

| INTENT                 | EXAMPLE UTTERANCES                                                                                                                                            |
|------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------|
| SetDateTime            | <i>Change the date of my meeting<br/>Update my meeting to start at noon on Saturday<br/>Change my hair appointment to Wednesday at 1pm</i>                    |
| SetDescription         | <i>Add a description to my meeting<br/>Add content to the meeting with John<br/>Set the description for the session tomorrow</i>                              |
| SetDuration            | <i>Change my meeting with Scott to 1 hour<br/>Extend the Design Sync to 2 hours<br/>Set the event to last 1 hour and 30 minutes</i>                           |
| SetLocation            | <i>Change the location of my next meeting<br/>Set the location of the meeting with Harry<br/>Update the location of the meeting tomorrow afternoon</i>        |
| SetOnlineMeetingAdd    | <i>Add a Microsoft Teams link<br/>Make my next meeting a Teams meeting<br/>Add a conference call to the meeting tomorrow afternoon</i>                        |
| SetOnlineMeetingRemove | <i>Remove the conference call from the meeting with Elaine<br/>Don't include a Teams link in the next meeting<br/>Remove Microsoft Teams from the meeting</i> |
| SetTitle               | <i>Update the title of my next meeting<br/>Set the subject of the meeting with Erin to Team Lunch<br/>Change the title of the session tomorrow morning</i>    |
| UpdateEvent            | <i>Update my meeting with Brian and Nicole<br/>Modify the Budget Review on Tuesday<br/>Edit my meeting on Friday</i>                                          |

## Language generation with Adaptive Cards

Language generation is the tool for writing bot responses in Composer. The Enterprise Calendar Bot uses a variety of LG templates including [Adaptive Cards](#). To enable the easy reuse of assets and cards, the templates are broken down into the following files which are imported throughout the project:

- **Actions.lg** – contains all the actions used in cards throughout the project. This helps keep all the actions consistent.
- **Icons.lg** – contains all the icons used throughout the project. This simplifies any card templates and makes updating the icons simpler.
- **Functions.lg** – contains any common functions used throughout the project.
- **Cards.lg** – contains standard Adaptive Card templates and structure. This reduces the time required to create cards and ensures consistency in layout.

## Custom actions for Microsoft Graph

The Enterprise Calendar Bot uses the Microsoft Graph API to access and manage the user's calendar. To simplify the visual flows in Composer, the Microsoft Graph API calls are made in code and exposed as custom actions in Composer. This enables the reuse of the Microsoft Graph actions in multiple dialogs and projects, as well as a more friendly experience in Composer. Learn more about [custom actions](#).

Here is what the custom Create Event action looks like in Composer:

## Microsoft Graph - Create Event

Microsoft.Graph.Calendar.CreateEvent

Create a calendar event using the Microsoft Graph API.

Result property [?](#)

abc \$createEventResult

Token \* [?](#)

abc =turn.token.token

New event \* [?](#)

event [▼](#)

Attendees [?](#)

```
[ ] [ {  
    "EmailAddress": {  
        "Address": "<attendee ad  
        ...  
    }  
} ]
```

Subject [?](#)

abc ex. My event

Start [?](#)

abc ex. 2021-01-01T00:00:00

End [?](#)

abc ex. 2021-01-01T00:00:00

Location [?](#)

abc ex. Building A, Seattle, WA

Description [?](#)

abc ex. My event description

Include Online Meeting [?](#)

y/n

[▼](#)

## Data usage and storage

The Enterprise Calendar Bot requires access to users' [Microsoft Graph](#) data. The following tables outline what data is accessed, where it is stored, and the default data retention policy for each service. We recommend reviewing this information and adjusting the default configuration of your services to meet your organization's compliance standards.

### Storage locations

This section outlines the different data stores used in a typical bot lifecycle, their purpose, and our recommendations for configuring them.

#### Azure Cosmos DB (Required)

[Cosmos DB](#) is the default and recommended provider for storing Bot State. Bot State is split into 4 different scopes with different default lifetimes over the course of a conversation. The default scopes are as follows:

- **Conversation State:** For storing values that should persist over the lifetime of a conversation with a user on a given channel. In a channel with persistent conversations such as Microsoft Teams or Facebook Messenger,

this could be stored indefinitely.

- **User State:** For storing values that should persist over the lifetime of a user on a given channel. In a channel with persistent conversations such as Microsoft Teams or Facebook Messenger, this could be stored indefinitely.
- **Dialog State:** For storing values that should persist for the duration of a dialog. If a dialog is left in a waiting state in a channel with persistent conversations such as Microsoft Teams or Facebook Messenger, this could be stored indefinitely.
- **Turn State:** For storing values that persist only for the current turn. A turn indicates the time between the user's response. Every time a new activity from a user is received, Turn State is reset. Turn State properties are never written to a persistent storage location.

| SCOPE        | MS GRAPH DATA STORED BY ENTERPRISE PEOPLE BOT                                                                                       |
|--------------|-------------------------------------------------------------------------------------------------------------------------------------|
| Conversation | N/A                                                                                                                                 |
| User         | Working hours<br>Time zone                                                                                                          |
| Dialog       | Events<br>Attendees (email and display name only)<br>Contacts (email and display name only)<br>People (email and display name only) |
| Turn         | User access token<br>User profile                                                                                                   |

Due to the potentially sensitive data stored by the skills in Bot State, we recommend the following configuration for your Cosmos DB resource:

- Configure a default Time to Live (TTL) for your Cosmos DB container. This value should be adjusted according to your organization's standards. [Learn more](#).

#### Application Insights (Recommended)

If you have [Application Insights](#) provisioned and configured in your Project Settings, each incoming and outgoing activity will be logged including varying properties depending on your settings. You can configure your telemetry settings in your Project Settings in Composer by toggling the following properties:

```
"runtimeSettings": {  
    "telemetry": {  
        "logActivities": true,  
        "logPersonalInformation": false  
    }  
}
```

The `logActivities` property controls whether activity will be logged in Application Insights, and the `logPersonalInformation` property controls whether the logged activities will include the `Activity.Text`, `Activity.Speak`, and `Activity.From.Name` properties. By default, any telemetry stored in Application Insights will follow the default retention duration. You may want to change this behavior depending on your organization's standards. [Learn more](#).

#### Azure Blob Storage (Optional)

If you have transcript storage configured in your Project Settings, each incoming and outgoing activity will be logged in Azure Blob Storage. This is useful for analysis of your bot's conversations but should be evaluated against your organization's compliance standards. You can configure transcript storage in your Project Settings in Composer by toggling the following property:

```
"runtimeSettings": {  
    "features": {  
        "traceTranscript": false  
    }  
}
```

#### NOTE

When emitting trace events in your dialogs, they may be written to Application Insights and Azure Blob Storage depending on your project configuration. Trace events are only sent when communicating with your bot via the Bot Framework Emulator.

## Next steps

To get started using the Enterprise Calendar Bot template in Bot Framework Composer, follow the steps in [Enterprise Calendar Bot Tutorial](#)

# Tutorial: Set up your Enterprise Calendar Bot

5/20/2021 • 3 minutes to read

APPLIES TO: Composer v2.x

## Create your project in Composer

The Enterprise Calendar Bot template is included in the Bot Framework Composer by default. Follow these steps to create a project:

1. Open Bot Framework Composer (version 2.0.0 or higher)
2. Click **Create new**
3. Select **Enterprise Calendar Bot** from the list
4. Enter your desired **Name**, **Location**, and **Runtime type** then select **Create**

## Provision Azure resources

The Enterprise Calendar Bot requires Azure resources in order to run locally. Follow these steps to provision the required resources:

1. Create a publishing profile and provision resources according to the steps in [Publish a bot to Azure](#). The following resources are required for local development:
  - Microsoft Application Registration
  - Azure Hosting
  - Microsoft Bot Channels Registration
  - Microsoft Language Understanding Authoring Account
2. Fill in the following settings in **Configure > Development Resources** with the resources you provisioned:
  - a. Language Understanding authoring key
  - b. Microsoft App Id
  - c. Microsoft App Password

## Configure authentication

You must configure an authentication connection on your Azure Bot in order to log in and access Microsoft Graph resources. You can configure these settings either through the Azure Portal or via the Azure CLI.

### Option 1: Using the Azure Portal

1. Open your Azure Bot resource and go to the **Configuration** tab
2. Click **Add OAuth Connection Settings**
3. Assign your connection setting a name (save this value for later)
4. Select **Azure Active Directory v2** from the Service Provider dropdown.
5. Fill in the following fields and click **Save**:
  - **Client id**: your Microsoft App ID
  - **Client secret**: your Microsoft App Password

- **Tenant ID:** your Azure Active Directory tenant ID, or "common" to support any tenant
- **Scopes:** Calendars.ReadWrite Contacts.Read People.Read User.ReadBasic.All User.Read

6. In the **Configuration** tab, click **Manage** next to your Microsoft App ID

7. In the API permissions tab, click **Add a permission**

8. Click **Microsoft Graph > Delegated Permissions** and add the following scopes:

- Calendars.ReadWrite
- Contacts.Read
- People.Read
- User.ReadBasic.All
- User.Read

9. In the Authentication tab, click **Add a platform**

a. Select **Web**

b. Set the URL to <https://token.botframework.com/.auth/web/redirect>

10. In Bot Framework Composer, open your **Project Settings** and toggle the **Advanced Settings View**

11. Set the following property to the value from Step 3:

```
{
  "oauthConnectionName": "Outlook",
}
```

## Option 2: Using Azure CLI

1. Get your Microsoft App Object ID (used in later steps):

```
az ad app show --id <bot-app-id> --query objectId
```

2. Set the Redirect URL on your Microsoft App:

```
az rest --method patch --url https://graph.microsoft.com/v1.0/applications/<objectId> --body "{'web': {'redirectUris': ['https://token.botframework.com/.auth/web/redirect']}}"
```

3. Add the required Microsoft Graph scopes to your Microsoft App:

```
az rest --method patch --url https://graph.microsoft.com/v1.0/applications/<objectId> --body "{'requiredResourceAccess': [ {'resourceAppId': '00000003-0000-0000-c000-000000000000', 'resourceAccess': [ { 'type': 'Scope', 'id': 'ba47897c-39ec-4d83-8086-ee8256fa737d' }, { 'type': 'Scope', 'id': 'ff74d97f-43af-4b68-9f2a-b77ee6968c5d' }, { 'type': 'Scope', 'id': '1ec239c2-d7c9-4623-a91a-a9775856bb36' }, { 'type': 'Scope', 'id': 'b340eb25-3456-403f-be2f-af7a0d370277' }, { 'type': 'Scope', 'id': 'e1fe6dd8-ba31-4d61-89e7-88639da4683d' } ]} ]}"
```

4. Add your OAuth setting to your Azure Bot Service. The values for `bot-name`, `bot-rg`, `bot-app-id`, and `bot-app-secret` can be found in your bot's publish profile under **Publish > Publishing profile > Edit > Import existing resources > Next**.

```
az bot authsetting create --name <bot-name> --resource-group <bot-rg> --client-id <bot-app-id> --client-secret <bot-app-secret> --service "Aadv2" --setting-name "Outlook" --provider-scope-string "Calendars.ReadWrite Contacts.Read People.Read User.Read User.ReadBasic.All" --parameters clientId="<bot-app-id>" clientSecret="<bot-app-secret>" tenantId=common
```

5. Update your Bot settings with your OAuth Connection name in the **Advanced Settings View**:

```
{  
  "oauthConnectionName": "Outlook",  
}
```

## Next steps

After you have completed the preceding steps to run and test locally, you can follow these steps to publish your bot to Azure:

### Publish to Azure

To publish your Enterprise Calendar Bot, follow these steps:

1. Create a publishing profile and provision resources according to the steps in [Publish a bot to Azure](#). The following resources are required for production environments:

- Microsoft Application Registration
- Azure Hosting
- Microsoft Bot Channels Registration
- Azure Cosmos DB
- Application Insights
- Microsoft Language Understanding Authoring Account
- Microsoft Language Understanding Prediction Account

2. Publish your bot via the **Publish** tab

# Enterprise People Bot Template

5/20/2021 • 7 minutes to read

APPLIES TO: Composer v2.x

## Overview

The bot helps to discover who a colleague is in an organization and the realm of their interconnectivity with other colleagues (peers, frequent collaborators, their manager).

## Conversational experience

The following are the applicable core scenarios:

### Enterprise People Bot

- Contact details of a colleague
- Their manager
- Works with
- Peers of
- Collaborators
- Directly call/email/message

### Showcase

- "Who is.."
- "Who is the manager of"
- "Who are frequent collaborators of...", and much more!

[Learn more](#) about the design principles used in the Enterprise Assistant Bot which includes the Enterprise People Bot as a skill.

## Developer experience

The Enterprise People Bot template highlights the recommended best practices for conversational experiences:

- Adaptive Dialogs
- Language Understanding with LUIS
- Language Generation and Adaptive Cards
- Custom actions for Microsoft Graph

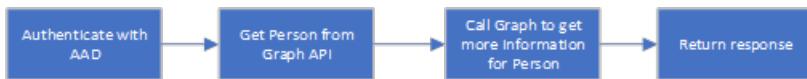
### Adaptive Dialogs

Adaptive Dialogs enable more flexibility and sophistication in dialog flows. Learn more about [Adaptive Dialogs](#). In this section, we will discuss the overall design of the adaptive dialogs in the Enterprise People Bot, and some key concepts in the design.

Each intent handled in the Enterprise People bot implements the following design pattern:

1. Authenticate with Azure Active Directory
2. Get Person object from Microsoft Graph API
3. Get additional information for Person from Graph API

#### 4. Return card with relevant details



The Enterprise People Bot contains the following dialogs:

| DIALOG                 | DESCRIPTION                                                                    |
|------------------------|--------------------------------------------------------------------------------|
| Authentication         | Signs the user into Azure Active Directory                                     |
| FindPersonDialog       | Flow for searching for a user by their name or email                           |
| GetCollaboratorsDialog | Flow for getting the frequent collaborators of a user                          |
| GetDirectReportsDialog | Flow for getting a user's direct report                                        |
| GetManagerDialog       | Flow for getting a user's manager                                              |
| GetPeersDialog         | Flow for getting a user's peers who share same manager                         |
| GetProfileDialog       | Flow for getting a user's profile                                              |
| ListDialog             | Flow for displaying, navigating and choosing from a list                       |
| ResolveUserDialog      | Flow for getting a user's profile from a list of users                         |
| UnknownIntentDialog    | Handles if the intent is unknown to the LU recognizer                          |
| UserListDialog         | Flow for handling displaying a list of users (include getting photos, profile) |

Each dialog follows these patterns:

- The ResolveUserDialog is called to attempt to identify the user to look up.
- Once the user has been identified, additional user data is retrieved from Microsoft Graph API.



- Reusable flows have been factored out into separate dialogs so that the logic is consistent throughout. Some examples in this project are **Authentication**, **ResolveUserDialog**, **ListDialog**, **GetProfileDialog**.

## Language Understanding with LUIS

LUIS provides the ability to recognize intents and extract entities from user utterances. Learn more about [language understanding](#).

The language model included in this project recognizes the following intents:

| INTENT           | EXAMPLE UTTERANCES                                                                              |
|------------------|-------------------------------------------------------------------------------------------------|
| GetManager       | Find the manager of John<br>Who is the manager of Janet?<br>What is the name of Jake's manager? |
| GetDirectReport  | Who reports to Hayden?<br>Please show me who Amy manages?<br>Tell me the reports of Mary        |
| GetPeers         | Show me Harry's team<br>Who are Kevin's colleagues?<br>Who are Lauren's peers?                  |
| GetProfile       | Tell me more about Megan<br>Who is Jackson?<br>I want to know who Morgan is                     |
| GetCollaborators | Who collaborates with me most?                                                                  |

## Language Generation with Adaptive Cards

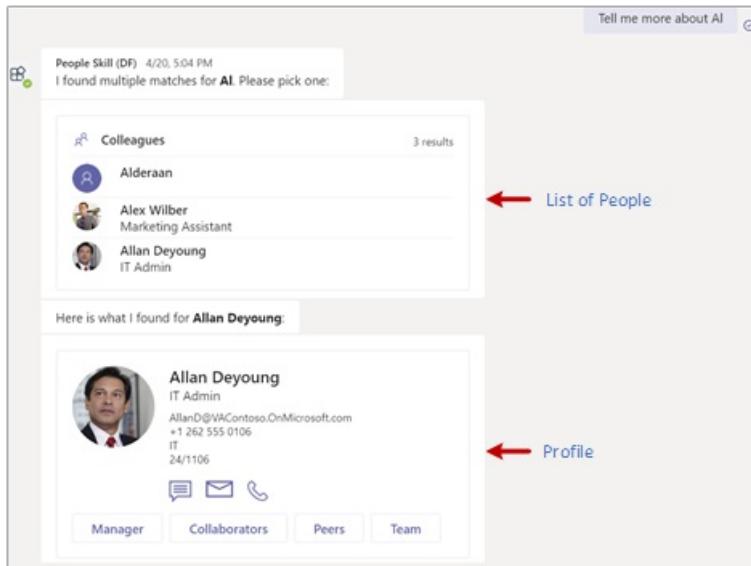
Language Generation is the tool for writing bot responses in Composer. The Enterprise People Bot uses a variety of LG templates and techniques. To enable the easy reuse of assets and cards, the templates are broken down into the following files which are imported throughout the project:

- **Actions.lg** – contains all the actions used in cards throughout the project. This helps keep all the actions consistent.
- **Icons.lg** – contains all the icons used throughout the project. This simplifies any card templates and makes updating the icons simpler.
- **Functions.lg** – contains any common functions used throughout the project.
- **Cards.lg** – contains standard Adaptive Card templates and structure. This reduces the time required to create cards and ensures consistency in layout.
- **Profile.lg** - contains the Adaptive Card templates for the user profile card
- **UserList.lg** - contains the Adaptive Card templates for the user list card

#### Adaptive Cards

The Enterprise People Bot uses [Adaptive Cards](#) to display profile information about a person and render lists of people.

The following screenshot shows the two types of cards we render in the Enterprise People Bot:



Each card is composed of Adaptive Card Containers that the user can select to trigger further actions. For example, in the profile card, the user can trigger the following actions by selecting the appropriate icon:

- Email the person
- Message the person via Teams
- Call the person via Teams

Additionally, they can select actions which trigger other intents in the bot:

- Get the manager of the person
- Get frequent collaborators of the person
- Get peers of the person (colleagues share same manager)
- Get direct reports of the person ("team")

#### Custom actions for Microsoft Graph

The Enterprise People Bot uses the Microsoft Graph API to answer questions about the people in an organization. To simplify the visual flows in Composer, the Microsoft Graph API calls are made in code and exposed as custom actions in Composer. This enables the reuse of the Microsoft Graph actions in multiple dialogs and projects, as well as a more friendly experience in Composer. Learn more about [custom actions](#).

Here is what the custom Get Direct Reports action looks like in Composer:

Microsoft Graph - Get direct reports of a user  
Microsoft.Graph.User.GetDirectReports  
Get the direct reports of a user in the organization using the Microsoft Graph API.

Result property ⓘ  
abc turn.items

Token \* ⓘ  
abc =turn.token.token

Max Results ⓘ  
123 15

User Properties To Select ⓘ  
abc id  
abc displayName  
abc mail  
abc businessPhones  
abc officeLocation  
abc jobTitle  
abc department  
ex. id, displayName, mail, officeLocation, jobTitle

User ID \* ⓘ  
abc =turn.UserFound.Id

Packag.es/Graph/Schemas/  
Microsoft.Graph.Users.GetDirectReports.schema

## Data usage and storage

The Enterprise People Bot requires access to users' [Microsoft Graph](#) data. The following tables outline what data is accessed, where it is stored, and the default data retention policy for each service. We recommend reviewing this information and adjusting the default configuration of your services to meet your organization's compliance standards.

### Storage locations

This section outlines the different data stores used in a typical bot lifecycle, their purpose, and our recommendations for configuring them.

#### Azure Cosmos DB (Required)

[Cosmos DB](#) is the default and recommended provider for storing Bot State. Bot State is split into 4 different scopes with different default lifetimes over the course of a conversation. The default scopes are as follows:

- **Conversation State:** For storing values that should persist over the lifetime of a conversation with a user on a given channel. In a channel with persistent conversations such as Microsoft Teams or Facebook Messenger, this could be stored indefinitely.
- **User State:** For storing values that should persist over the lifetime of a user on a given channel. In a channel with persistent conversations such as Microsoft Teams or Facebook Messenger, this could be stored indefinitely.
- **Dialog State:** For storing values that should persist for the duration of a dialog. If a dialog is left in a waiting state in a channel with persistent conversations such as Microsoft Teams or Facebook Messenger, this could be stored indefinitely.
- **Turn State:** For storing values that persist only for the current turn. A turn indicates the time between the user's response. Every time a new activity from a user is received, Turn State is reset. Turn State properties are never written to a persistent storage location.

| SCOPE        | MS GRAPH DATA STORED BY ENTERPRISE PEOPLE BOT |
|--------------|-----------------------------------------------|
| Conversation | N/A                                           |

| SCOPE  | MS GRAPH DATA STORED BY ENTERPRISE PEOPLE BOT                                                                                                                         |
|--------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| User   | N/A                                                                                                                                                                   |
| Dialog | N/A                                                                                                                                                                   |
| Turn   | User access token<br>User profile<br>- Phone number<br>- Email<br>- Job title<br>- Name<br>- Office Location<br>- Display name<br>- ID (Graph use only)<br>User photo |

Due to the potentially sensitive data stored by the bot in Bot State, we recommend the following configuration for your Cosmos DB resource:

- Configure a default Time to Live (TTL) for your Cosmos DB container. This value should be adjusted according to your organization's standards. [Learn more](#).

#### Application Insights (Recommended)

If you have [Application Insights](#) provisioned and configured in your Project Settings, each incoming and outgoing activity will be logged including varying properties depending on your settings. You can configure your telemetry settings in your Project Settings in Composer by toggling the following properties:

```
"runtimeSettings": {
  "telemetry": {
    "logActivities": true,
    "logPersonalInformation": false
  }
}
```

The `logActivities` property controls whether activity will be logged in Application Insights, and the `logPersonalInformation` property controls whether the logged activities will include the `Activity.Text`, `Activity.Speak`, and `Activity.From.Name` properties. By default, any telemetry stored in Application Insights will follow the default retention duration. You may want to change this behavior depending on your organization's standards. [Learn more](#).

#### Azure Blob Storage (Optional)

If you have transcript storage configured in your Project Settings, each incoming and outgoing activity will be logged in [Azure Blob Storage](#). This is useful for analysis of your bot's conversations but should be evaluated against your organization's compliance standards. You can configure transcript storage in your Project Settings in Composer by toggling the following property:

```
"runtimeSettings": {
  "features": {
    "traceTranscript": false
  }
}
```

**NOTE**

When emitting trace events in your dialogs, they may be written to Application Insights and Azure Blob Storage depending on your project configuration. Trace events are only sent when communicating with your bot via the Bot Framework Emulator.

## Next steps

To get started using the Enterprise People Bot template in Bot Framework Composer, follow the steps in

[Enterprise People Bot Tutorial](#)

# Tutorial: Set up your Enterprise People Bot

5/20/2021 • 3 minutes to read

APPLIES TO: Composer v2.x

## Create your project in Composer

The Enterprise Calendar Bot template is included in the Bot Framework Composer by default. Follow these steps to create a project:

1. Open Bot Framework Composer (version 2.0.0 or higher)
2. Click **Create new**
3. Select **Enterprise Calendar Bot** from the list
4. Enter your desired **Name**, **Location**, and **Runtime type** then select **Create**

## Provision Azure resources

The Enterprise People Bot requires Azure resources in order to run locally. Follow these steps to provision the required resources:

1. Create a publishing profile and provision resources according to the steps in [Publish a bot to Azure](#). The following resources are required for local development:
  - Microsoft Application Registration
  - Azure Hosting
  - Microsoft Bot Channels Registration
  - Microsoft Language Understanding Authoring Account
2. Fill in the following settings in **Configure > Development Resources** with the resources you provisioned:
  - a. Language Understanding authoring key
  - b. Microsoft App Id
  - c. Microsoft App Password

## Configure authentication

You must configure an authentication connection on your Azure Bot in order to log in and access Microsoft Graph resources. You can configure these settings either through the Azure Portal or via the Azure CLI.

### Option 1: Using the Azure Portal

1. Open your Azure Bot resource and go to the **Configuration** tab
2. Select **Add OAuth Connection Settings**
3. Assign your connection setting a name (save this value for later)
4. Select **Azure Active Directory v2** from the Service Provider dropdown.
5. Fill in the following fields and click **Save**:
  - **Client id**: your Microsoft App ID
  - **Client secret**: your Microsoft App Password

- **Tenant ID:** your Azure Active Directory tenant ID, or "common" to support any tenant
- **Scopes:** Contacts.Read Directory.Read.All People.Read People.Read.All User.ReadBasic.All User.Read.All

6. In the **Configuration** tab, click **Manage** next to your Microsoft App ID

7. In the API permissions tab, click **Add a permission**

8. Select **Microsoft Graph > Delegated Permissions** and add the following scopes:

- Contacts.Read
- Directory.Read.All
- People.Read
- People.Read.All
- User.ReadBasic.All
- User.Read.All

9. In the Authentication tab, click **Add a platform**

a. Select **Web**

b. Set the URL to <https://token.botframework.com/.auth/web/redirect>

10. In Bot Framework Composer, open your **Project Settings** and toggle the **Advanced Settings View**

11. Set the following property to the value from Step 3:

```
{
  "oauthConnectionName": "Outlook",
}
```

## Option 2: Using Azure CLI

1. Get your Microsoft App Object ID (used in later steps):

```
az ad app show --id <bot-app-id> --query objectId
```

2. Set the Redirect URL on your Microsoft App:

```
az rest --method patch --url https://graph.microsoft.com/v1.0/applications/<objectId> --body "{'web': {'redirectUris': ['https://token.botframework.com/.auth/web/redirect']}}"
```

3. Add the required Microsoft Graph scopes to your Microsoft App:

```
az rest --method patch --url https://graph.microsoft.com/v1.0/applications/<objectId> --body "{'requiredResourceAccess': [{ 'resourceAppId': '00000003-0000-0000-0000-000000000000', 'resourceAccess': [{ 'id': 'b89f9189-71a5-4e70-b041-9887f0bc7e4a', 'type': 'Scope' }, { 'id': 'b340eb25-3456-403f-be2f-af7a0d370277', 'type': 'Scope' }, { 'id': 'a154be20-db9c-4678-8ab7-66f6cc099a59', 'type': 'Scope' }, { 'id': '06da0dbc-49e2-44d2-8312-53f166ab848a', 'type': 'Scope' }, { 'id': 'ff74d97f-43af-4b68-9f2a-b77ee6968c5d', 'type': 'Scope' }, { 'id': 'ba47897c-39ec-4d83-8086-ee8256fa737d', 'type': 'Scope' }]} ]}"
```

4. Add your OAuth setting to your Azure Bot Service. The values for `bot-name`, `bot-rg`, `bot-app-id`, and `bot-app-secret` can be found in your bot's publish profile under **Publish > Publishing profile > Edit > Import existing resources > Next.**

```
az bot authsetting create --name <bot-name> --resource-group <bot-rg> --client-id <bot-app-id> --client-secret <bot-app-secret> --service "Aadv2" --setting-name "Outlook" --provider-scope-string "Contacts.Read Directory.Read.All People.Read People.Read.All User.ReadBasic.All User.Read.All" --parameters clientId=<bot-app-id> clientSecret=<bot-app-secret> tenantId=common
```

5. Update your Bot settings with your OAuth Connection name in the **Advanced Settings View**:

```
{  
    "oauthConnectionName": "Outlook",  
}
```

## Next steps

After you have completed the preceding steps to run and test locally, you can follow these steps to publish your bot to Azure:

### Publish to Azure

To publish your Enterprise People Bot, follow these steps:

1. Create a publishing profile and provision resources according to the steps in [Provision in Composer](#). The following resources are required for production environments:
  - Microsoft Application Registration
  - Azure Hosting
  - Microsoft Bot Channels Registration
  - Azure Cosmos DB
  - Application Insights
  - Microsoft Language Understanding Authoring Account
  - Microsoft Language Understanding Prediction Account
2. Publish your bot via the **Publish** tab

# Learn from samples

5/12/2021 • 3 minutes to read

Bot Framework Composer provides example bots designed to illustrate the scenarios you are most likely to encounter when developing your own bots. This article is designed to help you make the best use of these examples. You will learn how to create a new bot based off any of the examples, which you can use to learn from or as a starting point when creating your own bots in Composer.

## Prerequisites

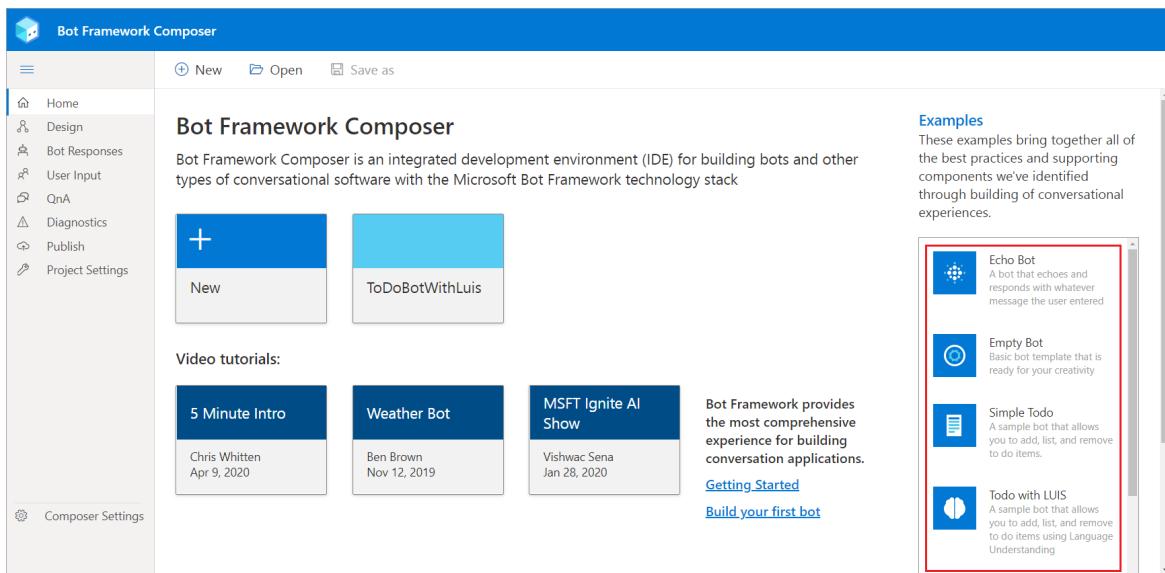
- [Install Bot Framework Composer](#).
- (Optional) [LUIS account](#) and a [LUIS authoring key](#).

## Open a sample

The **Examples** can be found on the right side of the Composer home page.

To open a bot sample from Composer follow the steps:

1. Select the sample you want to open from the **Examples** list.



2. In the **Create a bot project** form:

## Create a bot project

X

Specify a name, description, and location for your new bot project.

|                         |             |
|-------------------------|-------------|
| Name                    | Description |
| AskingQuestionsSample-0 |             |

Location

|            | Name ↑ | Date Modified     |
|------------|--------|-------------------|
| ..         |        | a few seconds ago |
| WeatherBot |        | 8 days ago        |

- a. **Name:** You can use the default name provided or enter a new name here.
  - b. **Description (optional):** Descriptive text to describe the bot you are creating.
  - c. **Location:** The location your bot source code files will be saved to.
3. After you select **OK** in the **Create a bot project** form, the new bot based off the example bot you selected will open in Composer.

The screenshot shows the Bot Framework Composer application. On the left, there's a sidebar with various project management and development tools like Home, Design, Bot Responses, User Input, QnA, Diagnostics, Publish, and Project Settings. The main workspace on the right is titled 'YOUR PROJECT' and shows a tree view of the 'AskingQuestionsSample-0' project. Under this project, several dialog nodes are listed, each with a small icon and a brief description. To the right of the tree view, there's a detailed configuration panel for an 'Adaptive dialog'. It includes sections for 'Language Understanding' (set to 'Regular expression recognizer'), 'Auto end dialog' (set to 'y/n false'), and 'Default result property' (set to 'abc | dialog.result'). Below these settings is a section titled 'Dialog Interface'. A large, semi-transparent hexagonal icon with two white dots is overlaid on the workspace area. A tooltip message 'Select a trigger on the left navigation to see actions' is visible near the bottom of the workspace.

### NOTE

When you select a bot from the **Examples** list, a copy of the original sample is created in the **Location** you specify in the **Create a bot project** form. Any changes you make to that bot will be saved without affecting the original example. You can create as many bots based off the examples as you want, without impacting the original examples and you are free to modify and use them as a starting point when creating your own bots.

4. Select the **Start all bots** button located on the Composer toolbar, then select **Test in Emulator** to test your new bot in the Bot Framework Emulator.

Welcome to Input Sample Bot.

I can show you examples on how to use actions, You can enter number 01-07

01 - TextInput  
02 - NumberInput  
03 - ConfirmInput  
04 - ChoiceInput  
05 - AttachmentInput  
06 - DateTimeInput  
07 - OAuthInput

Type your message ➤

```
[12:58:11] Emulator listening on http://[::]:59653
[12:58:11] ngrok listening on https://cf2142e4.ngrok.io
[12:58:11] ngrok traffic inspector: http://127.0.0.1:4040
[12:58:11] Will bypass ngrok for local addresses
[12:58:11] -> conversationUpdate
[12:58:11] <- message Welcome to Input Sample Bot. I can
show you examp...
[12:58:11] POST 200
conversations/<conversationId>/activities/<activityId>
[12:58:11] <- trace Bot State
[12:58:11] POST 200
conversations/<conversationId>/activities/<activityId>
[12:58:11] POST 200
directline/conversations/<conversationId>/activities
```

## Learn from Samples

Composer currently provides eleven bot samples with different specialties. These samples are a good resource to learn how to build your own bot using Composer. You can use the samples and learn how to [send text messages](#), [how to ask questions](#), and [how to control conversation flow](#), etc.

Below is a table of the eleven bot samples in Composer and their respective descriptions.

| SAMPLE                        | DESCRIPTION                                                                                                                                                                             |
|-------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Echo Bot                      | A bot that echoes whatever message the user enters.                                                                                                                                     |
| Empty Bot                     | A basic bot that is ready for your creativity.                                                                                                                                          |
| Simple Todo                   | A sample bot that shows how to use Regex recognizer to define intents and allows you to add, list and remove items.                                                                     |
| Todo with LUIS                | A sample bot that shows how to use LUIS recognizer to define intents and allows you to add, list and remove items. A <a href="#">LUIS authoring key</a> is required to run this sample. |
| Asking Questions              | A sample bot that shows how to prompt user for different types of input.                                                                                                                |
| Controlling Conversation Flow | A sample bot that shows how to use branching actions to control a conversation flow.                                                                                                    |
| Dialog Actions                | A sample bot that shows how to use actions in Composer (does not include <b>Ask a question</b> actions already covered in the <b>Asking Questions</b> example).                         |
| Interruptions                 | A sample bot that shows how to handle interruptions in a conversation flow. A <a href="#">LUIS authoring key</a> are required to run this sample.                                       |

| SAMPLE                | DESCRIPTION                                                                                                                                                               |
|-----------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| QnA Maker and LUIS    | A sample bot that shows how to use both QnA Maker and LUIS. A <a href="#">LUIS authoring key</a> and a <a href="#">QnA Knowledge Base</a> is required to run this sample. |
| QnA Sample            | A sample bot that is provisioned to enable users to create QnA Maker knowledge base in Composer.                                                                          |
| Responding with Cards | A sample bot that shows how to send different cards using language generation.                                                                                            |
| Responding with Text  | A sample bot that shows how to send different text messages to users using language generation.                                                                           |

## Next

- Learn [how to send text messages](#).

# Respond with messages

5/20/2021 • 8 minutes to read

**APPLIES TO:** Composer v1.x and v2.x

The primary way a bot communicates with users is through message activities. Some messages may simply consist of plain text, while others may contain richer content such as [cards](#). In this article, you will learn the different types of text messages you can use in Bot Framework Composer and how to use them.

## Text message types

In Composer, all messages that are sent to the user are defined in the Language Generation (LG) editor and follow the [.lg file format](#). For additional information about language generation in Composer, refer to the [language generation](#) article.

The table below lists the different types of text messages you can use in Composer.

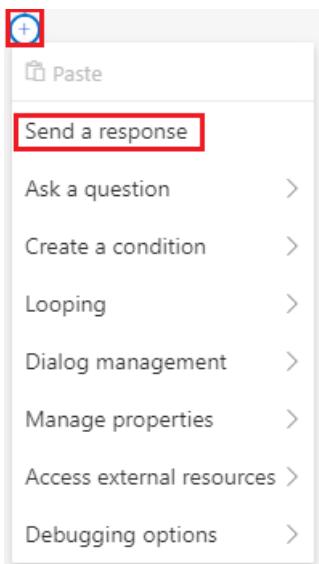
| MESSAGE TYPE      | DESCRIPTION                                                                                            |
|-------------------|--------------------------------------------------------------------------------------------------------|
| Simple text       | A simple LG defined to generate a simple text response.                                                |
| Text with memory  | An LG template that relies on a property to generate a text response.                                  |
| LG with parameter | An LG template that accepts a property as a parameter and uses that to generate a text response.       |
| LG composition    | An LG template composed with pre-defined templates to generate a text response.                        |
| Structured LG     | An LG template defined using <a href="#">structured response template</a> to generate a text response. |
| Multiline text    | An LG template defined with multiline response text.                                                   |
| If/Else           | An If/Else conditional template defined to generate text responses based on user's input.              |
| Switch            | A Switch conditional template defined to generate text responses based on user's input.                |

## The user scenario

When your bot receives messages from the user, all *intent* and *entity* values in the message are extracted and passed on to the dialog's event handler (trigger). In the trigger you can define actions the bot should take to respond to the user. Sending messages back to the user is one type of action you can define in the trigger.

Below is a screenshot of the **Send a response** action in Composer. How to get there:

1. Select + in the **Authoring canvas**.
2. Select **Send a response** from the action menu.

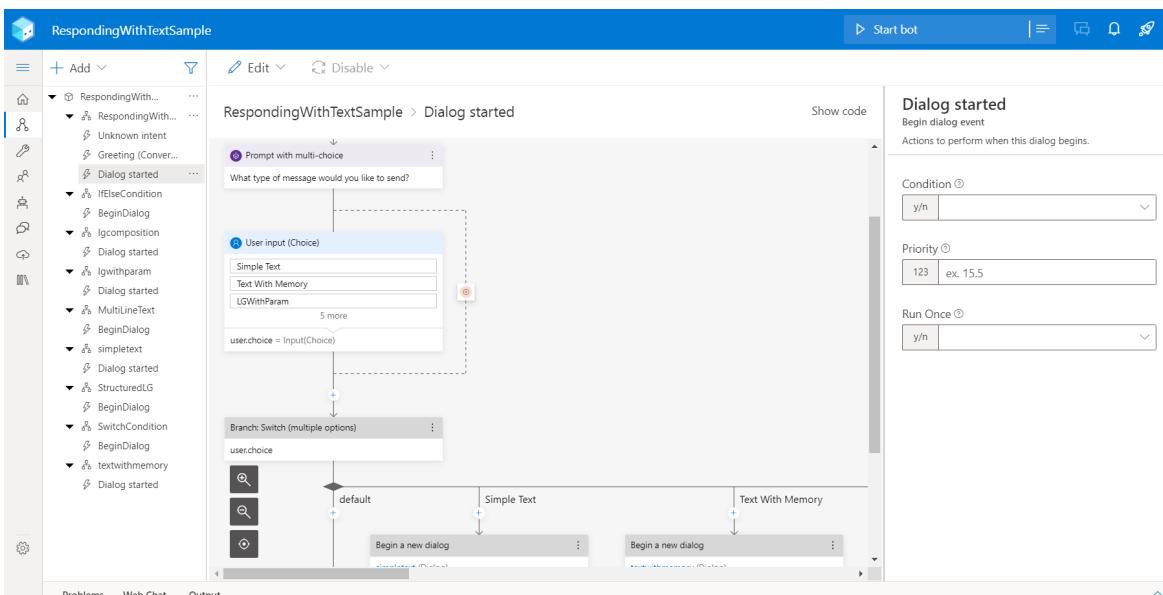


## The *Responding with Text* example

This section is an introduction to the **Responding with Text** example (sample bot) that is used in this article to explain how to send text messages in Composer.

- [Composer v2.x](#)
- [Composer v1.x](#)

1. Clone the [Bot Builder samples github repo](#) onto your machine.
2. Within the `composer-samples` folder you'll find C# and JavaScript projects, choose a language and navigate into the `projects` subfolder.
3. In this folder you'll find a `RespondingWithTextSample` project which you can open in Composer. Now that you have it loaded in Composer, you can click Start Bot and then [Open in Web Chat](#) to explore the sample interactively.
4. Now let's take a look to see how it works.
5. Select **Create** from the left side menu and then select the **Dialog started** trigger in the main dialog to get an idea of how this sample works.



6. Select **Bot Responses** from the Composer Menu. Then select **Common** to see the templates that are called when the user selects one of the items from the choices they are presented with when the **Prompt**

with multi-choice action executes. You will be referring to these templates throughout this article as each potential text message type is discussed in detail.

| Name               | Responses                                                                                             |
|--------------------|-------------------------------------------------------------------------------------------------------|
| #Greeting          | - nice to talk to you!                                                                                |
| #LGComposition     | - \${user.name} \${Greeting()}                                                                        |
| #LGWithParam       | - Hello \${user.name}, nice to talk to you!                                                           |
| #SimpleText        | - Hi, this is simple text<br>- Hey, this is simple text<br>- Hello, this is simple text               |
| #WelcomeUser       | -<br>... I can show you examples on sending messages. Restart the conversation to get started.<br>... |
| #greetInAWeek      | - SWITCH: \${dayOfWeek(utcNow())}<br>- CASE: \$()<br>- Happy Sunday!                                  |
| #timeOfDayGreeting | - IF: \${timeOfDay == 'morning'}<br>- good morning<br>- ELSEIF: \${timeOfDay == 'afternoon'}          |

## Text messages defined

Each of the sections below will detail how each type of text message is defined using the *simple response template* format in the .lg file that is exposed in Composer in the **Bot Responses** page. Each text message can also be defined in the **response editor** in the **Properties** panel when a **Send a response** action is selected.

### Simple text

To define a simple text message, use the hyphen (-) before the text that you want your bot to respond to users, for example:

```
- Here is a simple text message.
```

You can also define a simple text message with multiple variations. When you do this, the bot will respond randomly with any of the simple text messages, for example:

```
# SimpleText
- Hi, this is simple text
- Hey, this is simple text
- Hello, this is simple text
```

### Text with memory

This is how you would display a message to the user that is contained in a property that is stored in memory. This property can be defined programmatically as it is in the *Responding with Text* example, or can be set at run-time based on user input.

How to send a *text with memory* message:

1. Create a new action to send the response by selecting the "+" icon in the **Authoring canvas** and selecting **Send a response** from the list of actions.
2. In the **code editor**, enter the desired property. Note that all entries are preceded by a hyphen (-). In the example the following property is used: `- ${user.message}`.

**TIP**

- You reference a parameter using the syntax  `${user.message}` .
- You reference a template using the syntax  `${templateName()}` .

To learn more about setting properties in Composer, refer to the [Conversation flow and memory](#) article. To learn more about using expressions in your responses, refer to the [Adaptive expressions](#) article.

### LG with parameter

You can think of **LG with parameter** like a function with parameters, for example the template in the .lg file (entered in the code editor on the right or in the **Bot Responses** page on the left) looks like the following:

```
# LGWithParam(user)
- Hello ${user.name}, nice to talk to you!
```

In this LG template:

| ELEMENT                      | DESCRIPTION                                                                          |
|------------------------------|--------------------------------------------------------------------------------------|
| <code>LGWithParam()</code>   | The template name.                                                                   |
| <code>user</code>            | The object passed to the template as its parameter.                                  |
| <code> \${user.name} </code> | This is replaced with the value contained in the property <code> user.name </code> . |

### LG composition

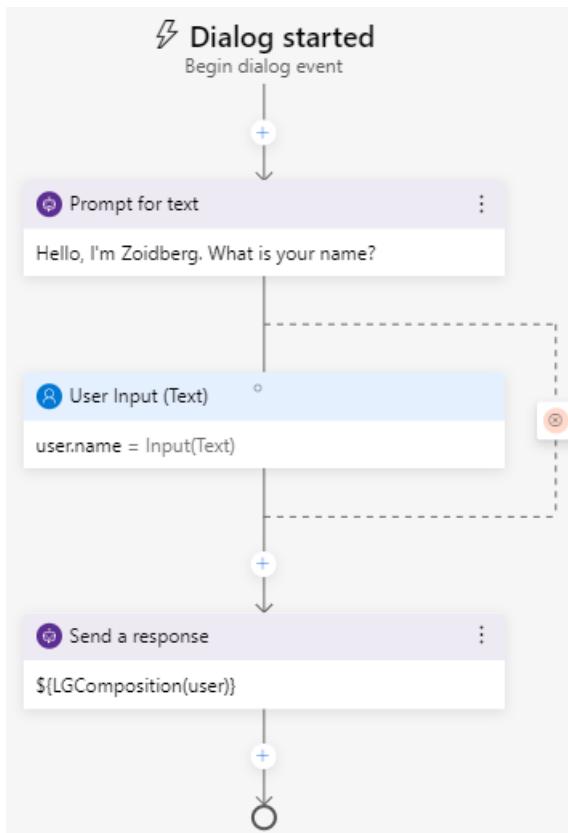
An **LG composition** message is a template composed of one or more existing LG templates. To define an **LG Composition** template you need to first define the component template(s) and then call them from your **LG composition** template. For example:

```
# Greeting
- nice to talk to you!

# LGComposition(user)
- ${user.name} ${Greeting()}
```

In this template  `# LGComposition(user)` , the  `# Greeting`  template is used to compose a portion of the new template. The syntax to include a pre-defined template is  `${templateName()}` .

The screenshot below is the **Dialog started** action in the **LGComposition** dialog in the *Responding with Text* example.



Note that the bot asks the user for input, saves the input in memory to `user.name`, and responds with a text message created from the `LGComposition` template.

## Structured LG

A **Structured LG** message uses the *structured response template* format. Structured response templates enable you to define complex structures such as cards.

For bot applications, the *structured response template* format natively supports:

- Activity definition. This is used by the **Structured LG** message.
- Card definition. See the [Sending responses with cards](#) article for more information.
- Any chatdown style constructs. For information on chatdown see the [chatdown](#) readme.

The [Responding with Text](#) example demonstrates using the Activity definition, for example:

```

# StructuredText
[Activity
  Text = text from structured
]

```

This is a simple structured LG with output response "text from structured". The definition of a structured template is as follows:

```

# TemplateName
> this is a comment
[Structure-name
  Property1 = <plain text> .or. <plain text with template reference> .or. <expression>
  Property2 = list of values are denoted via '|'. e.g. a | b
> this is a comment about this specific property
  Property3 = Nested structures are achieved through composition
]

```

- To learn more about *structured response templates*, you can refer to the [structured response template](#) article.

- To see how the *activity definition* is used in messages using cards, see the [AdaptiveCard](#) and [AllCards]](./how-to-send-cards.md#allcards) sections of the *Sending responses with cards* article.
- For a detailed explanation of the *activity definition* see the [Bot Framework -- Activity](#) readme on GitHub.

## Multiline text

If you need your response to contain multiple lines, you can include multi-line text enclosed in three accent characters: ```, for example:

```
# multilineText
- ``` you have such alarms
    alarm1: 7:am
    alarm2: 9:pm
```
```

### TIP

Multi-line variation can request template expansion and entity substitution by enclosing the requested operation in `{}$`. With multi-line support, you can have the language generation sub-system fully resolve a complex JSON or XML (e.g. SSML wrapped text to control bot's spoken reply).

## If/Else condition

Instead of using [conditional branching](#), you can define a conditional template to generate text responses based on user's input. For example:

```
# timeOfDayGreeting(timeOfDay)
- IF: ${timeOfDay == 'morning'}
    - good morning
- ELSEIF: ${timeOfDay == 'afternoon'}
    - good afternoon
- ELSE:
    - good evening
```

In this If/Else conditional template, bot will respond in text message `morning`, `afternoon` or `evening` based on user's input to match specific conditions defined in the template.

## Switch condition

The **Switch condition** template is similar to the **If/Else condition** template, you can define a **Switch condition** template to generate text messages in response to user's input, or based on a [prebuilt function](#) that requires no user interaction. For example, the *Responding with Text* example creates a template named **Switch condition** that calls the template `#greetInAWeek` that uses the [dayOfWeek](#) and [utcNow](#) functions:

```
# greetInAWeek
- SWITCH: ${dayOfWeek(utcNow())}
- CASE: ${0}
    - Happy Sunday!
-CASE: ${6}
    - Happy Saturday!
-DEFAULT:
    - Working day!
```

In this **Switch condition** template, the bot will respond with any of the following: `Happy Sunday!`, `Happy Saturday` or `Working day!` based on the value returned by the `{dayOfWeek(utcNow())}` functions. `utcNow()` is a pre-built function that returns the current timestamp as a string. `dayOfWeek()` is a function which returns the day of the week from a given timestamp. Read more about [prebuilt functions](#) in [Adaptive](#)

expressions.

## Further reading

- [Language generation](#)
- [.lg file format](#)
- [Structured response template](#)
- [Adaptive expressions](#)

## Next

- [Learn how to ask for user input.](#)

# Respond with cards

5/28/2021 • 12 minutes to read

**APPLIES TO:** Composer v1.x and v2.x

Cards enable you to create bots that can communicate with users in a variety of ways as opposed to simply using plain text messages. You can think of a card as an object with a standard set of rich user controls that you can choose from to communicate with and gather input from users. There are times when you need messages which simply consist of plain text and there are times when you need richer message content such as images, animated GIFs, video clips, audio clips and buttons. If you are looking for examples about sending text messages to users please read the [send text messages to users](#) article, if you need rich message content, cards offer several options which will be detailed in this article. If you are new to the concept of Cards, it might be helpful to read the *Cards* section of the [design the user experience](#) article.

## The structured response template

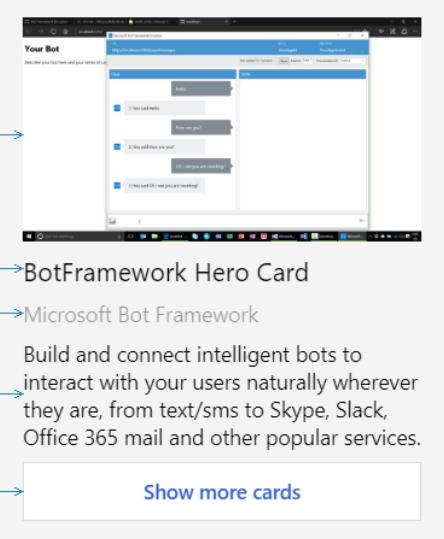
All of the **Bot Responses** are defined in a [.lg file](#) that is exposed in Composer in the **Bot Responses** page. Templates are the core element of a .lg file and there are three types of templates that can be used in .lg files: Simple response, conditional response and structured response templates. Cards are defined using the structured response template format.

A structured response template for cards consists of the following elements:

```
# TemplateName
[Card-name
    title = title of the card
    subtitle = subtitle of the card
    text = description of the card
    image = url of your image
    buttons = name of the button you want to show in the card
]
```

TEMPLATE COMPONENT	DESCRIPTION
# TemplateName	The template name. Always starts with "#". This is used when invoking the card.
[]	The entire template is contained within the square brackets.
Card-name	The name of the <a href="#">type</a> of card being referenced.
title	The title that will appear in the card when displayed to the user.
subtitle	The subtitle that will appear in the card when displayed to the user.
text	The text that will appear in the card when displayed to the user.

TEMPLATE COMPONENT	DESCRIPTION
image	The url pointing to the image that will appear in the card when displayed to the user.
buttons	Name of the button you want to show in the card.

The Card as seen in the bot	The Card as defined in the template
 <p>1 → </p> <p>2 → BotFramework Hero Card</p> <p>3 → Microsoft Bot Framework</p> <p>Build and connect intelligent bots to interact with your users naturally wherever they are, from text/sms to Skype, Slack, Office 365 mail and other popular services.</p> <p>4 → </p> <p>5 → <a href="#">Show more cards</a></p>	<pre># HeroCard [HeroCard] 2 → title = BotFramework Hero Card 3 → subtitle = Microsoft Bot Framework 4 → text = Build and connect intelligent bots to interact with your users naturally wherever they are, from text/sms to Skype, Slack, Office 365 mail and other popular services. 1 → image = https://sec.ch9.ms/ch9/7fff/e07cfef0-aa3b-40bb-9baa-7c9ef8ff7fff/buildr 5 → buttons = \${cardActionTemplate('imBack', 'Show more cards', 'Show more cards')} ]</pre>

## Additional resources for structured response templates

- The [structured response template](#) section of the [.lg file format](#) article.
- For more information on language generation in general, you can refer to the [language generation](#) concept article.

## Card types

Composer currently supports the following Card types:

CARD TYPE	DESCRIPTION
Hero Card	A card that typically contains a single large image, one or more buttons, and simple text.
Thumbnail Card	A card that typically contains a single thumbnail image, one or more buttons, and simple text.
Signin Card	A card that enables a bot to request that a user sign-in. It typically contains text and one or more buttons that the user can click to initiate the sign-in process.
Animation Card	A card that can play animated GIFs or short videos.
Video Card	A card that can play a video file.
Audio Card	A card that can play an audio file.
Adaptive Card	A customizable card that can contain any combination of text, speech, images, buttons, and input fields.

CARD TYPE	DESCRIPTION
All Card	To display all cards.

## The *Responding with Cards* example

This section is an introduction to the **Responding with Cards** example (sample bot) that is used in this article to explain how to incorporate cards into your bot using Composer.

Do the following to get the **Responding with Cards** example running in Composer:

- [Composer v2.x](#)
- [Composer v1.x](#)

1. Clone the [Bot Builder samples github repo](#) onto your machine.
2. Within the `composer-samples` folder you'll find C# and Javascript projects, choose a language and navigate into the `projects` subfolder.
3. In this folder you'll find a `RespondingWithCardsSample` project which you can open in Composer. Now that you have it loaded in Composer, you can click Start Bot and then [Open in Web Chat](#) to explore the sample interactively.
4. Now let's take a look to see how it works.
5. Select **Create** from the Composer Menu.
6. Select the **Unknown intent** trigger in the main dialog to get an idea of how this sample works.

The screenshot shows the Microsoft Bot Composer interface for the 'RespondingWithCardsSample' bot. The left sidebar shows the project structure with 'RespondingWithCardsSample' expanded, revealing 'Unknown intent' and 'Greeting' triggers. The main workspace displays the 'Unknown intent' trigger. It consists of the following sequence of actions:

- Condition:** Unknown intent recognized
- Action:** User Input (Choice)
  - Choices: HeroCard, HeroCardWithMemory, ThumbnailCard
  - Variable: userchoice = Input(Choice)
- Action:** Branch: Switch (multiple options)
  - Branches:
    - default: Send an Activity - {HeroCard[]}
    - HeroCard: Send an Activity - {HeroCard[]}
    - HeroCardWithMemory: Send an Activity - {HeroCardWithMemory[]}

On the right side, there is a detailed description of the trigger:

**Unknown intent**  
Unknown intent recognized  
Action to perform when user input is unrecognized or if none of the 'on intent recognition' triggers match recognized intent.

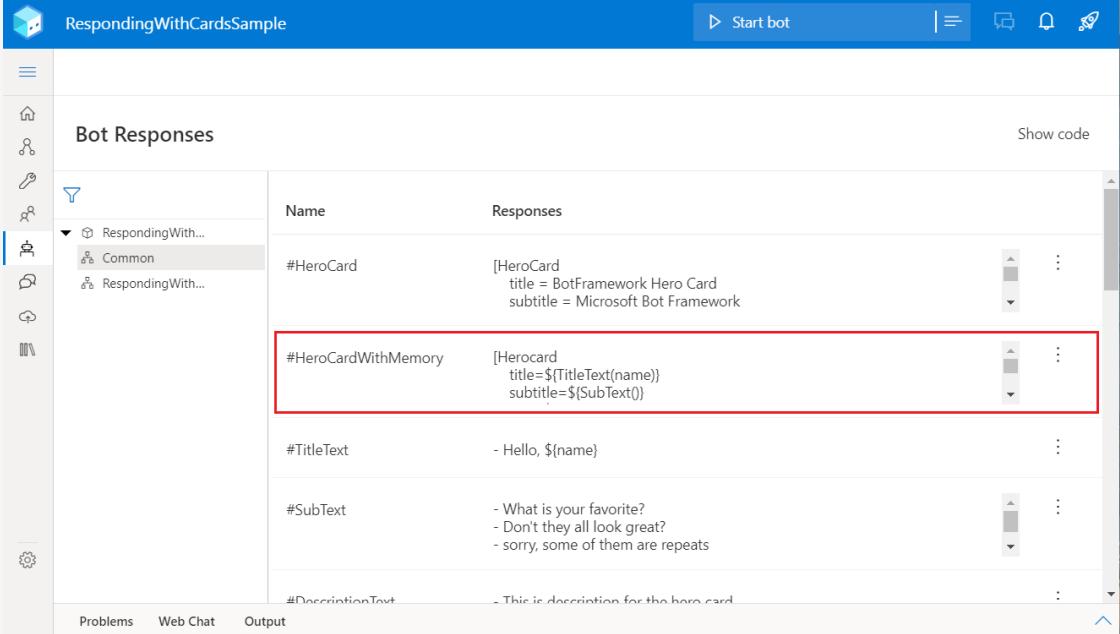
Configuration settings for the trigger include:

- Condition: y/n
- Priority: 123 ex. 15.5
- Run Once: y/n

## NOTE

In this sample, the **Unknown intent** trigger contains a **Prompt with multi-choice** action (from the **Ask a question** menu) where the **User Input** list style is set to **list**, and the user's selection is stored into the `user.choice` property. The `user.choice` property is passed to the next action, which is a **Branch: Switch (multiple options)** action (from the **Create a condition** menu). The item that the user selects from the list will determine which flow is taken.

For example, if `HeroCardWithMemory` is selected, `HeroCardWithMemory()` is called, which calls the `HeroCardWithMemory` template in the .lg file that can be found by selecting **Bot Responses** from the Composer Menu as shown in the following image.

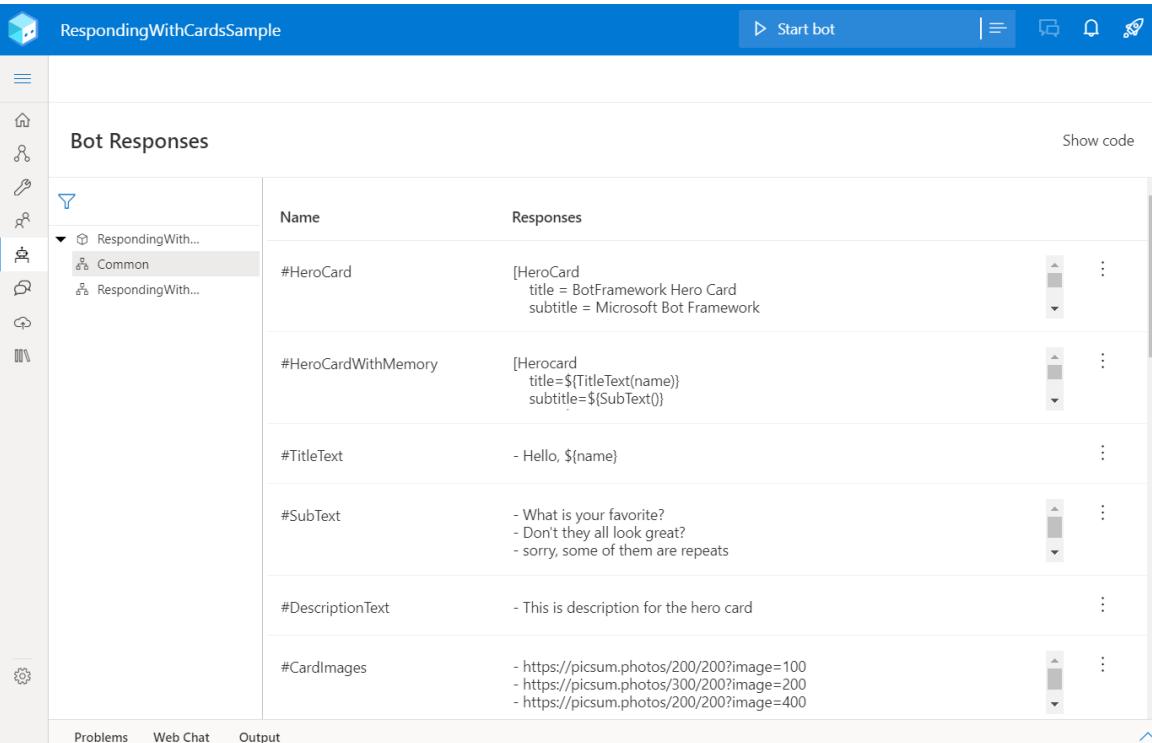


The screenshot shows the Microsoft Bot Framework Composer interface. The left sidebar has icons for Home, Run, Chat, and Bot. The main area is titled "Bot Responses". A tree view on the left shows a node "RespondingWith..." expanded, with "Common" and "RespondingWith..." children. The main table has columns "Name" and "Responses". The rows are:

Name	Responses
#HeroCard	[HeroCard title = BotFramework Hero Card subtitle = Microsoft Bot Framework]
#HeroCardWithMemory	[Herocard title=\${TitleText(name)} subtitle=\${SubText()}]
#TitleText	- Hello, \${name}
#SubText	- What is your favorite? - Don't they all look great? - sorry, some of them are repeats
#DescriptionText	- This is description for the hero card

At the bottom, there are tabs for "Problems", "Web Chat", and "Output".

7. Select **Bot Responses** from the Composer Menu to see the templates that are called when the user selects one of the items from the choices they are presented with when the **Prompt with multi-choice** action executes. You will be referring to these templates throughout this article as each card is discussed in detail.



The screenshot shows the Microsoft Bot Framework Composer interface. The left sidebar has icons for Home, Run, Chat, and Bot. The main area is titled "Bot Responses". A tree view on the left shows a node "RespondingWith..." expanded, with "Common" and "RespondingWith..." children. The main table has columns "Name" and "Responses". The rows are:

Name	Responses
#HeroCard	[HeroCard title = BotFramework Hero Card subtitle = Microsoft Bot Framework]
#HeroCardWithMemory	[Herocard title=\${TitleText(name)} subtitle=\${SubText()}]
#TitleText	- Hello, \${name}
#SubText	- What is your favorite? - Don't they all look great? - sorry, some of them are repeats
#DescriptionText	- This is description for the hero card
#CardImages	- <a href="https://picsum.photos/200/200?image=100">https://picsum.photos/200/200?image=100</a> - <a href="https://picsum.photos/300/200?image=200">https://picsum.photos/300/200?image=200</a> - <a href="https://picsum.photos/200/200?image=400">https://picsum.photos/200/200?image=400</a>

At the bottom, there are tabs for "Problems", "Web Chat", and "Output".

# Define rich cards

Each of the sections below will detail how each card is defined as a structured response template in the .lg file that is exposed in Composer in the **Bot Responses** page.

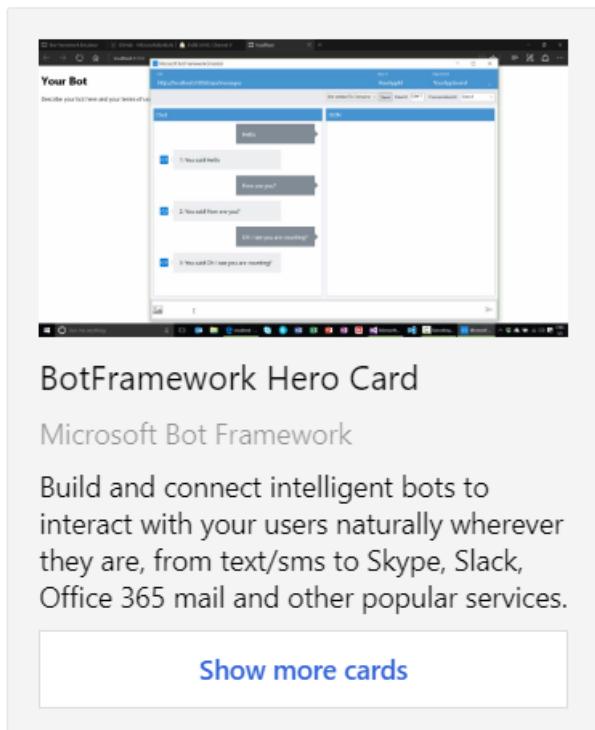
## NOTE

LG provides some variability in card definition, which will eventually be converted to be aligned with the [SDK card definition](#). For example, both `image` and `images` fields are supported in all the card definitions in LG even though only `images` are supported in the SDK card definition. For HeroCard and Thumbnail cards in LG, the values defined in either `image` or `images` field will be converted to an images list. For the other types of cards, the last defined value will be assigned to the `image` field. The values you assign to the `image/images` field can be in one of the following formats: string, [adaptive expression](#), or array in the format using `|`. Read more [here](#).

## HeroCard

A *hero card* is a basic card type that allows you to combine images, text and interactive elements such as buttons in one object and present a mixture of them to the user. A hero card is defined using a structured template as follows:

```
# HeroCard
[HeroCard
    title = BotFramework Hero Card
    subtitle = Microsoft Bot Framework
    text = Build and connect intelligent bots to interact with your users naturally wherever they are, from
text/sms to Skype, Slack, Office 365 mail and other popular services.
    image = https://sec.ch9.ms/ch9/7ff5/e07cfef0-aa3b-40bb-9baa-
7c9ef8ff7ff5/buildreactionbotframework_960.jpg
    buttons = ${cardActionTemplate('imBack', 'Show more cards', 'Show more cards')}
]
```



This example of hero card will enable your bot to send an image from a designated URL back to users when an event to send a hero card is triggered. The hero card will include a button to show more cards when pressed.

## HeroCardWithMemory

A HeroCardWithMemory is a hero card that demonstrates how to call simple response templates in the .lg file

just as you would call a function.

```
# HeroCardWithMemory(name)
[Herocard
  title=${TitleText(name)}
  subtitle=${SubText()}
  text=${DescriptionText()}
  images=${CardImages()}
  buttons=${cardActionTemplate('imBack', 'Show more cards', 'Show more cards')}
]
```

If you look in the **Bot Responses** page you will see where the values come from that populate the hero card:

```
# HeroCardWithMemory(name)
[Herocard
  1 → title= ${TitleText(name)}
  2 → subtitle= ${SubText()}
  3 → text= ${DescriptionText()}
  4 → images= ${CardImages()}
  5 → buttons= ${cardActionTemplate('imBack', 'Show more cards', 'Show more cards')}
]

  1 # TitleText(name)
    - Hello, ${name}
  2 # SubText
    - What is your favorite?
    - Don't they all look great?
    - sorry, some of them are repeats
  3 # DescriptionText
    - This is description for the hero card
  4 # CardImages
    - https://picsum.photos/200/200?image=100
    - https://picsum.photos/300/200?image=200
    - https://picsum.photos/200/200?image=400
  5 # cardActionTemplate(type, title, value)
[CardAction
  Type = ${if(type == null, 'imBack', type)}
  Title = ${title}
  Value = ${value}
  Text = ${title}
]
```

What is your name?

Frank



1 → Hello, Frank

2 → sorry, some of them are repeats

3 → This is description for the hero card

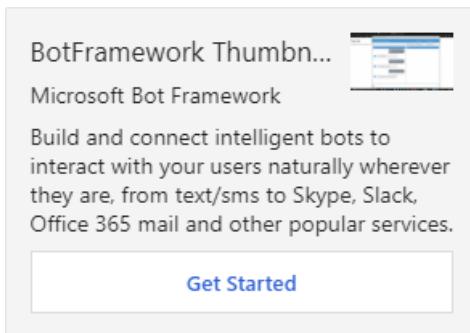
4 → Show more cards

Type your message  Send

## ThumbnailCard

A thumbnail card is another type of basic card that combines a mixture of images, text and buttons. Unlike hero cards, which present designated images in a large banner, thumbnail cards present images as thumbnail. They typically contain a single thumbnail image, one or more buttons, and simple text. A thumbnail card is defined using a structured template as follows:

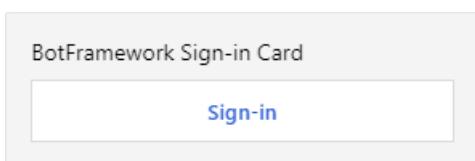
```
# ThumbnailCard
[ThumbnailCard
  title = BotFramework Thumbnail Card
  subtitle = Microsoft Bot Framework
  text = Build and connect intelligent bots to interact with your users naturally wherever they are, from
text/sms to Skype, Slack, Office 365 mail and other popular services.
  image = https://sec.ch9.ms/ch9/7ff5/e07cfef0-aa3b-40bb-9baa-
7c9ef8ff7ff5/buildreactionbotframework_960.jpg
  buttons = Get Started
]
```



## SigninCard

A sign-in card is a card that enables a bot to request that a user sign in. A sign-in card is defined using a structured template as follows:

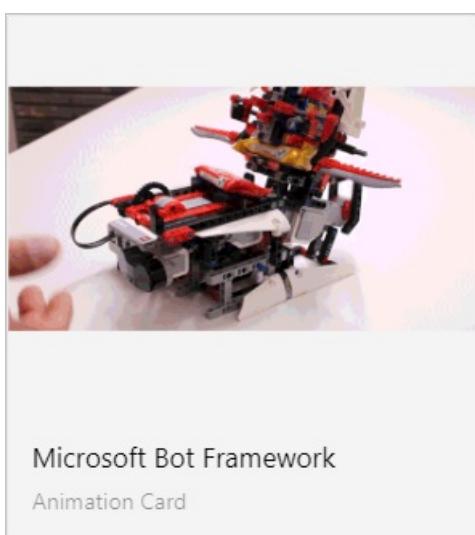
```
# SigninCard
[SigninCard
    text = BotFramework Sign-in Card
    buttons = ${cardActionTemplate('signin', 'Sign-in', 'https://login.microsoftonline.com/'})
]
```



## AnimationCard

Animation cards contain animated image content (such as .gif files). Typically this content does not contain sound, and is usually presented with minimal transport controls (like pause or play), or no transport controls at all. Animation cards follow all shared rules defined for transport controls (like rewind, restart, pause, and play). Video cards follow all shared rules defined for [Media cards](#). An animation card is defined using structured template as follows:

```
# AnimationCard
[AnimationCard
    title = Microsoft Bot Framework
    subtitle = Animation Card
    image = https://docs.microsoft.com/en-us/bot-framework/media/v1x/how-it-works/architecture-resize.png
    media = http://i.giphy.com/Ki55RUbOV5njy.gif
]
```



## VideoCard

Video cards contain video content in video format such as `.mp4`. Typically this content is presented to the user with advanced transport controls (like rewind, restart, pause, and play). Video cards follow all shared rules defined for [Media cards](#). A video card is defined using structured template as follows:

```
# VideoCard
[VideoCard
    title = Big Buck Bunny
    subtitle = by the Blender Institute
    text = Big Buck Bunny (code-named Peach) is a short computer-animated comedy film by the Blender Institute
    image = https://upload.wikimedia.org/wikipedia/commons/thumb/c/c5/Big_buck_bunny_poster_big.jpg/220px-Big_buck_bunny_poster_big.jpg
    media = http://download.blender.org/peach/bigbuckbunny_movies/BigBuckBunny_320x180.mp4
    buttons = Learn More
]
```



### Big Buck Bunny

by the Blender Institute

Big Buck Bunny (code-named Peach) is a short computer-animated comedy film by the Blender Institute

[Learn More](#)

## AudioCard

Audio cards contain audio content in audio formats such as `.mp3` and `.wav`. Audio cards follow all shared rules defined for [Media cards](#). An audio card is defined using a structured template as follows:

```
# AudioCard
[AudioCard
    title = I am your father
    subtitle = Star Wars: Episode V - The Empire Strikes Back
    text = The Empire Strikes Back (also known as Star Wars: Episode V - The Empire Strikes Back)
    image = https://upload.wikimedia.org/wikipedia/en/3/3c/SW_-_Empire_Strikes_Back.jpg
    media = http://www.wavlist.com/movies/004/father.wav
    buttons = Read More
]
```



## AdaptiveCard

[Adaptive Cards](#) are a card exchange format adopted by Composer that let developers define their card content in a common and consistent way using JSON. Once defined, the Adaptive Card can be used in any supported channel, automatically adapting to the look and feel of the host.

Adaptive Cards not only support custom text formatting; they also support the use of containers, speech, images, buttons, customizable backgrounds, user input controls for dates, numbers, text, and even customizable drop-down lists.

An Adaptive Card is defined as follows:

```
# AdaptiveCard
[Activity
    Attachments = ${json(adaptivecardjson())}
]
```

This tells Composer that it's referencing a template named **adaptivecardjson** that is in the JSON format. If you look in the **Bot Responses**, you will see that template. The following is the template used to generate the Adaptive Card:

```
{
    "$schema": "http://adaptivecards.io/schemas/adaptive-card.json",
    "version": "1.0",
    "type": "AdaptiveCard",
    "speak": "Your flight is confirmed for you and 3 other passengers from San Francisco to Amsterdam on Friday, October 10 8:30 AM",
    "body": [
        {
            "type": "TextBlock",
            "text": "Passengers",
            "weight": "bolder",
            "isSubtle": false
        },
        {
            "type": "TextBlock",
            "text": "${PassengerName()}",
            "separator": true
        },
        {
            "type": "TextBlock",
            "text": "${PassengerName()}",
            "spacing": "none"
        },
        {
            "type": "TextBlock",
            "text": "${PassengerName()}",
            "spacing": "none"
        }
    ]
}
```

```
  },
  {
    "type": "TextBlock",
    "text": "2 Stops",
    "weight": "bolder",
    "spacing": "medium"
  },
  {
    "type": "TextBlock",
    "text": "Fri, October 10 8:30 AM",
    "weight": "bolder",
    "spacing": "none"
  },
  {
    "type": "ColumnSet",
    "separator": true,
    "columns": [
      {
        "type": "Column",
        "width": 1,
        "items": [
          {
            "type": "TextBlock",
            "text": "San Francisco",
            "isSubtle": true
          },
          {
            "type": "TextBlock",
            "size": "extraLarge",
            "color": "accent",
            "text": "SFO",
            "spacing": "none"
          }
        ]
      },
      {
        "type": "Column",
        "width": "auto",
        "items": [
          {
            "type": "TextBlock",
            "text": " "
          },
          {
            "type": "Image",
            "url": "http://adaptivecards.io/content/airplane.png",
            "size": "small",
            "spacing": "none"
          }
        ]
      },
      {
        "type": "Column",
        "width": 1,
        "items": [
          {
            "type": "TextBlock",
            "horizontalAlignment": "right",
            "text": "Amsterdam",
            "isSubtle": true
          },
          {
            "type": "TextBlock",
            "horizontalAlignment": "right",
            "size": "extraLarge",
            "color": "accent",
            "text": "AMS",
            "spacing": "none"
          }
        ]
      }
    ]
  }
]
```

```
        ]
    },
    {
        "type": "TextBlock",
        "text": "Non-Stop",
        "weight": "bolder",
        "spacing": "medium"
    },
    {
        "type": "TextBlock",
        "text": "Fri, October 18 9:50 PM",
        "weight": "bolder",
        "spacing": "none"
    },
    {
        "type": "ColumnSet",
        "separator": true,
        "columns": [
            {
                "type": "Column",
                "width": 1,
                "items": [
                    {
                        "type": "TextBlock",
                        "text": "Amsterdam",
                        "isSubtle": true
                    },
                    {
                        "type": "TextBlock",
                        "size": "extraLarge",
                        "color": "accent",
                        "text": "AMS",
                        "spacing": "none"
                    }
                ]
            },
            {
                "type": "Column",
                "width": "auto",
                "items": [
                    {
                        "type": "TextBlock",
                        "text": " "
                    },
                    {
                        "type": "Image",
                        "url": "http://adaptivecards.io/content/airplane.png",
                        "size": "small",
                        "spacing": "none"
                    }
                ]
            },
            {
                "type": "Column",
                "width": 1,
                "items": [
                    {
                        "type": "TextBlock",
                        "horizontalAlignment": "right",
                        "text": "San Francisco",
                        "isSubtle": true
                    },
                    {
                        "type": "TextBlock",
                        "horizontalAlignment": "right",
                        "size": "extraLarge",
                        "color": "accent",
                        "+text": "SEO"
                    }
                ]
            }
        ]
    }
}
```

```

        "text": "SFO",
        "spacing": "none"
    }
]
}
]
},
{
    "type": "ColumnSet",
    "spacing": "medium",
    "columns": [
        {
            "type": "Column",
            "width": "1",
            "items": [
                {
                    "type": "TextBlock",
                    "text": "Total",
                    "size": "medium",
                    "isSubtle": true
                }
            ]
        },
        {
            "type": "Column",
            "width": 1,
            "items": [
                {
                    "type": "TextBlock",
                    "horizontalAlignment": "right",
                    "text": "$4,032.54",
                    "size": "medium",
                    "weight": "bolder"
                }
            ]
        }
    ]
}
]
}
}

```

**Passengers**

Tom  
Tom  
Tom

2 Stops  
Fri, October 10 8:30 AM

San Francisco		Amsterdam
<b>SFO</b>		<b>AMS</b>

Non-Stop  
Fri, October 18 9:50 PM

Amsterdam		San Francisco
<b>AMS</b>		<b>SFO</b>

Total **\$4,032.54**

#### AdaptiveCard References

- [Adaptive Cards overview](#)
- [Adaptive Cards Sample](#)
- [Adaptive Cards for bot developers](#)

## AllCards

The #AllCards template displays all of the cards as `Attachments` of the `Activity` object.

```
# AllCards
[Activity
    Attachments = ${HeroCard()} | ${ThumbnailCard()} | ${SigninCard()} | ${AnimationCard()} |
${VideoCard()} | ${AudioCard()} | ${AdaptiveCard()}
    AttachmentLayout = ${AttachmentLayoutType()}
]
```

## Further reading

- [Bot Framework - Cards](#)
- [Add media to messages](#)
- [Language generation](#)
- [Structured response template](#)

## Next

- Learn [how to define triggers and events](#).

# Ask for user input

6/11/2021 • 14 minutes to read

**APPLIES TO:** Composer v1.x and v2.x

Bot Framework Composer makes it easy to collect and validate a variety of data types, and handle instances when users input is invalid or unrecognized data.

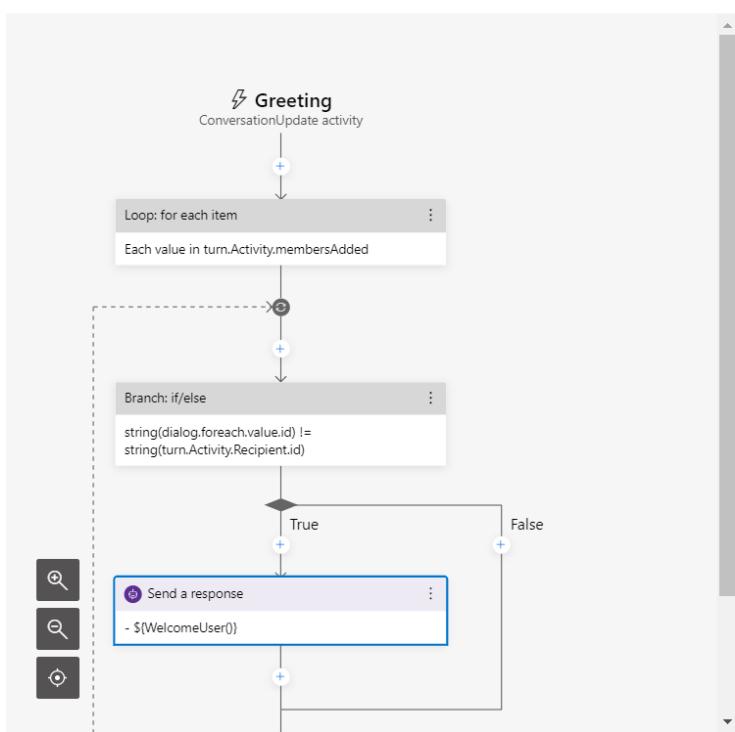
## The *Asking Questions* example

This section is an introduction to the **Asking Questions** example (sample bot) that is used in this article to explain how to incorporate prompts for user input into your bot using Composer.

- [Composer v2.x](#)
- [Composer v1.x](#)

Do the following to get the **Asking Questions** example running in Composer:

1. Clone the [Bot Builder samples github repo](#) onto your machine.
2. Within the `composer-samples` folder you'll find C# and Javascript projects, choose a language and navigate into the `projects` subfolder.
3. In this folder you'll find a `AskingQuestionsSample` project which you can open in Composer. Now that you have it loaded in Composer, you can click Start Bot and then [Open in Web Chat](#) to explore the sample interactively.
4. Now let's take a look to see how it works.
5. Select **Create** from the Composer Menu.
6. Select the **Greeting** trigger in the main dialog to get an idea of how this sample works.



## Send a response

Send Activity

Respond with an activity.

[Learn more](#)

Bot responses ⓘ

[Show response editor](#)

```

1 - ${WelcomeUser()}
  
```

Template name: #SendActivity\_xh61dm()

7. In this sample, the **Greeting** trigger always is the first thing that runs when the bot starts. This trigger executes the **Send a response** action. The **Send a response** action calls the *WelcomeUser* template: `#{WelcomeUser()}`. To see what the *WelcomeUser* template does, select **Bot Responses** from the **Composer Menu** and search for `#WelcomeUser` in the **Name** column.

Name	Responses
#WelcomeUser	-“Welcome to Input Sample Bot. I can show you examples on how to use actions. You can enter number 01-07 01 - TextInput
#ShowImage	[HeroCard title = Here is the attachment image = \${contentUrl}]
#Welcome	-Welcome to input samples.
<a href="#">New template</a>	

### IMPORTANT

When the bot first starts, it executes the **greeting** trigger. The **Send a response** action associated with the **greeting** trigger starts to execute and calls the `#{WelcomeUser()}` template where different options are defined and presented to the user. When the user responds by typing in the name or number of the item they wish to select, the bot searches the user input for known patterns. When a pattern is found the bot sends an event with the corresponding intent. That intent is captured by the **Intent recognized** trigger that was created to handle that intent. For example, if the user enters '01' or 'TextInput' the **TextInput** trigger handles that event and calls the **TextInput** dialog. The remaining steps in this section will walk you through this process.

8. Select the main dialog and look at the **Properties** pane. Note that the **Recognizer Type** is set to **Regular expression recognizer**.
9. Navigate to the **Common** dialog under **Bot Responses** to see the template used to create a menu of options. Because the **Regular expression recognizer** is selected, users can either respond with the name of the input type or the number to test the associated example.

The screenshot shows the Microsoft Bot Framework Composer interface. The left sidebar has 'Bot responses' selected. The main area shows a tree view of input types under 'AskingQuestionsSample'. A code editor window displays a snippet of bot logic. Below the code editor are tabs for 'Problems', 'Web Chat', and 'Output'.

For example, if user a responds with *05* or *AttachmentInput*, the bot will start the attachment input example.

In each of the following sections you will learn how to create each of these user input types, using the corresponding dialog as an example.

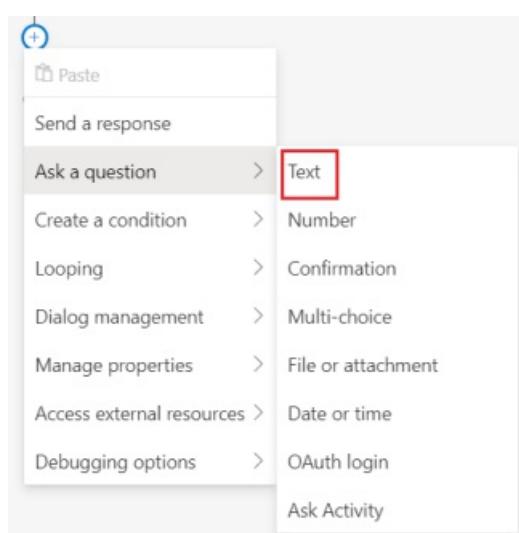
## Text input

The **Text input** prompts users for their name then responds with a greeting using the name provided. This is demonstrated in the *Asking Questions* example in the **TextInput** dialog. You create a text input prompt by selecting the + icon in the **Authoring canvas** then selecting **Text input** from the **Ask a question** menu. Optionally, the next section details how to create an entire text input dialog or you can go directly to the [Number Input](#) section.

### Create a text input action

To create a text input action:

1. Select the + icon then select **Text** from the **Ask a question** menu.



2. Enter **Hello, I'm Zoidberg. What is your name?** (This can't be interrupted) into the **Prompt for text** field in the **Properties** panel.

**Prompt for text**

Text Input

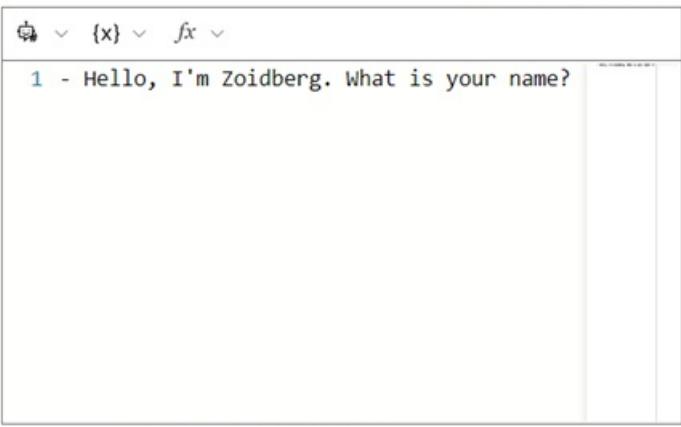
Collection information - Ask for a word or sentence.

[Learn more](#)

---

**Bot Asks**    **User Input**    **Other**

Prompt for text ⓘ    [Switch to response editor](#)



Template name: #TextInput\_Prompt\_0778480

3. Select the **User Input** tab, then enter `user.name` into the **Property** field

**Prompt for text**

Text Input

Collection information - Ask for a word or sentence.

[Learn more](#)

---

**Bot Asks**    **User Input**    **Other**

Property ⓘ    `abc user.name`

Output format ⓘ    `abc ex. =toUpperCase(this.value), ${toUpperCase(this.value)}`

Value ⓘ    `abc ex. hello world, Hello ${user.name}, user.lastN...`

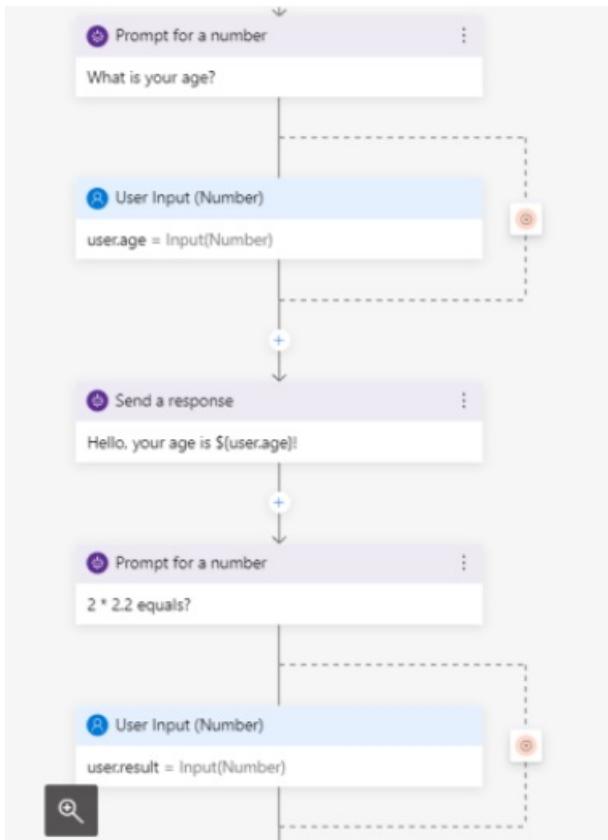
4. Create a new action by selecting the + icon in the **Authoring canvas** then select **Send a response** from the list of actions.

5. Enter `Hello ${user.name}, nice to talk to you!` into the **response editor** in the **Properties** panel.

## Number input

The **NumberInput** example prompts the user for their age and other numerical values using the **Number input** action.

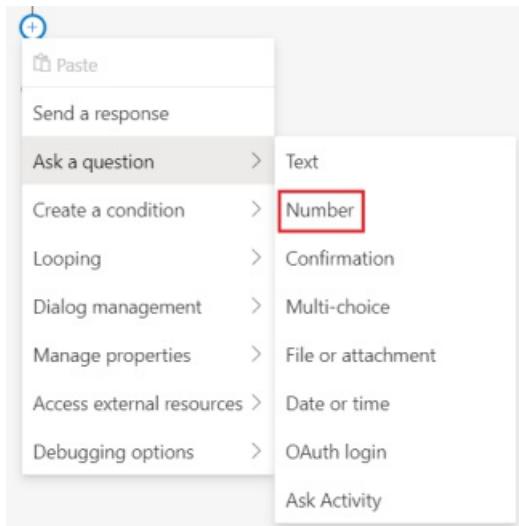
As seen in the **NumberInput** dialog the user is prompted for two numbers: their age stored as `user.age` and the result of `2*2.2` stored as a `user.result`. When using number prompts you can set the **Output format** to either `float` or `integer`.



### Create a number input action

To create a number input action:

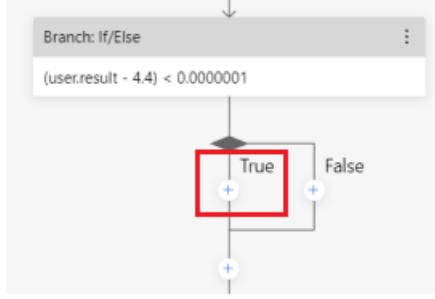
1. Select the + icon in the **Authoring canvas**. When the list of actions appear, select **Number** from the **Ask a question** menu.



- In the **Bot Asks** tab of the **Properties** panel, enter - **What is your age?**
  - Select the **User Input** tab then enter `user.age` into the **Property** field.
  - In the **Output format** field enter `int(this.value)`. The default is `float(this.value)`.
  - Select the **Other** tab, click **Validation**. Enter - **Please input a number.** into the **Invalid Prompt** field.
2. Create another action by selecting the + icon in the **Authoring panel** and selecting **Send a response** and enter - **Hello, your age is \${user.age}!** into the prompt field. This will cause the bot to respond back to the user with their age.
  3. Next, follow step 1 in this section to create another **Number Input** action.

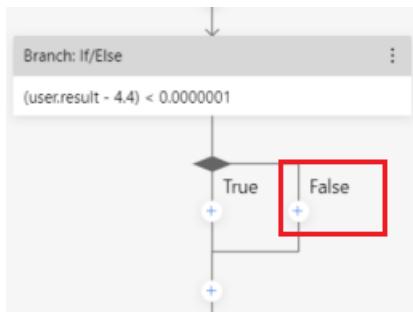
- In the **Bot Asks** tab of the **Properties** panel, enter `- 2 * 2.2 equals?`
  - Select the **User Input** tab then enter `user.result` into the **Property** field.
  - Select the **Other** tab and click **Validation**. Enter `Please input a number.` into the **Invalid Prompt** field.
4. Create a **Branch: If/else** action by selecting the **+** icon in the **Authoring canvas** and selecting **Branch: If/else** from the **Create a condition** menu. In the properties panel, specify the condition and follow the [Adaptive expressions](#) syntax. The if-else action creates two branches. If the expression evaluates to `true`, the actions on the **True** branch are processed; otherwise, the actions on the **False** branch are processed. After the appropriate branch completes, processing continues with the next action after the **Branch: If/else** action.

5. Select the **+** icon in the *true* branch and select **Send a response**.



6. Enter `-2 * 2.2 equals ${user.result}, that's right!` into the prompt field. This will cause the bot to respond back to the user with "2 \* 2.2 equals 4.4, that's right!".

7. Create a conditional action by selecting the **+** icon in the *false* branch. This will execute when the user enters an invalid answer.



8. Create another action by selecting the **+** icon in the **Authoring panel** and selecting **Send a response**.

9. Enter `-2 * 2.2 equals ${user.result}, that's wrong!` into the prompt field.

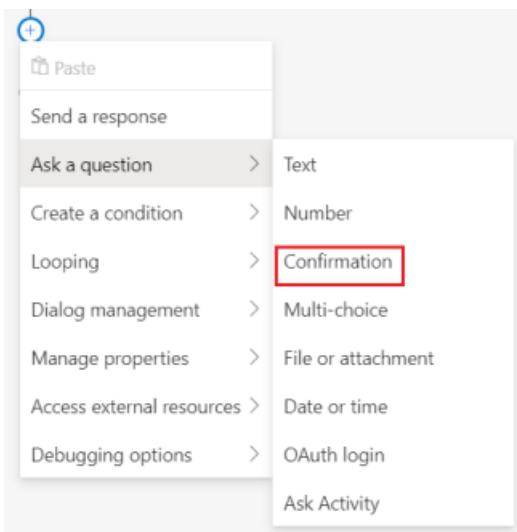
## Confirmation input

**Confirmation prompts** are useful after you've asked the user a question and want to confirm their answer. Unlike the **Multiple choice** action that enables your bot to present the user with a list to choose from, confirmation prompts ask the user to make a binary (yes/no) decision.

### Create a confirmation action

To create a confirmation action:

1. Select the **+** icon then select **Confirmation** from the **Ask a question** menu.



2. Enter **-Would you like ice cream?** in the **Prompt** field of the **Properties** panel.

3. Switch to the **User Input** tab.

**TIP**

You can also switch to the **User Input** tab by selecting the **User answers** action in the **Authoring canvas**.

4. Enter `user.confirmed` in the **Property** field.

5. Select the **+** icon in the **Authoring panel** and select **Send a response**.

6. Enter `-confirmation: ${user.confirmed}` into the prompt field.

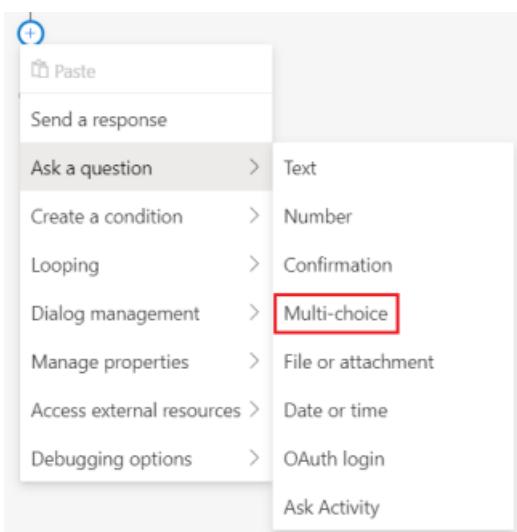
## Multiple choice input

Multiple choice enables you to present your users with a list of options to choose from.

### Create a multiple choice action

To create a prompt with a list of options that the user can choose from:

1. Select the **+** icon then select **Multi-choice** from the **Ask a question** menu.



2. Select **Bot Asks** tab and enter **- Please select a value from below:** in the **Prompt with multi-choice** field.

3. Switch to the **User Input** tab.

- Enter `user.style` in the **Property** field.
  - Scroll down to the **Array of choices** section and select one of the three options (**simple choices**, **structured choices**, **expression**) to add your choices.
4. To define simple multi-choices, select **simple choices**, you can add the choices one at a time in the **Array of choices** field. Every time you add a choice option, make sure you press Enter. For example:

- Test1
- Test2
- Test3

5. To add additional data or value to the choice, select **structured choices**, then select **Add** and add the choices one at a time with the following properties.

PROPERTY	DESCRIPTION	EXAMPLE
Choice name	The name of the choice and the value to return when this choice is selected.	<code>Test01</code>
Synonym (Optional)	A list of synonyms to recognize in addition to the <code>Choice name</code> value.	<code>test1, test01</code>
Action	Card action to define for this choice.	See the example below. Read more in the <a href="#">botframework-schema CardAction reference doc</a> .

The following is an example of how to define an **Action** in JSON format:

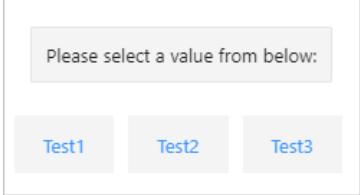
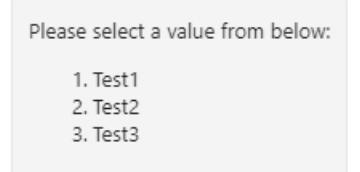
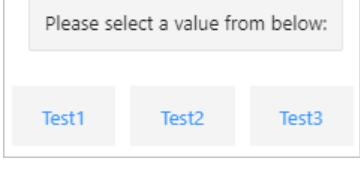
```
{
  "type": "messageBack",
  "title": "Title",
  "image": "https://sec.ch9.ms/ch9/7ff5/e07cfef0-aa3b-40bb-9baa-7c9ef8ff7ff5/buildreactionbotframework_960.jpg",
  "displayText": "display text",
  "text": "Text"
}
```

6. If you want to configure a dynamic multiple choice with an array or a variable, select **=Write an expression** and enter the property name of the initialized array like `=user.name`. This means any value from the `user.name` property will be returned dynamically.

#### Additional information: The User Input tab

- The **Output format** field is set to `value` by default. This means the value, not the index will be returned, for this example that means any one of these three values will be returned: 'test1', 'test2', 'test3'.
- By default the **locale** is set to `en-us`. The locale sets the language the recognizer should expect from the user (US English in this sample).
- By default the **List style** is set to `Auto`. The **List style** sets the style for how the choice options are displayed. The table below shows the differences in appearance for the three choices:

LIST STYLE	APPEARANCE	DESCRIPTION
None	Please select a value from below:	No options will be display.

LIST STYLE	APPEARANCE	DESCRIPTION
Auto		Composer decides the formatting, usually Suggested Action buttons.
Inline		Separates options using the value in the <b>Inline separator</b> field.
List		Displays options as list. A numbered list if <b>Include numbers</b> is selected.
SuggestedAction		Displays options as SuggestedAction buttons.
HeroCard		Displays Hero Card with options as buttons <i>within</i> card.

- There are three boxes related to inline separation, or how your bot separates the text of your choices:
  - **Inline separator** - character used to separate individual choices when there are more than two choices, usually `,`.
  - **Inline or** - separator used when there are only two choices, usually `or`.
  - **Inline or more** - separator between last two choices when there are more than two options, usually `, or`.
- The **Include numbers** option allows you to use plain or numbered lists when the **List Style** is set to **List**.

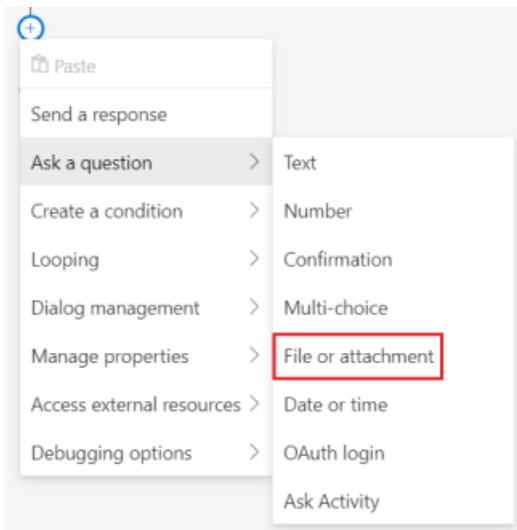
## File or attachment input

The **Attachment Input** example demonstrates how to enable users to upload images, videos, and other media. When running this example bot in the Emulator, once this option is selected from the main menu you will be prompted to "Please send an image.", select the paperclip icon next to the text input area and select an image file.

### Create an attachment input action

To implement an Attachment Input action:

1. Select the + icon then select **File or attachment** from the **Ask a question** menu.



2. Enter - **Please send an image.** in the **Prompt** field of the **Properties** panel.

3. Switch to the **User Input** tab ()

- Enter `dialog.attachments` in the **Property** field.
- Enter `a11` in the **Output format** field.

**TIP**

You can set the **Output format** to `first` (only the first attachment will be output even if multiple were selected) or `a11` (all attachments will be output when multiple were selected).

4. Select the + icon in the **Authoring panel** and select **Send a response**.

5. Enter `-${ShowImage(dialog.attachments[0].contentUrl, dialog.attachments[0].contentType)}` into the **Prompt** field.

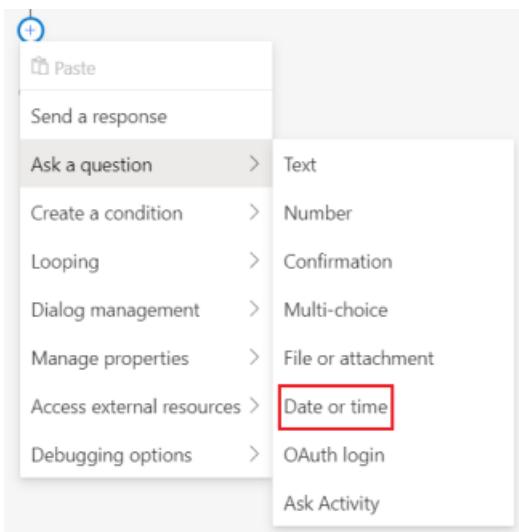
## Date or time input

The **DateTimeInput** sample demonstrates how to get date and time information from your users using **Date or time** prompt.

### Create a date time input action

To prompt a user for a date:

1. Select the + icon then select **Date or time** from the **Ask a question** menu.



2. Enter **-Please enter a date**. in the **Prompt** field of the **Properties** panel.
3. Switch to the **User Input** tab and enter `user.date` in the **Property** field.
4. Switch to the **Other** tab and enter **- Please enter a date**. in the **Invalid Prompt** field.
5. Select the **+** icon in the **Authoring panel** and select **Send a response**.
6. Enter `-You entered: ${user.date[0].value}` .

## Prompt settings and validation

In the **Properties** pane under the **Other** tab, you can set user input validation rules using [adaptive expressions](#) as well as respond independently to user input to accommodate an *unrecognized*, *invalid* and *Default value* responses.

### IMPORTANT

The value to be validated is present in the `this.value` property. `this` is a memory scope that pertains to the active action's properties. Read more in the [memory](#) concept article.

### Recognizers

Expand **Recognizers** to view and edit **Unrecognized Prompt**.

**Unrecognized Prompt** is the message that is sent to a user if the response entered was not recognized. It is a good practice to add some guidance along with the prompt. For example when a user input is the name of a city but a five-digit ZIP Code is expected, the *Unrecognized Prompt* can be the following:

```
Sorry, I do not understand '${this.value}'. Please enter a ZIP Code in the form of 12345.
```

### Validation

Expand **Validation** to define **Validation rules** and **Invalid prompts**.

- **Validation rules** are defined in [adaptive expressions](#) to validate the user's response. The input is considered valid only if the expression evaluates to `true`. An example validation rule specifying that the user input be 5 characters long can look like the following:

```
length(string(this.value)) == 5
```

- **Invalid prompt** is the message that is sent to a user if the response entered is invalid according to the

*Validation Rules.* It is a good practice to specify in the message that it is not valid and what is expected. For example:

Sorry, '\${this.value}' is not valid. I'm looking for a 5 digit number as ZIP Code. Please specify a ZIP Code in the form 12345

.

## Prompt Configurations

Expand **Prompt Configurations** to view and edit the following properties:

- **Default value response:** The value that is returned after the *max turn count* has been hit. This will be sent to the user after the last failed attempt. If this is not specified, the prompt will simply end and move on without telling the user a default value has been selected. In order for the *default value response* to be used, you must specify both the *default value* and the *default value response*.
- **Max turn count:** The maximum number of re-prompt attempts before the default value is selected. When *Max turn count* is reached to limit, the property will end up being set to `null` unless a default value is specified. Please note that if your dialog is not designed to handle a null value, it may crash the bot.
- **Default value:** The value returned when no value is supplied. When a default value is specified, you should also specify the *default value response*.
- **Allow interruptions** (true/false): This determines whether parent should be able to interrupt child dialog. Consider using the *Allow Interruptions* property to either handle a global interruption or a local interruption within the context of the dialog.
- **Always prompt** (true/false): Collect information even if specified property isn't empty.

## Next

- Learn how to [manage conversation flow](#) using conditionals and dialogs.
- [Best practices for building bots using Composer](#).

# Control conversation flow

5/28/2021 • 13 minutes to read

APPLIES TO: Composer v1.x and v2.x

The conversations a bot has with its users are controlled by the content of its dialog. Dialogs contain response templates (language generation) for messages the bot will send, along with instructions for the bot to carry out tasks. While some dialogs are linear - just one message after the other - more complex interactions will require dialogs that branch and loop based on what the user says and the choices they make. This article explains how to add both simple and complex conversation flow using examples from a sample bot provided as part of Composer samples.

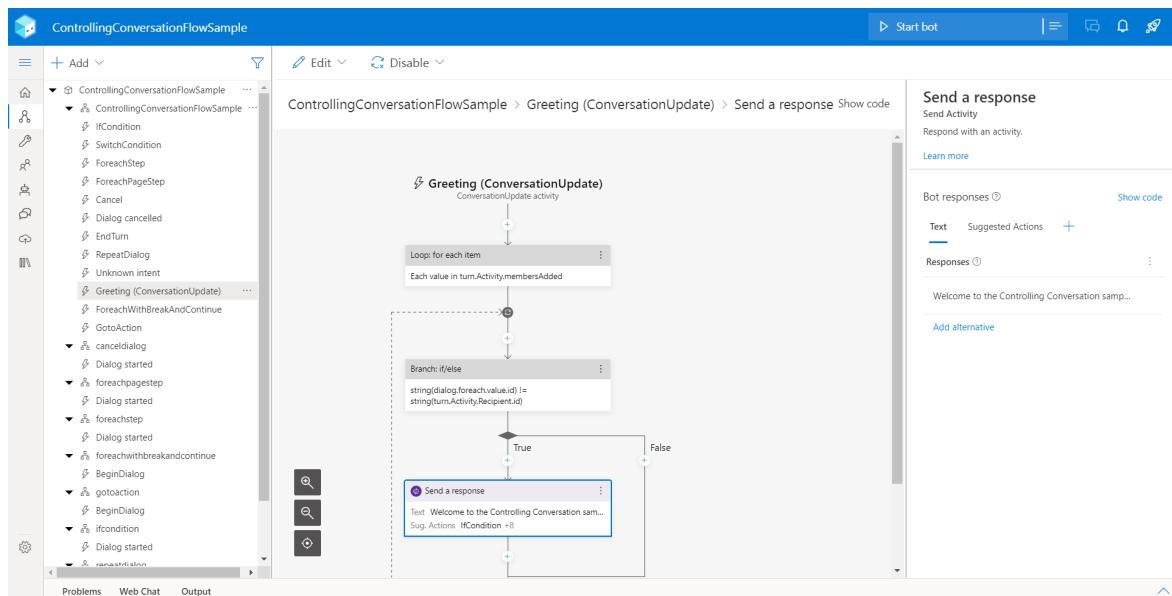
## The Controlling Conversation Flow example

This section is an introduction to the **Controlling Conversation Flow** example that is used to explain how to control conversation flow in your bot using Composer.

Do the following to get the **Controlling Conversation Flow** example running in Composer:

- [Composer v2.x](#)
- [Composer v1.x](#)

1. Clone the [Bot Builder samples github repo](#) onto your machine.
2. Within the `composer-samples` folder you'll find C# and JavaScript projects, choose a language and navigate into the `projects` subfolder.
3. In this folder you'll find a `ControllingConversationFlowSample` project which you can open in Composer.  
Now that you have it loaded in Composer, you can click Start Bot and then [Open in Web Chat](#) to explore the sample interactively.
4. Now let's take a look to see how it works.
5. Select the **Greeting** trigger in the main dialog to get an idea of how this sample works.



6. In this sample, the **Greeting** trigger always is the first thing that runs when the bot starts. This trigger

executes the **Send a response** action. The **Send a response** action calls the *WelcomeUser* template: `#{WelcomeUser()}`. To see what the help template does, open **Bot Responses**, select **Common**, and then **Show code** to see the `#WelcomeUser` template.

```

# WelcomeUser
[Activity
  Text = ${helpText}
  SuggestedActions = IfCondition|SwitchCondition|ForeachStep|ForeachPageStep|Cancel|Endturn|RepeatDialog|ForeachWithBreakAndContinue|GotoAction
]
# helpText
-->Welcome to the Controlling Conversation sample. Choose from the list below to try.
You can also type "Cancel" to cancel any dialog or "Endturn" to explicitly accept an input.---

```

### IMPORTANT

When a conversation with the bot first starts, it executes the **Greeting** trigger. The **Greeting** trigger presents the user with different options using **SuggestedActions**. When the user selects one of them, the bot sends an event with the corresponding intent. That intent is captured by the **Intent recognized** trigger created to handle that intent. For example, if the user enters *IfCondition* the **ifCondition** trigger handles that event and calls the *IfCondition* dialog. The remaining steps in this section will walk you through this process.

- Select the main dialog and look at the **Properties** pane. Note that the **Recognizer Type** is set to the **Regular expression recognizer**.

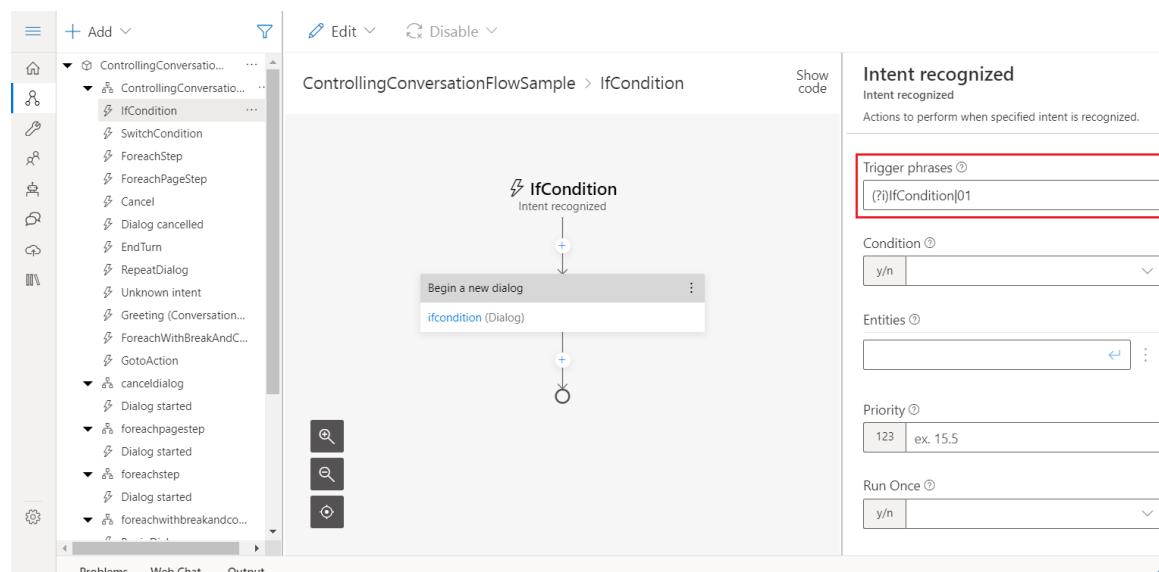
#### Language Understanding (?)

##### Recognizer Type

Regular expression recognizer

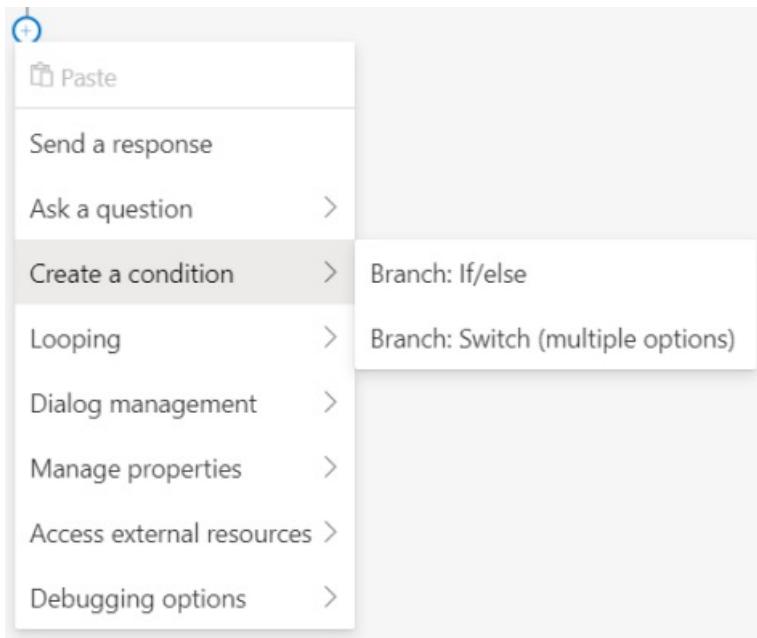
In each of the following sections you will learn how to create each of these different ways to control the conversation flow, using the corresponding dialog as an example.

- To see the *intents* for each trigger select the trigger and look in the **Properties** panel. The following image shows a Regular Expression such that if the user enters either *IfCondition* or *01* the **IfCondition** trigger will execute, the `(?i)` starts case-insensitive mode in a regular expression.



# Conditional branching

Composer offers several mechanisms for controlling the flow of the conversation. These building blocks instruct the bot to make a decision based on a [property in memory](#) or the result of an [expression](#). Below is a screenshot of the **Create a condition** menu:



- **Branch: If/Else** instructs the bot to choose between one of two paths based on a `yes / no` or `true / false` type value.
- **Branch: Switch (multiple options)** branch instructs the bot to choose the path associated with a specific value - for example, a switch can be used to build a multiple-choice menu.

## Branch: If/Else

The **Branch: If/Else** action creates a decision point for the bot, after which it will follow one of two possible branches. To create an **Branch: If/Else** branch select the + icon in the **Authoring canvas** then select **Branch: If/else** in the **Create a condition** menu.

The decision is controlled by the **Condition** field in the **Properties** panel, and must contain an [expression](#) that evaluates to true or false. For example, in the screenshot below the bot is evaluating whether `user.age` is greater than or equal to 18.

## Branch: If/Else

### If Condition

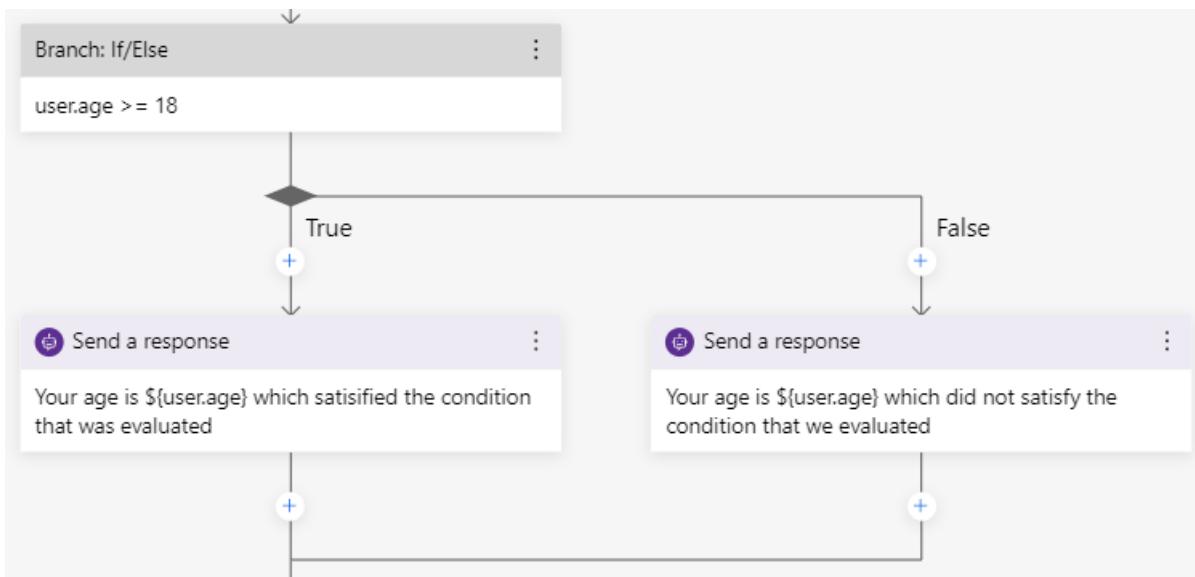
Two-way branch the conversation flow based on a condition.

[Learn more](#)

Condition \* ?

y/n	<code>user.age &gt;= 18</code>
-----	--------------------------------

Once the condition has been set, the corresponding branches can be built. The editor will now display two parallel paths in the flow - one that will be used if the condition evaluates to `true`, and one if the condition evaluates `false`. Below the bot will **Send a response** based on whether `user.age>=18` evaluates to `true` or `false`.



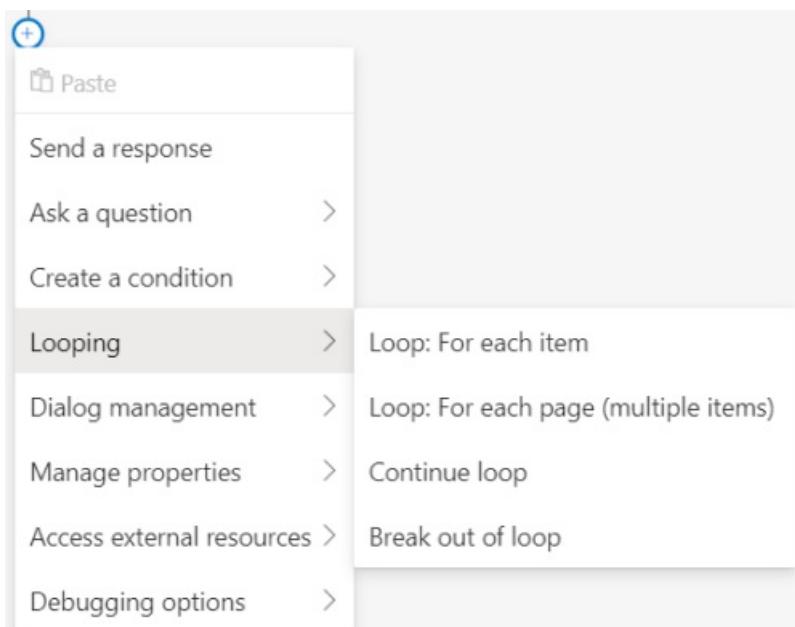
### Branch: Switch (multiple options)

In a **Branch: Switch (multiple options)**, the value of the parameter defined in the **Condition** field of the **Properties** panel is compared to each of the values defined in the **Cases** section that immediately follows the **Condition** field. When a match is found, the flow continues down that path, executing the actions it contains. To create a **Branch: Switch (multiple options)** action, select the **+** icon in the **Authoring canvas** then select **Branch: Switch (multiple options)** from the **Create a condition** menu.

Like **Branch:If/Else** you set the **Condition** to be evaluated in the **Properties** panel. Underneath you can create **Branches** in your switch condition by entering the value and press **Enter**. As each case is added, a new branch will appear in the flow which can then be customized with actions. See below how the *Nick* and *Tom* branches are added both in the property panel on the right and in the authoring canvas. In addition, there will always be a "default" branch that executes if no match is found.

## Loops

Below is a screenshot of the **Looping** menu:



- **Loop: For each item** instructs the bot to loop through a set of values stored in an array and carry out the same set of actions with each one. For very large arrays there is
- **Loop: For each page (multiple items)** that can be used to step through the array one page at a time.
- **Continue loop** instructs the bot to stop executing this template and continue with the next iteration of the

loop.

- **Break out of loop** instructs the bot to stop executing this loop.

### Loop: For each item

The **Loop: For each item** action instructs the bot to loop through a set of values stored in an array and carry out the same set of actions with each element of the array.

For the sample in this section you will first create and populate an array, then create the for each item loop.

#### Create and populate an array

To create and populate an array:

1. Select **Edit an array property** from the **Manage properties** menu.

2. In the **Properties** panel, edit the following fields:

- **Type of change:** Push
- **Items property:** dialog.ids
- **Value:** 10000+1000+100+10+1

3. Repeat the previous two steps to add two more elements to the array, setting the **Value** to:

- 200\*200
- 888888/4

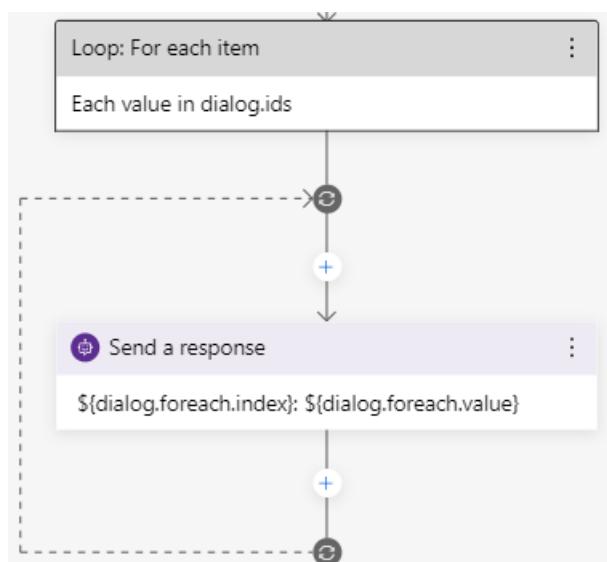
4. (optional) Send a response to the user. You do that by selecting the + in the **Authoring canvas** then **Send a response**. Enter -Pushed dialog.id into a list into the **Properties** panel.

Now that you have an array to loop through, you can create the loop.

#### Loop through the array

To create the for each loop:

1. Select the + icon in the **Authoring Canvas** then **Loop: For each item** from the **Looping** menu.
2. Enter the name of the array you created, dialog.ids into the **Items property** field.
3. To show the results when the bot is running, enter a new action to occur with each iteration of the loop to display the results. You do that by selecting the + in the **Authoring canvas** then **Send a response**. Enter -\${dialog.foreach.index}: \${dialog.foreach.value} into the **Properties** panel.



Once the loop begins, it will repeat once for each item in the array. Note that it is not currently possible to end the loop before all items have been processed. If the bot needs to process only a subset of the items, use **Branch: If/Else** and **Branch: Switch (multiple options)** branches within the loop to create nested

conditional paths.

### Loop: For each page

Loop: For each page (multiple items) loops are useful for situations in which you want to loop through a large array one page at a time. Like Loop: For each item, the bot iterates an array, the difference is that For Each Loops executes actions per item page instead of per item in the array.

For the sample in this section you will first create and populate an array, then create the for each page loop.

#### Create and populate an array

To create and populate an array:

1. Select **Edit an array** property from the **Manage properties** menu.
2. Add properties to the array you created by selecting + in the **Authoring canvas**, then **Edit an array property** from the **Manage properties** menu.
3. In the **Properties** panel, edit the following fields:
  - **Type of change:** `Push`
  - **Items property:** `dialog.ids`
  - **Value:** `1`
4. Repeat step 3, incrementing the **Value** by 1 each time until you have 6 properties in your array.

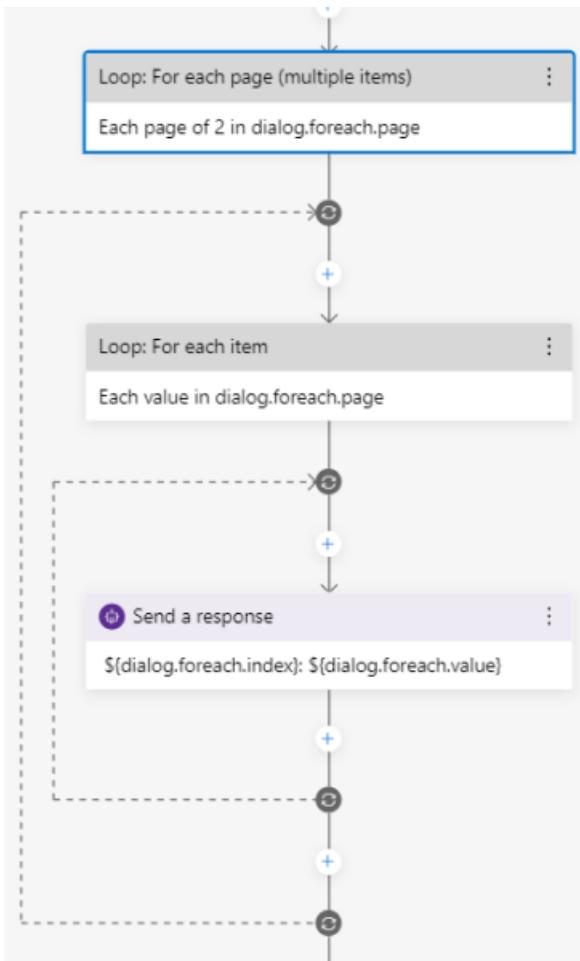
#### IMPORTANT

You will notice that this differs from the *Controlling Conversation Flow* example, the reason is to show another example of the **Page size** field in the loop that you will create next.

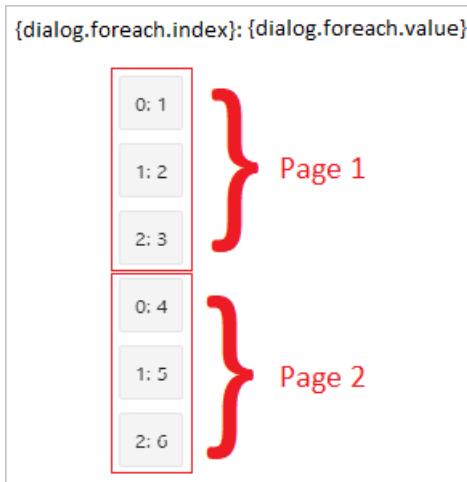
#### Loop through the array

To create the Loop: For each page action:

1. Select the + icon in the **Authoring canvas**, then **Loop: For each page (multiple items)** from the **Looping** menu.
2. Enter the name of the array you created, `dialog.ids` into the **Items property** field.
3. To show the results when the bot is running, enter a new action to occur with each iteration of the loop to display the results. You do that by selecting the + in the **Authoring canvas** then **Send a response**. Enter `- ${dialog.foreach.index}: ${dialog.foreach.value}` into the **Properties** panel.

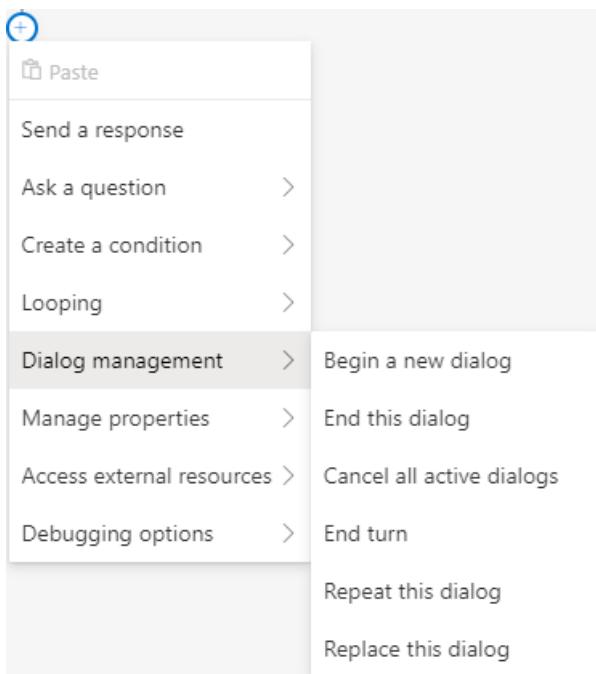


After setting the aforementioned properties your **Loop: for each page (multiple items)** loop is ready. As seen in the sample below, you can nest for **Loop: for each item** within your **Loop: for each page (multiple items)** loop, causing your bot to loop through all the items in one page and take an action before handling the next page.



## Using dialogs to control conversation

In addition to conditional branching and looping, it is also possible to compose multiple dialogs into a larger more complex interaction. Below are the available **Dialog management** options:



## Begin a new Dialog

Child dialogs are called from the parent dialog using the **Begin a new dialog** action from within any trigger. You do this by selecting the + icon in the **Authoring canvas** then select **Begin a new dialog** from the **Dialog management** menu.

Once the child dialog is called, the parent dialog pauses execution until the child completes and returns control back to its parent which then resumes where it left off.

It is possible to pass parameters into the child dialog. Parameters can be added to **Options** field in the **Begin a new dialog** action **Properties** panel. The value of each parameter is saved as a property in memory.

- [Composer v2.x](#)
- [Composer v1.x](#)

## Begin a new dialog

Begin dialog

Begin another dialog.

[Learn more](#)

Dialog name 

ChildDialog 

Property 

abc	ex. dialog.userName
-----	---------------------

Options 

object 

Key \*

Value

Add a new key

Add a new value  

Activity processed 

y/n	true
-----	------

## Begin a new dialog

Begin dialog

Begin another dialog.

[Learn more](#)

Dialog name 

ChildDialog 

Property 

abc	ex. dialog.userName
-----	---------------------

Options 

expression 

{}	={value:dialog.value, foo:"bar"}
----	----------------------------------

Activity processed 

y/n	true
-----	------

If you choose to use an **expression**:

In the screen shot above, ChildDialog will be started, and passed 2 options:

- the first will contain the value of the key `foo` and be available inside the menu dialog as `dialog.<field>`, in this case, `dialog.foo`.
- the second will contain the value of the key `value` and will available inside the menu dialog as `dialog.<field>`, in this case, `dialog.value`.

Note that it is not necessary to map memory properties that would otherwise be available automatically - that is, the `user` and `conversation` scopes will automatically be available for all dialogs. However, values stored in the `turn` and `dialog` scope do need to be explicitly passed.

In addition to passing these key/value pairs into a child dialog, it is also possible to receive a return value from the child dialog. This return value is specified as part of the **End this dialog** action, as [described below](#).

In addition to **Begin a new dialog**, there are a few other ways to launch a child dialog.

### Replace this dialog

**Replace this dialog** works just like **Begin a new dialog**, with one major difference: the parent dialog *does not* resume when the child finishes. To replace a dialog select the + icon in the **Authoring canvas** then select **Replace this dialog** from the **Dialog management** menu.

### Repeat this dialog

**Repeat this Dialog** causes the current dialog to repeat from the beginning. Note that this does not reset any properties that may have been set during the course of the dialog's first run. To repeat a dialog select the + icon in the **Authoring canvas** then select **Repeat this dialog** from the **Dialog management** menu.

### Ending Dialogs

Any dialog called will naturally end and return control to its parent dialog when it reaches the last action in its flow. While it is not necessary to explicitly call **End this dialog**, it is sometimes desirable to end a dialog before it reaches the end of the flow - for example, you may want to end a dialog if a certain condition is met.

Another reason to call the **End this dialog** action is to pass a return value back to the parent dialog. The return value of a dialog can be a property in memory or an expression, allowing developers to return complex values when necessary. To do this, select the + icon in the **Authoring canvas** then select **End this dialog** from the **Dialog management** menu.

Imagine a child dialog used to collect a display name for a user profile. It asks the user a series of questions about their preferences, finally helping them enter a valid user name. Rather than returning all of the information collected by the dialog, it can be configured to return only the user name value, as seen in the example below. The dialog's **End this dialog** action is configured to return the value of `dialog.new_user_name` to the parent dialog.

## Conditional versions of a message in Bot Responses

In addition to creating explicit branches and loops in the flow, it is also possible to create conditional versions of messages using the Language Generation syntax. The LG syntax supports the same `Adaptive expressions` as is used in the action blocks.

For example, you can create a welcome message that is different depending on whether the `user.name` property is set or not. The message template could look something like this:

```
- IF: ${user.name != null}
  - Hello, {user.name}
- ELSE:
  - Hello, human!
```

Learn more about [using memory and expressions in LG](#).

## Further Reading

[Adaptive dialogs](#)

[Adaptive expressions](#)

## Next

- [Language Generation](#)

# Define triggers

5/20/2021 • 10 minutes to read

**APPLIES TO:** Composer v1.x and v2.x

In Bot Framework Composer, each dialog includes a set of triggers (event handlers), and each trigger contains actions (instructions) for how the bot will respond to inputs received when the dialog is active. There are several different types of triggers in Composer. They all work in a similar manner and can even be interchanged in some cases. This article explains how to define each type of trigger. Before you walk through this article, please read the [events and triggers](#) concept article.

The table below lists the different types of triggers in Composer and their descriptions.

Trigger type	Description
Intent recognized	When either a LUIS or regular expression intent is recognized the <b>Intent recognized</b> trigger fires.
QnA Intent recognized	When an intent (QnAMaker) is recognized the <b>QnA Intent recognized</b> trigger fires.
Unknown intent	When user input is unrecognized or no match is found in any of the <b>Intent recognized</b> triggers, the <b>Unknown intent</b> trigger fires.
Dialog events	This type of trigger handles dialog specific events, such as a <b>Begin dialog</b> event.
Activities	This trigger type enables you to associate actions to an incoming activity, such as when a new user joins and the bot begins a new conversation ( <b>ConversationUpdate Activity</b> ).
Duplicated intents recognized	The <b>Duplicated intents recognized</b> trigger fires when multiple intents are recognized. It compares recognition results from more than one recognizer to decide a winner.
Custom events	When an <b>Emit a custom event</b> occurs the <b>Custom event</b> trigger fires.

## Intent recognized

The **Intent recognized** trigger is used to define actions to take when an intent is recognized. This trigger works in conjunction with the LUIS recognizer and the regular expression recognizer.

While LUIS offers the flexibility of a more fully featured language understanding technology, the **regular expression** recognizer works well when you need to match a narrow set of highly structured commands or keywords.

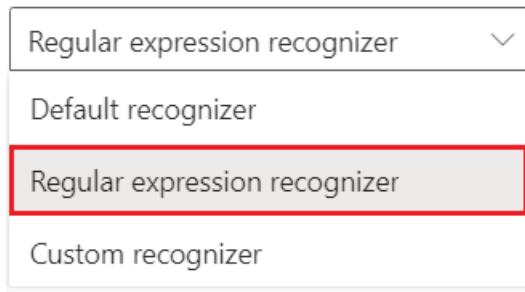
#### NOTE

In Bot Framework Composer, there are three types of recognizers: **Default recognizer** (which includes LUIS and QnA Maker), **Regular expression recognizer**, and **Custom recognizer**. Read more about [recognizers](#) in the dialogs concept article. You will also find the [recognizers in adaptive dialogs](#) article from the Bot Framework SDK documentation helpful.

Follow the steps to define an **Intent recognized** trigger with **Regular expression recognizer**:

1. Select a dialog from the **Navigation** pane of Composer's **Design** page.
2. In the **Properties** pane of your selected dialog, choose **Regular expression recognizer** as the recognizer type for the dialogs.

#### Recognizer Type



A [regular expression](#) (regex) is a special text string for describing a search pattern that can be used to match simple or sophisticated patterns in a string. Composer exposes the ability to define intents using regular expressions and also allows regular expressions to extract simple entity values.

After you select the recognizer type, create an **Intent recognized** trigger in the dialog.

3. Select the three dots next to the dialog you selected previously. Then select **+ Add a trigger**.
4. In the trigger creation screen, select **Intent recognized** from the drop-down list.
5. Enter a name in the **What is the name of this trigger?** field. This is also the name of the *intent*.
6. Enter a regular expression pattern in the **Please input regEx pattern** field. Select **Submit**.

The following image shows the definition of an **Intent recognized** trigger named **BookFlight**. User input that matches the regex pattern will fire this trigger.

A screenshot of the "Create a trigger" dialog. It has three main input fields:

- "What is the type of this trigger?" dropdown set to "Intent recognized".
- "What is the name of this trigger (RegEx)" input field containing "BookFlight".
- "Please input regEx pattern" input field containing "Book a flight to (?<toCity>.\*)".

At the bottom are "Cancel" and "Submit" buttons.

In the example above, a *BookFlight* intent is defined. However, this will *only* match the very narrow pattern "book flight to [somewhere]", whereas the LUIS recognizer will be able to match a much wider

variety of messages.

7. (Optional) Learn how to define an **Intent recognized** trigger with the LUIS recognizer in the [Add LUIS for language understanding](#) article.

## QnA Intent recognized

The **QnA Intent recognized** trigger is used to define actions to take when a QnA Maker intent is recognized. This trigger works in conjunction with the QnA Maker recognizer.

### NOTE

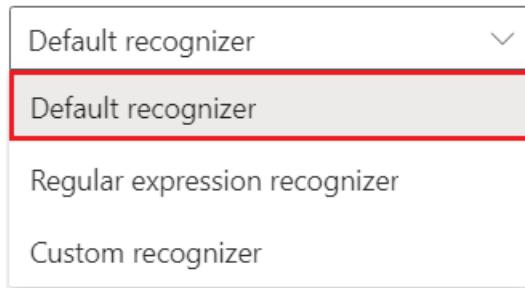
The **Default recognizer** can work as a QnA Maker recognizer when you define a QnA Maker knowledge base. Read more about [recognizers](#) in the [dialogs](#) concept article.

Follow these steps to define a **QnA Intent recognized** trigger:

- [Composer v2.x](#)
- [Composer v1.x](#)

1. Select a dialog from the **Navigation** pane of Composer's **Design** page.
2. In the **Properties** pane of your selected dialog, select **Default recognizer** as recognizer type for the dialog.

### Recognizer Type



3. Select the three dots next to the dialog you selected previously and click + **Add QnA Maker knowledge base** to create a new QnA knowledge base. You can upload a knowledge source by entering the URL, or create one from scratch by selecting the **Create custom knowledge base** button:

### Add QnA Maker knowledge base

Use Azure QnA Maker to extract question-and-answer pairs from an online FAQ. [Learn more](#)

Knowledge base name \*

Type a name for this knowledge base

FAQ website (source)

English (United States) \*

Type or paste URL

Enable multi-turn extraction

[Create custom knowledge base](#)

[Cancel](#)

[Create](#)

The questions defined in the knowledge base will be used as the QnA Maker intent to be recognized for the **QnA Intent recognized** trigger to fire.

4. Choose the type of knowledge base you want to create, enter the necessary values, and then select **Create**.

A **QnA Intent recognized** trigger with pre-configured actions will appear in the authoring canvas.

**TIP**

For more information see how to [create a QnA knowledge base in Composer](#).

## Unknown intent

The **Unknown intent** trigger is used to define actions to take when a user input is unrecognized or no match is found in any of the **Intent recognized** triggers. You can also use this as your first trigger in your main dialog in place of the **OnBeginDialog** to perform any needed tasks when the dialog first starts.

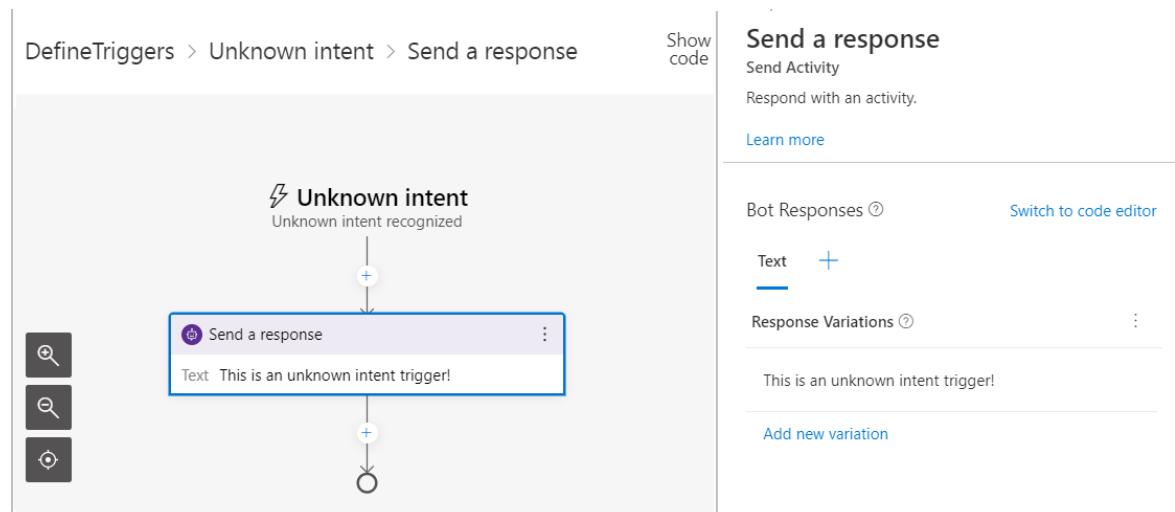
Follow these steps to define an **Unknown intent** trigger:

1. Select a dialog from the **Navigation** pane of Composer's **Design** page.
2. Select the three dots next to the dialog Then select **+ Add a trigger**. Select **Unknown intent** and select **Submit**.

You will see an empty **Unknown intent** trigger in the authoring canvas.

3. Select the **+** sign under the **Unknown intent** trigger node to add action node(s). For example, if you select **Send a response** and enter "This is an unknown intent trigger!" in the **Text** box. The message will be sent to the user anytime the **Unknown intent** trigger fires.

- [Composer v2.x](#)
- [Composer v1.x](#)



**TIP**

Read more about [actions](#) in the dialogs concept article.

## Dialog events

The **Dialog events** triggers handle dialog specific events, like **Dialog started (Begin dialog event)**. Most

dialogs will include a trigger configured to respond to the `BeginDialog` event, which fires when the dialog begins and allows the bot to respond immediately.

Follow these steps to define a **Dialog started** trigger:

1. Select a dialog from the **Navigation** pane of Composer's **Design** page.
2. Select the three dots next to the dialog. Then select **+ Add a trigger**.
3. In the trigger creation window, select **Dialog events** from the drop-down list.
4. Select **Dialog started (Begin dialog event)** from the **Which event?** drop-down list then select **Submit**.

## Create a trigger

What is the type of this trigger?

Dialog events

Which event?

Select an event type

Dialog started (Begin dialog event)

Dialog cancelled (Cancel dialog event)

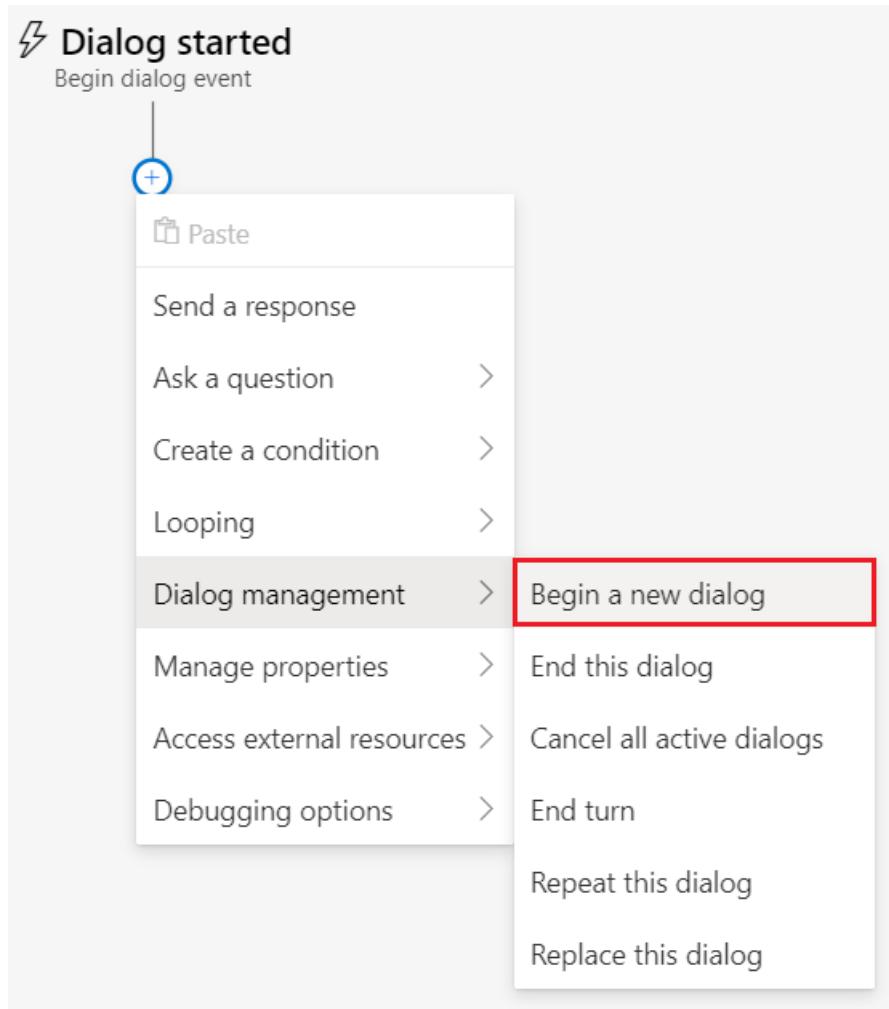
Error occurred (Error event)

Re-prompt for input (Reprompt dialog event)

Cancel

Submit

5. Select the **+** sign under the **Dialog started** node and then select **Begin a new dialog** from the **Dialog management** menu.



6. Before you can use this trigger, you need to associate a dialog to it. You do this by selecting a dialog from the **Dialog name** drop-down list in the **Properties** panel on the right side of the Composer window. You can select an existing dialog or create a new one. The example below demonstrates selecting an existing dialog named *getWeather*.

## Begin a new dialog

Begin Dialog

Begin another dialog.

[Learn more](#)

Dialog name

A screenshot of a dropdown menu for selecting a dialog. The menu items are: getWeather (selected and highlighted with a red box), Write an expression, and Create a new dialog.

## Activities

The **Activities** triggers are used to handle activity events such as your bot receiving a `ConversationUpdate` Activity. This indicates a new conversation begins and you should use a **Greeting (ConversationUpdate activity)** trigger to handle it.

Follow the steps to create a **Greeting (ConversationUpdate activity)** trigger to send a welcome message:

1. Select a dialog from the **Navigation** pane of Composer's **Design** page.
2. Select the three dots next to the dialog you selected previously. Then select **+ Add a trigger**.
3. In the trigger creation window, select **Activities** from the drop-down list.
4. Select **Greeting (ConversationUpdate activity)** from the **Which activity type?** drop-down list then select **Submit**.

## Create a trigger

What is the type of this trigger?

Activities

Which activity type?

Greeting (ConversationUpdate activity)

Cancel

Submit

5. After you select **Submit**, you will see the trigger node created in the authoring canvas.
6. You will see pre-configured actions on the authoring canvas after creating the trigger. Add a welcome message in the **Send a response** **action** to greet users.

## Duplicated intents recognized

The **Duplicated intents recognized** trigger is used to define actions to take when multiple intents are recognized. It compares recognition results from more than one recognizer to decide a winner. This trigger works in conjunction with CrossTrained recognizer.

### NOTE

The **Default recognizer** can work as a cross trained recognizer when you have both LUIS and QnA intents defined. Read more about [recognizers](#) in the [dialogs](#) concept article. You will also find the [recognizers in adaptive dialogs](#) article from the Bot Framework SDK documentation helpful.

Follow the steps to define a **Duplicated intents recognized** trigger:

1. Select a dialog from the **Navigation** pane of Composer's **Design** page.
2. In the **Properties** pane of your selected dialog, select **Default recognizer** as recognizer type for the dialog.

## Recognizer Type

Default recognizer

Default recognizer

Regular expression recognizer

Custom recognizer

3. Select the three dots next to the dialog you selected previously. Then select **+ Add a trigger**.

4. In the trigger creation screen, select **Duplicated intents recognized** from the drop-down list.

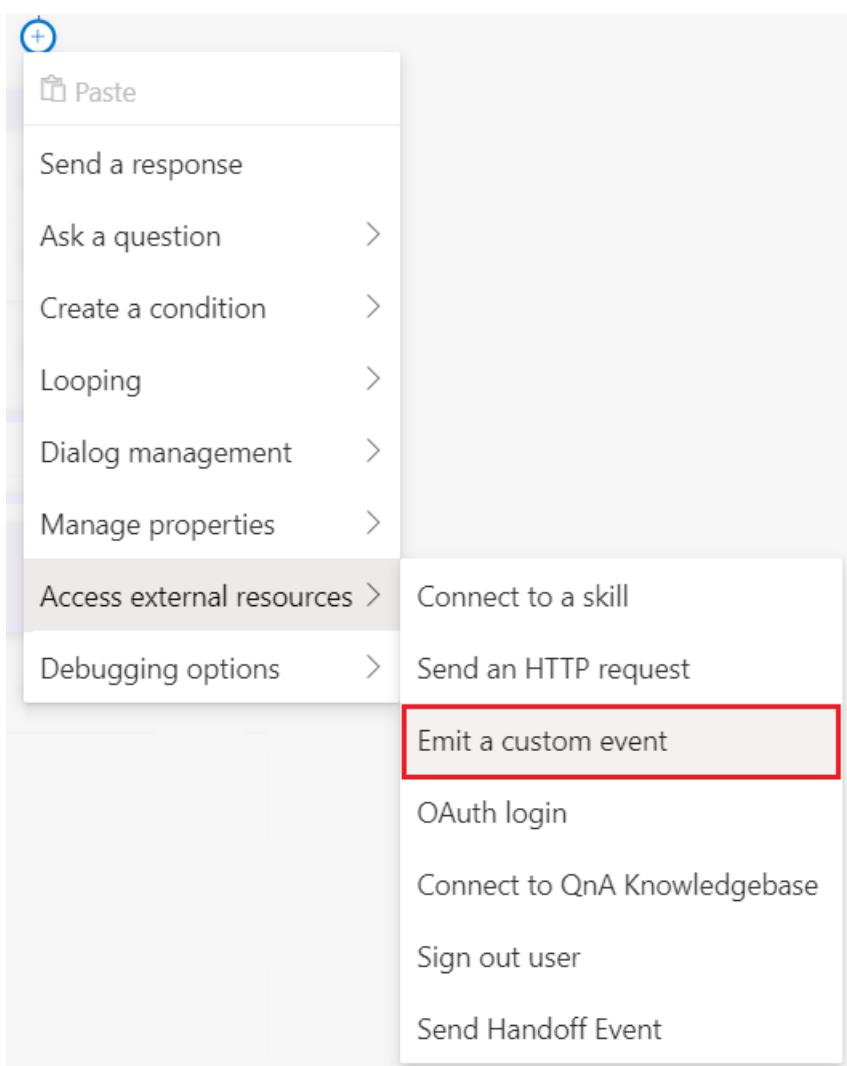
After you select **Submit**, you will see in the authoring canvas a **Duplicated intents recognized** trigger created with some pre-configured actions.

## Custom events

The **Custom events** trigger will only fire when a matching **Emit a custom event** occurs. It's a trigger that any dialog in your bot can consume. To define and consume a **Custom events** trigger, you need to first create an **Emit a custom event** and then create a **Custom events** trigger to handle the event.

### Create an **Emit a custom event**

1. Select the trigger you want to associate your **Custom event** with. Select the **+** sign and then select **Emit a custom event** from the **Access external resources** drop-down list.



2. In the **Properties** pane on the right side of Composer, select the **abc** button under the **Event name** box and then select **abc custom event**. Enter a name (like `getWeather`, seen below) and set **Bubble event** to be **true**.

## Emit a custom event

Emit Event

Emit an event. Capture this event with a trigger.

[Learn more](#)

Event name \* ⓘ

abc	getWeather
-----	------------

Event value ⓘ

abc	ex. Hello \${user.name}
-----	-------------------------

Bubble event ⓘ

y/n	true	▼
-----	------	---

### TIP

When **Bubble event** is set to be **true**, any event that is not handled in the current dialog will *bubble up* to that dialog's parent dialog where it will continue to look for handlers for the custom event.

Now that your **Emit a custom event** has been created, you can create a **Custom event** trigger to handle this event. When the **Emit a custom event** occurs, any matching **Custom event** trigger at any dialog level will fire. Follow the steps to create a **Custom event** trigger to be associated with the previously defined **Emit a custom event**.

### Create a **Custom events** trigger

1. Select a dialog from the **Navigation** pane of Composer's **Design** page.
2. Select **+ Add** from the toolbar then select **Add new trigger**.
3. In the trigger creation window, select **Custom events** from the drop-down list.
4. Enter the event name of the **Emit a custom event** you created in the previous section in the **What is the name of the custom event?** field. Select **Submit**.

## Create a trigger

What is the type of this trigger?

Custom events



What is the name of the custom event? \*

getWeather

Cancel

Submit

- Now you can add an action to your custom event trigger, which defines what will happen when it's triggered. Do this by selecting the + sign and then **Send a response** from the actions menu. Enter the desired response for this action in the Language Generation editor, for this example enter *This is a custom trigger!*.

- Composer v2.x
- Composer v1.x

DefineTriggers > getWeather > Send a response Show code



### Send a response

Send Activity

Respond with an activity.

[Learn more](#)

Bot Responses ⓘ

[Switch to code editor](#)

Text +

Response Variations ⓘ

This is a custom trigger!

[Add new variation](#)

Now you have completed both of the required steps needed to create and execute a custom event. When **Emit a custom event** fires, it's handled by the **Custom events** trigger by sending the response you defined.

This is a custom trigger!



Type your message



Next

- Learn how to control conversation flow.

# Define intents with entities

5/20/2021 • 7 minutes to read

**APPLIES TO:** Composer v1.x and v2.x

Conversations do not always progress in a linear fashion. Users will want to cover specify information, present information out of order, or make corrections etc. Bot Framework Composer supports language understanding in these advanced scenarios, with the advanced dialog capabilities offered by adaptive dialogs and LUIS application.

In this article, we will cover some details of how LUIS recognizer extracts the intent and entity you may define in Composer. the code snippets come from the *To do with LUIS* example. Read the [How to use samples](#) article and learn how to open the example bot in Composer.

## Prerequisites

- A basic understanding of the [intent and entity](#) concepts.
- A basic understanding of [how to define an Intent Recognized trigger](#).
- A basic understanding of [how to use LUIS in Composer](#).
- A [LUIS account](#) and a [LUIS authoring key](#).

## The *Todo with LUIS* example

This section is an introduction to the **Todo with LUIS** example (sample bot) that is used in this article to explain how to define intent with entities using Composer.

Do the following to get the **Todo with LUIS** example running in Composer:

- [Composer v2.x](#)
  - [Composer v1.x](#)
1. Clone the [Bot Builder samples github repo](#) onto your machine.
  2. Within the `composer-samples` folder you'll find C# and JavaScript projects, choose a language and navigate into the `projects` subfolder.
  3. In this folder you'll find a `ToDoBotWithLUISsample` project which you can open in Composer.
  4. You will notice in the upper right hand side of the screen that you have two errors. If you select the error icon it will bring up a the Diagnostic pane.

The screenshot shows the Microsoft Bot Framework Composer interface. On the left, there's a navigation sidebar with options like Home, Create, Configure, User input, Bot responses, Knowledge base, Publish, and Package manager. Below that is a 'Composer settings' section. The main workspace is titled 'AddItem' and contains a large hexagonal icon with two dots. Below it, the text 'Select a trigger in the left navigation to see actions' is displayed. To the right of the icon, there's a detailed configuration panel for 'AddItem'. It includes sections for 'Adaptive dialog', 'Language Understanding', 'Recognizer Type' (set to 'Default recognizer'), 'Auto end dialog' (set to 'y/n true'), and 'Default result property' (set to 'abc dialog.result'). At the bottom of the configuration panel, there's a table with columns 'Bot ↑', 'Location', and 'Description', showing two entries related to 'appsettings.json'.

5. Selecting either error will bring up the **Configure your bot** screen where you can add your LUIS key and region

This screenshot shows the 'Configure your bot' screen. The left sidebar has 'Configure' selected. The main area is titled 'Configure your bot' and contains a 'ToDoBotWithLUISample (root)' section. Under this section, there's a 'Language Understanding authoring key' field with a placeholder 'Type Language Understanding authoring key' and a note below it stating 'LUIS key is required with the current recognizer setting to start your bot locally, and publish'. There's also a 'Select region' dropdown and a 'Set up Language Understanding' button. Below this, there's an 'Azure QnA Maker' section with a note about OnA Maker being an Azure Cognitive service. At the bottom, there's a table with columns 'Bot ↑', 'Location', and 'Description', showing two entries related to 'appsettings.json'.

Now that you have the example loaded in Composer, take a look to see how it works.

## Luis for entity extraction

In addition to specifying intents and utterances as instructed in the [how to use LUIS in Composer](#) article, it is also possible to train LUIS to recognize named entities. Extracted entities are passed along to any triggered actions or child dialogs using the syntax `@{Entity Name}`. For example, given an intent definition like below:

```
# BookFlight
- book me a flight to {city=shanghai}
- travel to {city=new york}
- i want to go to {city=paris}
```

When triggered, if LUIS is able to identify a city, the city name will be made available as `@city` within the triggered actions. The entity value can be used directly in expressions and LG templates, or [stored into a memory property](#) for later use. The JSON view of the query "book me a flight to London" in LUIS app looks like

this:

```
{  
  "query": "book me a flight to london",  
  "prediction": {  
    "normalizedQuery": "book me a flight to london",  
    "topIntent": "BookFlight",  
    "intents": {  
      "BookFlight": {  
        "score": 0.9345866  
      }  
    },  
    "entities": {  
      "city": [  
        "london"  
      ],  
      "$instance": {  
        "city": [  
          {  
            "type": "city",  
            "text": "london",  
            "startIndex": 20,  
            "length": 6,  
            "score": 0.834206,  
            "modelTypeId": 1,  
            "modelType": "Entity Extractor",  
            "recognitionSources": [  
              "model"  
            ]  
          }  
        ]  
      }  
    }  
  }  
}
```

## Flexible entity extraction

In Composer, you can achieve flexible entity extraction by setting the **Recognizer Type** of your desired dialog to LUIS and subsequently using language understanding notation to define user response to the specific input.

You can take a look at the `AskForName` input and all its configured properties under the `BeginDialog` trigger of the `UserProfile` dialog in the `ToDoWithLuis` example.

The screenshot shows the Microsoft Bot Framework Composer interface. On the left, the navigation tree for the `ToDoWithLuis` sample includes nodes for `WelcomeUser`, `Add`, `Delete`, `View`, `UserProfile`, `whatCanYouDo`, `cancel`, `Unknown intent`, `additem`, `BeginDialog`, `deleteitem`, `BeginDialog`, `help`, `BeginDialog`, `userprofile`, and `BeginDialog`. The `userprofile` node is expanded, showing `Cancel`, `Why`, `NoValue`, `viewitem`, and `BeginDialog`. The `BeginDialog` node under `userprofile` is selected.

The main workspace displays the `UserProfile > BeginDialog > AskForName` path. The `AskForName` input is highlighted with a blue box. Its configuration pane shows the following details:

- Trigger:** `BeginDialog`
- Text Input:** `Hi, what is your name? You can \n - state your name\n - say things like 'my name is <your name>'\n - ask 'why do you need my name' \n - say 'I'm not going to give you my name'.`
- User Input:** `user.name = Input(Text)`
- Output format:** `abc ex. .toUpper(this.value), ${toUpper(this.value)}`
- Value:** `abc = coalesce(@userName, @personName)`
- Expected responses (intent: #TextInput\_Response\_267073):**
  - 1 - my name is {@userName = vishwac}
  - 2 - I'm {@userName = tom}
  - 3 - you can call me {@userName = chris}
  - 4 - I'm {@userName = scott} and I'm {@user
  - 5 - > add few patterns
  - 6 - my name is {@userName}
  - 7
  - 8 - > add entities
  - 9 - @prebuilt personName hasRoles userNam
  - 10
  - 11
  - 12

The input has the following configuration:

```
"property": "user.name"
"value": "=coalesce(@userName, @personName)"
"allowInterruptions": "!@userName && !@personName"
```

And the following `Expected response` LU configuration:

```
- my name is {@userName = vishwac}
- I'm {@userName = tom}
- you can call me {@userName = chris}
- I'm {@userName = scott} and I'm {@userAge = 36} years old
> add few patterns
- my name is {@userName}

> add entities
@ prebuilt personName hasRoles userName
```

There are two key properties in the example above: `value` and `allowInterruptions`.

The expression specified in `value` property will be evaluated on every single time user responds to the specific input. In this case, the expression `=coalesce(@userName, @personName)` attempts to take the first non null entity value `userName` or `personName` and assigns it to `user.name`. The input will issue a prompt if the property `user.name` is null even after the value assignment unless `always prompt` evaluates to `true`.

The next property of interest is `allowInterruptions`. This is set to the following expression:

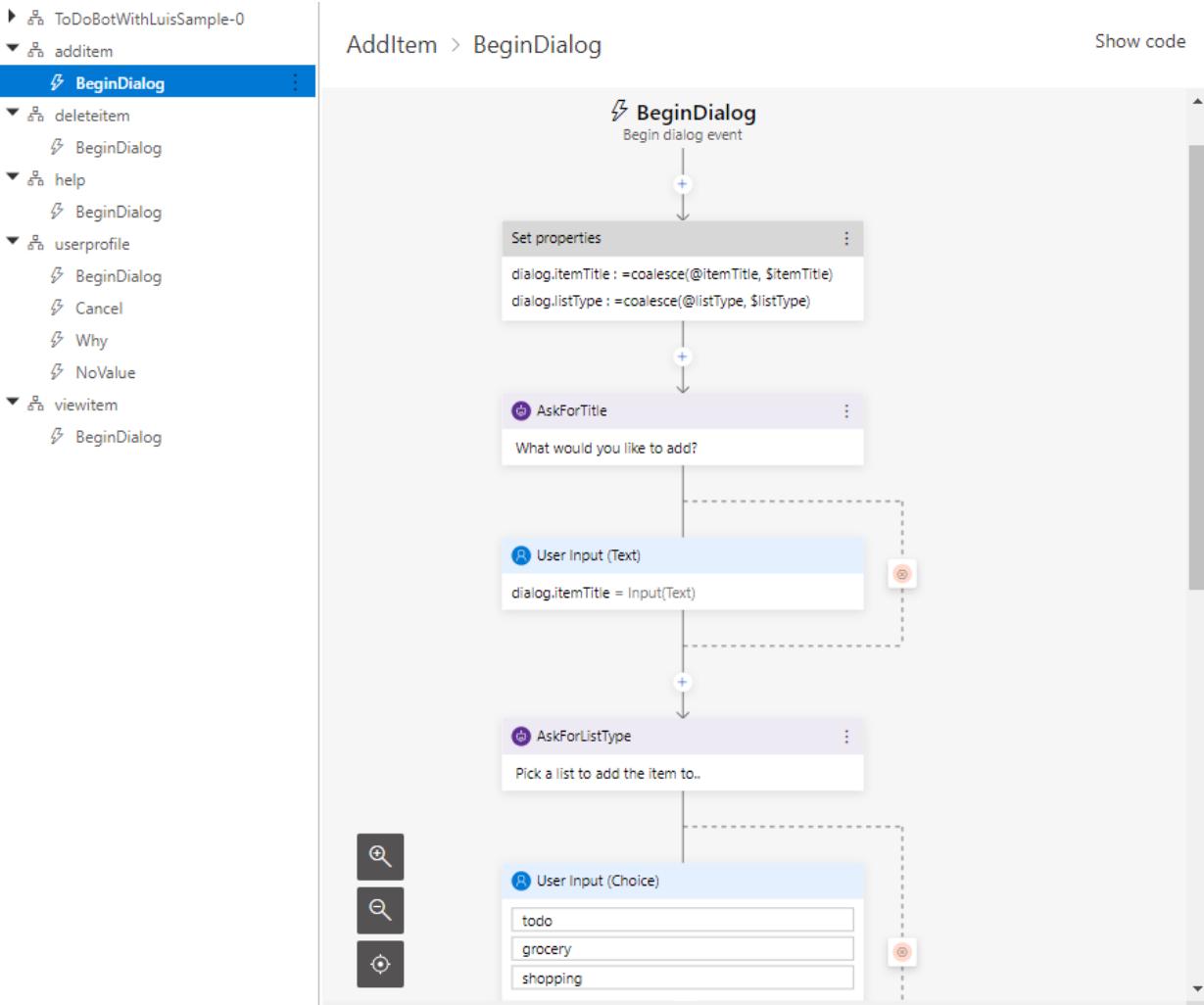
`!@userName && !@personName`. This literally means what this expression reads - *allow an interruption if we did not find a value for entity `userName` or entity `personName`*.

Notice that you can just focus on things the user can say to respond to this specific input in the `Expected responses`. With these capabilities, you get to provide labelled examples of the entity and use it no matter where or how it was expressed in the user input.

If a specific user input does not work, simply try adding that utterance to the `Expected response`.

## Out of order entity extraction

To see how the out of order entity extraction is wired up, you can see the `AskForTitle` and `AskForListType` inputs, which are under the `BeginDialog` trigger of the `Additem` dialog in the `ToDoWithLuis` example.



Take a look at this example below:

```

user: add an item to the list
bot: sure, what is the title?
user: buy milk
bot: ok. pick the list type - todo | shopping
user: shopping list
bot: ok. i've added that.

```

The user could have answered multiple questions in the same response. Here is an example

```

user: add an item to the list
bot: sure, what is the title?
user: add buy milk to the shopping list
bot: ok. I've added that.

```

By including the `value` property on each of these inputs, we can pick up any entities recognized by the recognizer even if it was specified out of order.

## Interruption

Interruptions can be handled at two levels - locally within a dialog as well as re-routing as a global interruption. By default adaptive dialog does this for any inputs:

1. On every user response to an input action's prompt,
2. Run the recognizer configured on the parent adaptive dialog that holds the input action
3. Evaluate the `allowInterruption` expression. a. If it evaluates to `true`, evaluate the triggers that are tied to the

parent adaptive dialog that holds the input action. If any triggers match, execute the actions associated with that trigger and then issue a re-prompt when the input action resumes. b. If it evaluates to `false`, evaluate the `value` property and assign it as a value to the `property`. If `null` run the internal entity recognizer for that input action (e.g. number recognizer for number input etc) to resolve a value for that input action.

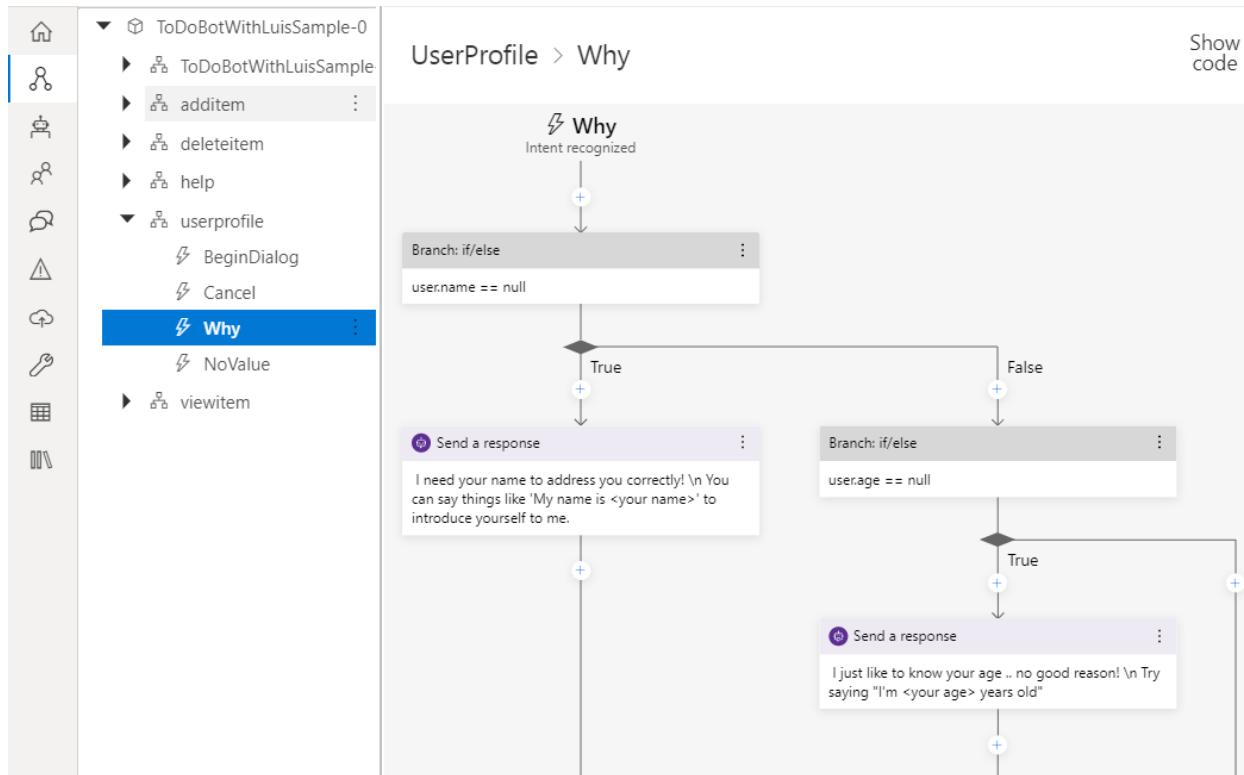
The `allowInterruption` property is located in the **Properties** panel of the **Prompt Configurations** section (**Other tab**) of an input action. You can set the value to be `true` or `false`.

### Handling interruptions locally

With this, you can add contextual responses to inputs via `OnIntent` triggers within a dialog. Consider this example:

```
user: hi
bot: hello, what is your name?
user: why do you need my name?
bot: I need your name to address you correctly.
bot: what is your name?
user: I will not give you my name
bot: Ok. You can say "My name is <your name>" to re-introduce yourself to me.
bot: I have your name as "Human"
bot: what is your age?
```

You can see the `Why`, `NoValue` or `Cancel` triggers, which are under the `userprofile` dialog in the *ToDoWithLuis* example.



### Handling interruptions globally

Adaptive dialogs have a consultation mechanism which propagates a user message up the parent dialogs until a dialog has a trigger that fires. If no dialog's triggers fired upon consultation then the active input action gets the user utterance back for its own processing. Consider this example:

```
user: hi
bot: hello, what is your name?
user: what can you do?
bot: I'm a demo bot. I can manage todo or shopping lists.
bot: what is your name?
```

Notice that the bot understood interruption and presented the help response. You can see the `UserProfile` and `Help` dialogs in the *ToDoWithLuis* example.

## Further reading

- [Entities and their purpose in LUIS](#)
- [.lu file format](#)

# Add LUIS for language understanding

5/20/2021 • 6 minutes to read

**APPLIES TO:** Composer v1.x and v2.x

This article will instruct how to integrate language understanding in your bot using the cloud-based service [LUIS](#). LUIS lets your bots identify valuable information from user input by interpreting user needs ([intents](#)) and extracting key information ([entities](#)). Understanding user intent makes it possible for your bot to know how to respond with helpful information using [language generation](#).

## Prerequisites

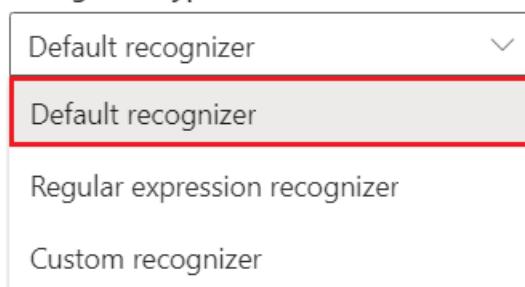
- [A basic bot built using Composer](#)
- Knowledge of [language understanding](#) and [events and triggers](#)
- A [LUIS account](#) and a [LUIS authoring key](#)

## Update the recognizer type to LUIS

Composer uses [recognizers](#) to interpret user input. A dialog can use only one type of recognizer, each of which are set independently from the other dialogs in your bot.

1. Select the dialog that needs language understanding capabilities in the **Navigation** pane.
2. In the **Properties** panel, select **Default recognizer** from the **Recognizer Type** drop-down list.

### Recognizer Type



#### NOTE

The **Default recognizer** can be one of the following recognizers:

- *None* - do not use recognizer.
- *LUIS recognizer* - to extract intents and entities from a user's utterance based on the defined [LUIS](#) application.
- *QnA Maker recognizer* - to extract intents from a user's utterance based on the defined [QnAMaker](#) application.
- *Cross-trained recognizer set* - to compare recognition results from more than one recognizer to decide a winner.

## Add language understanding data and conditions

- [Composer v2.x](#)
- [Composer v1.x](#)

1. Select the desired dialog in the **Navigation** pane. Click the three dots next to the dialog and then click **+**

Add new trigger.

+ Add new trigger

+ Add QnA Maker knowledge base

2. On the **Create a trigger** screen.

- a. Select **Intent recognized** for the trigger type.
- b. Enter a name for the trigger, like **Greeting**.
- c. Enter example phrases in the **Trigger phrases** field using the [.lu file format](#).

- Bye.
- Goodbye.
- See you later.
- Take it easy.

This is what your create a trigger screen should look like:

## Create a trigger

What is the type of this trigger?

Intent recognized

What is the name of this trigger?

Goodbye

Trigger phrases

+ Add entity ▾ > Insert entity ▾

- Bye.
- Goodbye.
- See you later.
- Take it easy.

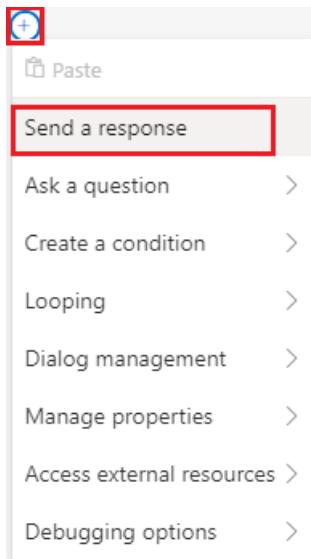
Cancel

Submit

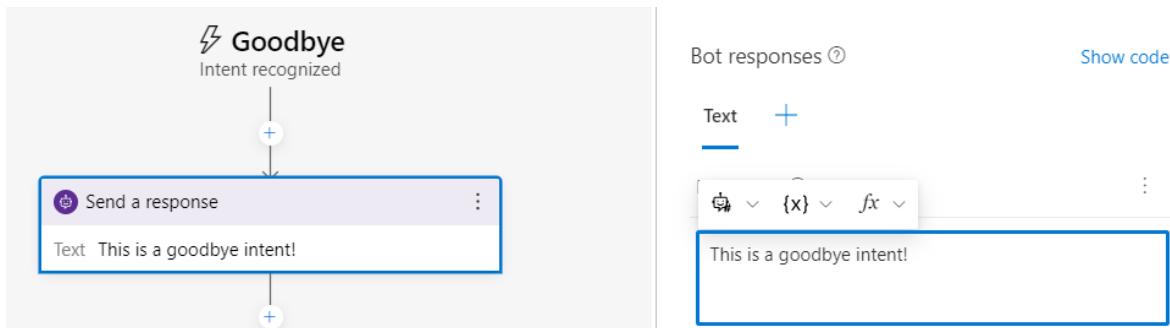
3. After you click **Submit**, you will see the trigger node created in the authoring canvas.

4. If you want to specify a threshold for a specific intent, set a condition for this intent in the **Condition** field on the **Properties** panel on the right. An example condition can be `=#goodbye.score >=0.8`.
5. You can add actions to execute when the trigger fires, for example, the **Send a response** action. Create a new action by selecting the plus (+) icon in the **Authoring canvas**, then **Send a response** from the

drop-down list.



6. Enter **This is a greeting intent!** in the **response editor** of the **Properties** panel.

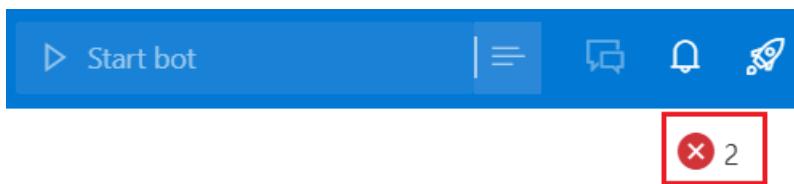


## Update LUIS keys

After you are done with all previous steps you need to provide Composer with information from LUIS.

- [Composer v2.x](#)
- [Composer v1.x](#)

1. Select the error icon in the upper right corner of Composer. The number 2 is next to it, indicating that there are two errors.



These errors occur because you have yet to add LUIS information Composer needs. After selecting the error icon, the **Problems pane** will appear, as seen below:

Problems	Web Chat	Output
<span style="color: red;">✖</span> 2 errors ⚠ 0 warnings		
	Bot ↑	Location
	empty_test	appsettings.json
	empty_test	appsettings.json

2. Select the `appsettings.json` link in either of the errors. Composer will switch to the **Azure Language**

Understanding of the Configure your bot page.

## Azure Language Understanding

Language Understanding (LUIS) is an Azure Cognitive Service that uses machine learning to understand natural language input and direct the conversation flow. [Learn more](#). Use an existing Language Understanding (LUIS) key from Azure or create a new key. [Learn more](#)

Application name [\(?\)](#)

empty\_test

Language Understanding authoring key [\\* \(?\)](#)

Type Language Understanding authoring key

 LUIS key is required with the current recognizer setting to start your bot locally, and publish

Language Understanding region [\\* \(?\)](#)

Select region

**Set up Language Understanding**

3. Select the **Set up Language Understanding** button. The **Set up Language Understanding** window will appear, shown below:

## Set up Language Understanding

X

To understand natural language input and direct the conversation flow, your bot needs a language understanding service. [Learn more](#)

- Use existing resources
- Create and configure new Azure resources
- Generate instructions for Azure administrator

Next

Cancel

The following sections explain the different options for setting up language understanding. Below is a link to each set up option and when to choose each:

- [Use existing resources](#): You already have existing Azure resources.
- [Create and configure new Azure resources](#): You don't have existing Azure resources and need to create and configure new ones.

- **Generate instructions for Azure administrator:** You need an administrator to create Azure resources for you.

#### Use existing resources

Choose this option if you already have existing language understanding Azure resources for your bot.

1. Select **Use existing resources** and then select **Next** at the bottom of the window.
2. Now select the **Azure directory**, **Azure subscription**, and **Language Understanding resource name**, shown in the screenshot below:

## Select Language Understanding resources X

Select your Azure directory, then choose the subscription where your existing Language Understanding resource is located.[Learn more](#)

**Azure directory \***

**Azure subscription \***

**Language Understanding resource name**

**Back**

**Next**

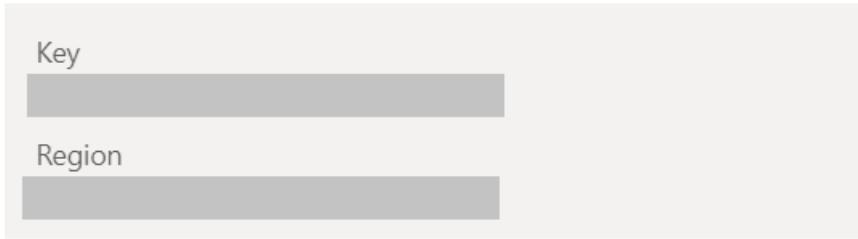
**Cancel**

3. Select **Next** on the bottom. Your **Key** and **Region** will appear on the next screen.

## Select Language Understanding resources

X

The following Language Understanding keys have been successfully added to your bot project:



**Done**

below:

4. Select **Done**.

Your bot is now ready to test.

### Create and configure new Azure resources

Choose this option if you need to create and configure new Azure resources for language understanding.

1. Select **Create and configure new Azure resources** and then select **Next** at the bottom of the window.
2. Select the **Azure directory** and **Azure subscription** you want to use to create your resources.

## Create Language Understanding resources X

Select your Azure directory, then choose the subscription where you'd like your new Language Understanding resource.[Learn more](#)

Azure directory \*



Azure subscription \*



[Back](#)

[Next](#)

[Cancel](#)

3. Fill in the values for the following on the next window:

- a. **Azure resource group:** An existing Azure resource group, or you can create a new one (seen in the screenshot below).
- b. **Region:** A region from the drop-down list.
- c. **Language Understanding resource name:** The name of your language understanding resource.

## Create Language Understanding resources X

Select the resource group and region in which your Language Understanding service will be created.

### Azure resource group

Create new ▼

### Resource group name \*

### Region \*

 ▼

### Language Understanding resource name \*

Back

Next

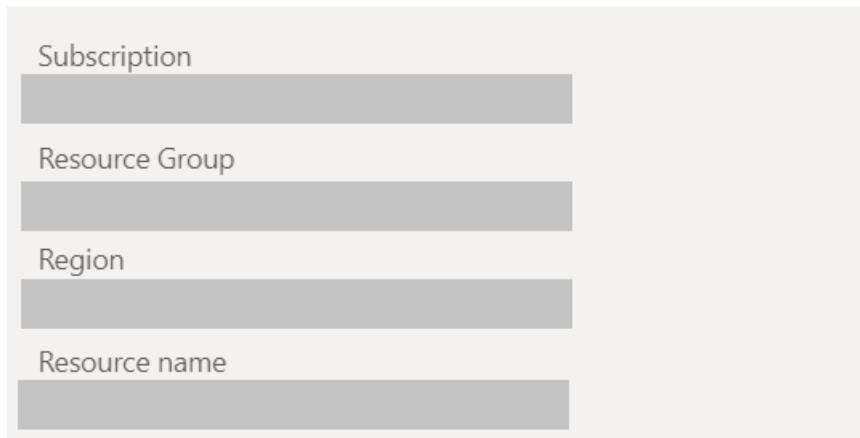
Cancel

4. Select the **Next** button. Blue text will appear on the bottom left of the window indicating that Azure is **Creating resources**.
5. Once your resource has been successfully created, Composer will display the **Subscription**, **Resource Group**, **Region**, and **Resource name**, as show in the screenshot below:

## Create Language Understanding resources

X

The following Language Understanding resource was successfully created and added to your bot project:



**Done**

6. Select **Done**.

Your bot is now ready to test.

### Generate instructions for Azure administrator

Choose this option if you need an administrator to create Azure resources for you.

1. Select **Generate instructions for Azure administrator** and then select **Next** at the bottom of the window.
2. Copy the instructions on the next window and send them to your Azure administrator. They will create the resources you need to LUIS in your bot.

## Generate instructions for Azure administrator

X

If Azure resources and subscription are managed by others, use the following information to request creation of the resources that you need to build and run your bot.

### Instructions



I am creating a conversational experience using Microsoft Bot Framework project. For my project to work, it needs Azure resources including Language Understanding. Below are the steps to create these resources.

1. Using the Azure portal, please create a Language Understanding resource.
2. Once created, securely share the resulting credentials with me as described in the link below.

Detailed instructions:  
<https://aka.ms/bfcomposerhandoffluis>

[Back](#)

[Okay](#)

3. Select **Okay**.

4. Your administrator will send you a **JSON** with your bot's configuration information. On the **Configure your bot** page, toggle on the **Advanced Settings View (json)**. Replace the existing JSON with the one supplied by your administrator.

Your bot is now ready to test.

## Test

It's always a good idea to verify that your bot works correctly when you add new functionality. You can test your bot's new language understanding capabilities using the Emulator.

1. Click **Start bot** on the top right to start your bot's local runtime.
2. After the bot finishes building click the **Open Web Chat** button that pops up in the **Local bot runtime manager**.
3. When Web Chat pops open, send it messages using the various utterances you created to see if it executes your **Intent recognized** triggers.

## Next

- Learn [how to add a QnA Maker knowledge base to your bot](#).

# Add a QnA Maker knowledge base to your bot

5/20/2021 • 5 minutes to read

**APPLIES TO:** Composer v1.x and v2.x

This article will show you how to add an existing QnA Maker knowledge base to your bot. Composer provides the **Connect to QnA Knowledgebase** action which makes the process easy. You will find this helpful when you want to send a user question to your bot then have the QnA Maker knowledge base provide the answer. If you want to create a QnA Maker knowledge base directly in Composer, see the [Create a QnA Maker knowledge base](#) article.

**TIP**

If you are looking for how to create a QnA Maker knowledge base in Composer, read the [Create a QnA Maker knowledge base](#) article.

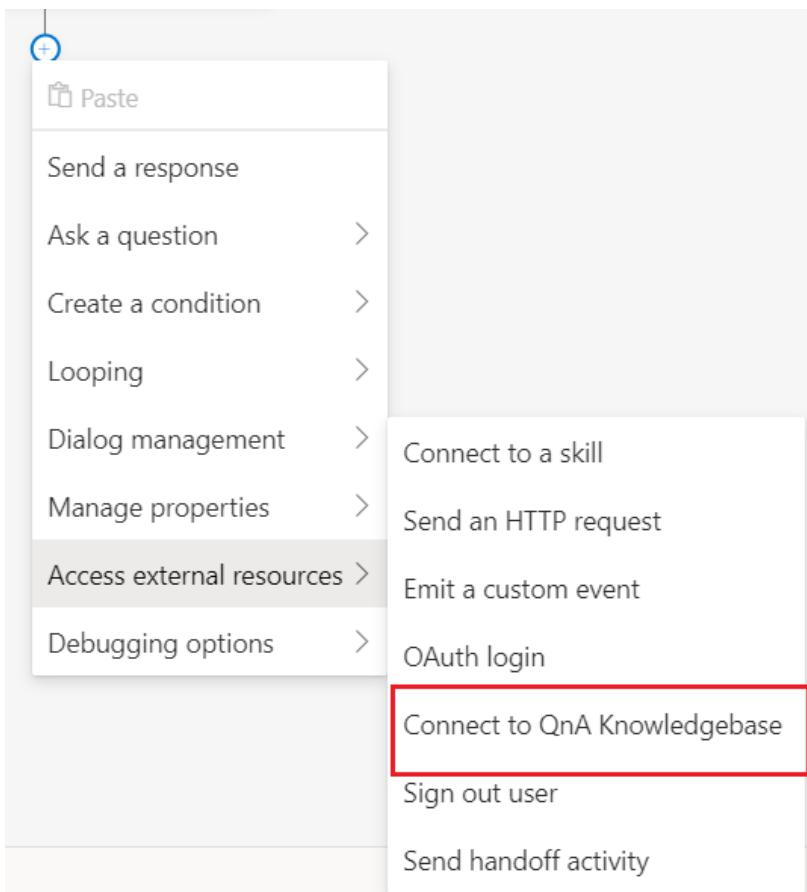
## Prerequisites

- [A basic bot built using Composer](#)
- [A QnA Maker knowledge base](#)

## Add QnA Maker integration

- [Composer v2.x](#)
- [Composer v1.x](#)

To access the **Connect to QnA Knowledgebase** action, you need to select + under the node you want to add the QnA knowledge base and then select **Connect to QnAKnowledgeBase** from the **Access external resources** action menu.



## Review settings

- [Composer v2.x](#)
- [Composer v1.x](#)

Review the QnA Maker settings panel after selecting the QnA Maker dialog. While you can edit settings in the panel, a security best practice is to edit security-related settings (such as the **endpoint key**, **knowledge base**, ID, and **hostname**) from the **Settings** menu. This menu writes the values to the `appsettings.json` file and persists the values in the browser session. If you edit the settings from the QnA Maker settings panel, these settings are less secure because they are written to the dialog file.

## Required and optional settings

The following settings configure the bot's integration with QnA Maker.

REQUIRED	SETTING	INFORMATION
Required	Knowledge base ID - provided by <code>appsettings.json</code> as <code>settings.qna.knowledgebaseid</code>	<p><i>You shouldn't need to provide this value. QnA Maker portal's <b>Settings</b> for the knowledge base, after the knowledge base is published. For example,</i></p> <div style="border: 1px solid #ccc; padding: 5px; width: fit-content;">12345678-MMM-M-ZZZZ-AAAA-123456789012</div>

REQUIRED	SETTING	INFORMATION
Required	Endpoint key - provided by <code>appsettings.json</code> as <code>settings.qna.endpointkey</code>	<p><i>You shouldn't need to provide this value.</i> QnA Maker portal's <b>Settings</b> for the knowledge base, after the knowledge base is published. For example,</p> <div style="border: 1px solid black; padding: 2px; display: inline-block;">12345678-AAAA-BBBB-CCCC-123456789012</div>
Required	Hostname - provided by <code>appsettings.json</code> as <code>settings.qna.hostname</code>	<p><i>You shouldn't need to provide this value.</i> QnA Maker portal's <b>Settings</b> for the knowledge base, after the knowledge base is published. For example,</p> <div style="border: 1px solid black; padding: 2px; display: inline-block;"><a href="https://qnamakerv1111.azurewebsites.net/qnamaker">https://qnamakerv1111.azurewebsites.net/qnamaker</a></div>
Optional	Fallback answer	This answer is specific to this bot and is not pulled for the QnA Maker service's match for no answer. For example, <div style="border: 1px solid black; padding: 2px; display: inline-block;">Answer not found in kb.</div>
Required	Threshold	This answer is a floating point number such as <div style="border: 1px solid black; padding: 2px; display: inline-block;">0.3</div> indicating 30% or better.
Optional	Active learning card title	Text to display to user before providing follow-up prompts, for example: <div style="border: 1px solid black; padding: 2px; display: inline-block;">Did you mean: .</div>
Optional	Card no match text	Text to display as a card to the user at the end of the list of follow-up prompts to indicate none of the prompts match the user's need. For example: <div style="border: 1px solid black; padding: 2px; display: inline-block;">None of the above.</div>
Optional	Card no match response	Text to display as a card to the user as a response to the user selecting the card indicating none of the follow-up prompts matched the user's need. For example: <div style="border: 1px solid black; padding: 2px; display: inline-block;">Thanks for the feedback.</div>

## Edit settings

- [Composer v2.x](#)
- [Composer v1.x](#)

Securely editing the QnA Maker settings should be completed using **Configure -> Advanced Settings View (json)**.

```

{
  "customFunctions": [],
  "defaultLanguage": "en-us",
  "defaultLocale": "en-us",
  "importedLibraries": [],
  "languages": [
    "en-us"
  ],
  "logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft": "Warning",
      "Microsoft.Hosting.Lifetime": "Information"
    }
  },
  ...
}

```

1. Select **Configure** in the side menu. This provides the ability to edit the settings of your bot project.
2. On the Bot management and configurations page, toggle **Advanced Settings View (json)** to edit the settings page in JSON format.
3. Edit the values for the `knowledgebaseid`, `qnaRegion`, `endpointKey`, and the `hostname`. The endpoint key and host name are available from the QnA Maker portal's **Publish** page.

#### TIP

You can get these values from the [QnA Maker](#) service by selecting the **View Code** for the desired knowledge base while in the **My knowledge bases** view.

## Knowledge base limits

You can use **Connect to a QnAKnowledgeBase** Action to connect to only one knowledge base per dialog.

If your knowledge bases are domain agnostic and your scenario does not require you to keep them as separate knowledge bases, you can merge them to create one knowledge base and use **Connect to QnAKnowledgeBase** Action to build your dialog.

If your knowledge bases have content from different domains and your scenario requires you to connect multiple knowledge bases and show the single answer from the knowledge base with higher confidence score to the end user, use the **Send an HTTP request** Action to make two HTTP calls to two published knowledge bases, manipulate the response payload to compare the confidence scores and decide which answer should be shown to the end user.

## Bots with Language Understanding (LUIS) and QnA Maker

Composer allows you to build bots that contain both QnA Maker and LUIS dialogs. A best practice is to set the confidence threshold for LUIS intent prediction and trigger QnA Maker through **Intent** event.

# Create a QnA Maker knowledge base

6/18/2021 • 14 minutes to read

**APPLIES TO:** Composer v1.x and v2.x

You can create, edit, and publish QnA Maker knowledge bases in Bot Framework Composer, in addition to the existing LUIS integration for language understanding. This article shows the basic steps to [create a QnA knowledge base](#), [edit a knowledge base](#), [publish a knowledge base](#) to a QnA Maker service, and [test the published knowledge base](#), all within the Composer environment.

## NOTE

If you are looking for how to add a QnA Maker knowledge base to your bot, read [Add a QnA Maker knowledge base to your bot](#).

## Prerequisites

- [Composer v2.x](#)
- [Composer v1.x](#)
- A subscription to [Microsoft Azure](#), or an administrator with a subscription.
- [Bot Framework Composer](#).

## IMPORTANT

If you [built Composer from source](#), you need to run a command before you can create a QnA Maker knowledge base in Composer. Before running `yarn startall` to start your Composer:

- On Windows, `set QNA_SUBSCRIPTION_KEY=<Your_QnA_Subscription_Key>`
- On macOS or Linux, `export QNA_SUBSCRIPTION_KEY=<Your_QnA_Subscription_Key>`

If you are using the desktop application version of Composer, this step is not necessary.

## About QnA Maker

[QnA Maker](#) is a cloud-based natural language processing service that creates a natural conversational layer over your data. QnA Maker is especially useful when you have static information to manage in your bot. The static question-and-answer pairs are referred to as a [QnA Maker knowledge base](#), which the QnA Maker service uses to process the questions and respond with the matching answers.

## Create or access your QnA Maker resource in Azure Cognitive Services

- [Composer v2.x](#)
- [Composer v1.x](#)

1. Open the [Configure](#) page in Composer. Then select the [Development resources](#), and scroll down to [Azure QnA Maker](#).
2. Select [Set up QnA Maker](#).

## Azure QnA Maker

QnA Maker is an Azure Cognitive services that can extract question-and-answer pairs from a website FAQ. [Learn more](#). Use an existing key from Azure or create a new key. [Learn more](#).

QnA Maker Subscription key \* [?](#)

Type subscription key

 QnA Maker Subscription key is required to start your bot locally, and publish

**Set up QnA Maker**

There are three ways to add a QnA Maker resources in Composer:

- [Use existing resources](#): Select this if you already have a QnA Maker resource.
- [Create and configure new Azure resources](#): Select this if you have access to Azure and need to create a new QnA Maker resource.
- [Generate instruction for Azure administrator](#): Select this if you are working with an Azure administrator. They will provision your QnA Maker resource.

### Set up QnA Maker

X

Use Azure QnA Maker to create a simple question-and-answer bot from a website FAQ. [Learn more](#)

- Use existing resources  
 Create and configure new Azure resources  
 Generate instructions for Azure administrator

Next

Cancel

#### Use existing resources

Select this option if you already have a QnA Maker resource.

1. Select **User existing resources** and select **Next** on the bottom right. Then sign into Azure when prompted.
2. Select the **Azure Directory**, **Azure subscription**, and **QnA Maker resource name** for your bot. Then

select **Next** on the bottom.

## Select QnA Maker resources X

Select your Azure directory, then choose the subscription where your existing QnA Maker resource is located.[Learn more](#)

Azure directory \*



Azure subscription \*



QnA Maker resource name



Back

Next

Cancel

3. Composer will add the keys from your existing resource to your bot. The following window will appear once your key has been successfully added:

## Select QnA Maker resources

X

The following QnA Maker keys have been successfully added to your bot project:

Key	[REDACTED]
Region	[REDACTED]

**Done**

4. Press **Done** to finish. You now can start adding QnA capabilities to your bot.

### Create and configure new Azure resources

Select this option if you have access to Azure and need to create a new QnA Maker resource

1. Select **Create and configure new Azure resources** and select **Next** on the bottom right. Then sign into Azure when prompted.
2. Select the **Azure Directory** and **Azure subscription** for your new resource. Then select **Next** on the

## Create QnA Maker resources

X

Select your Azure directory, then choose the subscription where you'd like your new QnA Maker resource.[Learn more](#)

Azure directory \*

Azure subscription \*

Back

Next

Cancel

bottom.

3. On the next window, select a **Resource group**, **Region**, and **Pricing tier**. Also enter the **QnA Maker resource name**. Then select **Next**.

## Create QnA Maker resources

X

Select the resource group and region in which your QnA Maker service will be created.

Azure resource group

Region \*

 West US

QnA Maker resource name \*

Pricing tier \*

 Free

Back

Next

Cancel

4. Press Done to finish. You now can start adding QnA capabilities to your bot.

### Generate instruction for Azure administrator

Select this option if you are working with an Azure administrator. They will provision your QnA Maker resource

1. Select **Generate instruction for Azure administrator**.
2. Copy the instructions on the next page and send them to your administrator. They will provision the resources you need. Then press **Okay**.
3. Your administrator will generate a JSON for you. On the **Configure** page in Composer, toggle on the **Advanced settings view (json)**.



Advanced Settings View (json)

4. Replace the existing JSON with the one from your administrator. You now can start adding QnA capabilities to your bot.

## Create and add a knowledge base

- [Composer v2.x](#)
- [Composer v1.x](#)

There are three places to create a knowledge base in Composer, explained in the sections below:

- After creating a bot with the [Core Bot with QnA template](#).(#add-knowledge-base-from-the-authoring-canvas).
- From the [authoring canvas](#).
- From the [Knowledge base page](#)

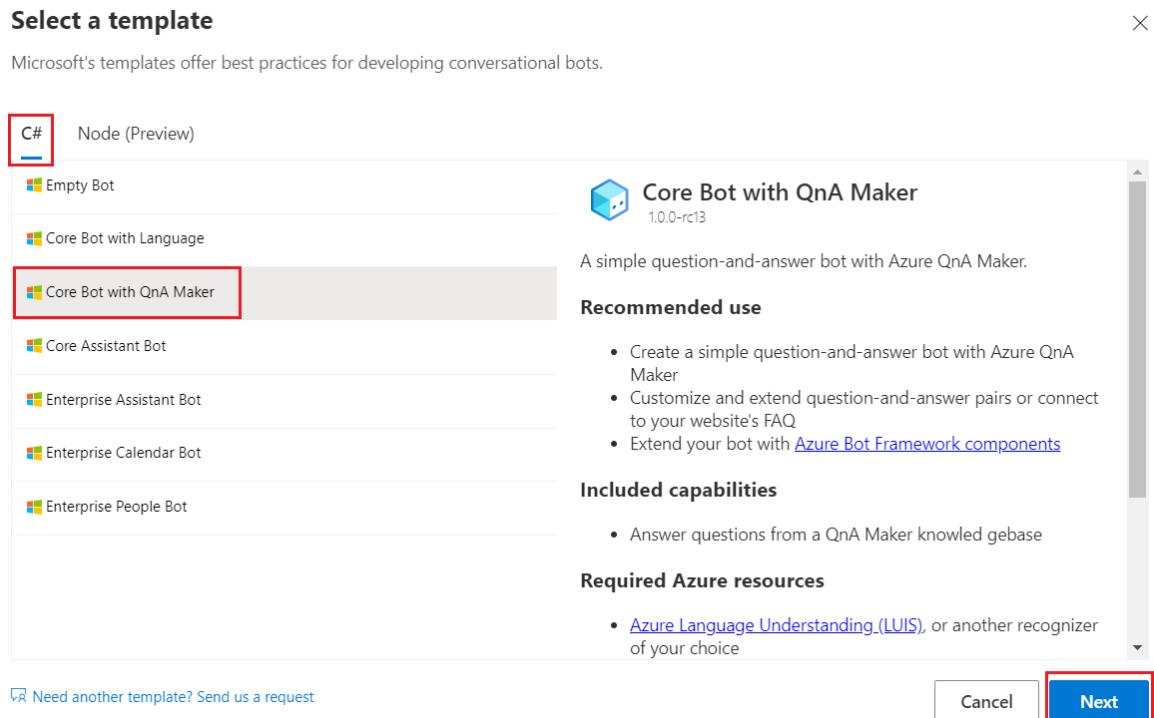
With all options, you can either:

- **Add a knowledge base from a URL:** Select this option if you already have a URL of a file that contains QnA pairs, like  
`https://download.microsoft.com/download/2/9/B/29B20383-302C-4517-A006-B0186F04BE28/surface-pro-4-user-guide-EN.pdf`
- **Create a custom knowledge base:** Select this option if you want to manually add question and answer pairs.

#### Create a Core Bot with QnA Maker

Select this option if you want to create a new bot with QnA capabilities.

1. Open Composer.
2. Select **+ Create** on the homepage. The templates window will open.
3. Select the **Core Bot with QnA Maker** template under the C# section. Then select **Next**.



4. Enter the **Name** of your bot project. Set the **Runtime type** to **Azure Web App**, and choose a location for your bot project. Then select **Next** on the bottom right.

## Create a bot project

X

Specify a name, description, and location for your new bot project.

Name \*

Runtime type

Azure Web App

Location

[Create new folder](#)

	Name	Date modified
	..	a few seconds ago
	[REDACTED]	25 days ago
	[REDACTED]	3 days ago

[Cancel](#)

[Next](#)

5. It will take a few moments for Composer to create your bot project. After it's done, screen will appear with two options, described in the sections below Choose the option that's best for your bot project:

- [Add a knowledge base from a URL](#)
- [Create a custom knowledge base](#)

### Add QnA Maker knowledge base

Use Azure QnA Maker to extract question-and-answer pairs from an online FAQ. [Learn more](#)

Knowledge base name \*

Type a name for this knowledge base

FAQ website (source)

English (United States) \*

Type or paste URL

Enable multi-turn extraction

[Create custom knowledge base](#)

[Cancel](#)

[Create](#)

### Add knowledge base from the authoring canvas

Select this option if you already have a bot and want to add QnA capabilities to it.

1. Open a bot project in Composer.
2. Select the three dots next to a dialog. Then select + Add QnA Maker knowledge base.
3. On the next screen you have two options, described in the sections below. Choose the option that's best for your bot project:
  - [Add a knowledge base from a URL](#)
  - [Create a custom knowledge base](#):

## Add QnA Maker knowledge base

Use Azure QnA Maker to extract question-and-answer pairs from an online FAQ. [Learn more](#)

Knowledge base name \*

Type a name for this knowledge base

FAQ website (source)

English (United States) \*

Type or paste URL

Enable multi-turn extraction

[Create custom knowledge base](#)

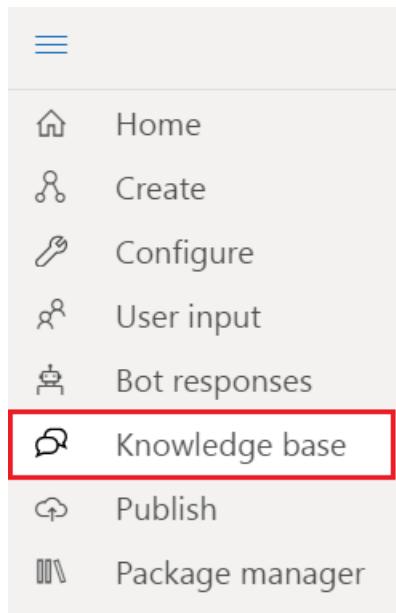
[Cancel](#)

[Create](#)

### Add knowledge base from the Knowledge (QnA) page

Select this option if you already have a bot and want to add QnA capabilities to it.

1. Open a bot project in Composer.
2. Select the **Knowledge base** icon on the left menu.



3. Composer will open the **Knowledge (QnA)** page. To add a knowledge base, you can either select the **Create new KB** button on the bottom, or you can select the three dots next to your bot and then **+ Add QnA Maker knowledge base**. They will take you to the same place.

Create a knowledge base from scratch or import knowledge from a URL or PDF files

**Create new KB**

- On the next screen you have two options, described in the sections below. Choose the option that's best for your bot project:

- [Add a knowledge base from a URL](#)
- [Create a custom knowledge base](#):

### Add QnA Maker knowledge base

Use Azure QnA Maker to extract question-and-answer pairs from an online FAQ. [Learn more](#)

**Knowledge base name \***

Type a name for this knowledge base

**FAQ website (source)**

**English (United States) \***

Type or paste URL

Enable multi-turn extraction

**Create custom knowledge base**

**Cancel**

**Create**

#### Add knowledge base from a URL

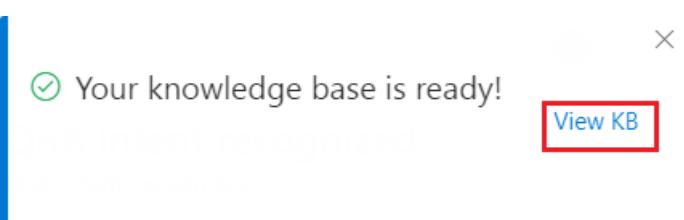
Select this option if you already have a URL of a file with QnA pairs, like

```
https://download.microsoft.com/download/2/9/B/29B20383-302C-4517-A006-B0186F04BE28/surface-pro-4-user-guide-EN.pdf
```

- Enter a **Knowledge base name** and **FAQ website (source)**. Also select if you want to **Enable multi-turn extraction**.
- Select **Create** to create the knowledge base.

#### Create a custom knowledge base

- Select **Create custom knowledge base**.
- On the next page, enter a **Knowledge base name**. Then select **Create**.
- Once Composer has created the knowledge base, a message will appear on the top right indicating so. Select **View KB** to view your new knowledge base.



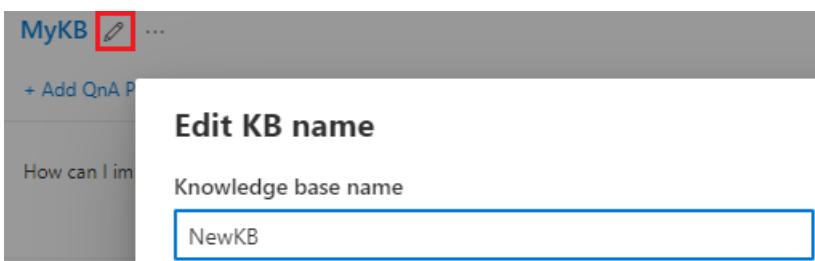
4. You will be taken to the Knowledge base page. Select + Add QnA Pair to add a question and answer pair.

Question	Answer
▼ test-kb-1  ... <a href="#">+ Add QnA Pair</a>	

## Edit a knowledge base

- [Composer v2.x](#)
- [Composer v1.x](#)

1. Select **Knowledge base** from the Composer menu and navigate to the **Knowledge (QnA)** page.
2. Select a dialog from the navigation pane. You can see all the knowledge bases you created in the selected dialog.
3. Select a knowledge base you want to edit.
4. **To edit a knowledge base name:** Select the pencil icon to the right of the name to edit your knowledge base name.

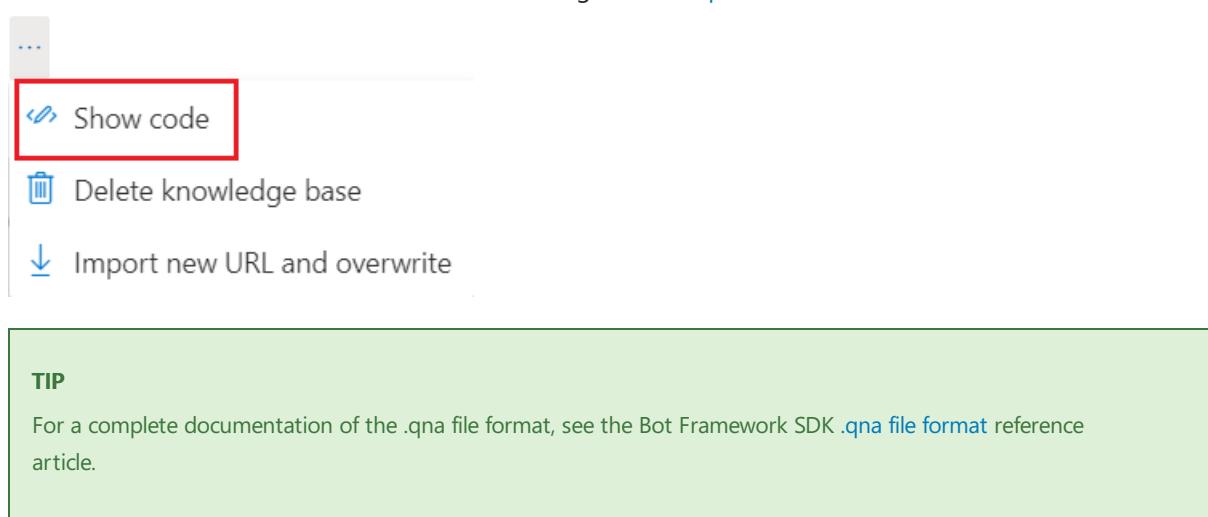


5. **To add a question-and-answer pair:**
  - a. Select + Add QnA Pair under your desired knowledge base
  - b. Enter questions and answers in the **Question** and **Answer** columns respectively.
6. **To include alternative phrases of a question:**
  - a. Hover over a question and answer pair and then select + Add alternative phrasing from the **Question** column.
  - b. Enter your desired alternative phrases of the question.
7. **To delete a question-and-answer pair:**
  - a. Hover over the pair and select the delete icon on the right.



8. **To edit the .qna file of a knowledge base:**

- a. Select the three-dot icon right to the knowledge base name.
- b. Select **Show code** to view and edit the knowledge base in [.qna file format](#).



...  
**Show code**  
Delete knowledge base  
Import new URL and overwrite

**TIP**  
For a complete documentation of the .qna file format, see the Bot Framework SDK [.qna file format](#) reference article.

- c. Select the left arrow on top to exit from the edit mode.



```
< test-kb-1

> !# @source.url=https://download.microsoft.com/
    |   |   |
    |   |   > !# @source.multiTurn=false
    |   |   > # QnA pairs
    |
    > !# @qna.pair.source = onlineFile.pdf

<a id = "1"></a>

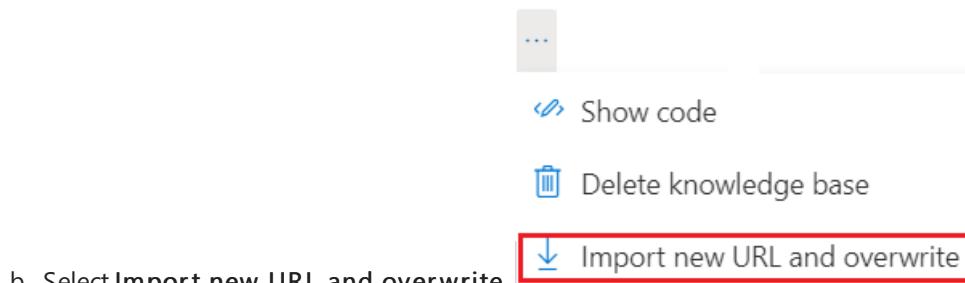
▽ # ? With Windows 10

    ^``markdown
    **With Windows 10**


    Published: September 2016
```

#### 9. To import a new URL and overwrite:

- a. Select the three-dot icon right to the knowledge base name.



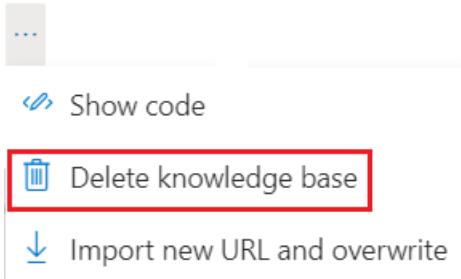
...  
Show code  
Delete knowledge base  
**Import new URL and overwrite**

- b. Select **Import new URL and overwrite**.
- c. Enter the URL of the file to overwrite the existing knowledge base in **FAQ website (Source)**. Then select **Done** on the bottom right.

## Delete a knowledge base

- [Composer v2.x](#)
- [Composer v1.x](#)

1. Select **Knowledge base** from Composer menu and navigate to the **Knowledge (QnA)** page.
2. On the **Knowledge (QnA)** page, select the dialog from the navigation pane that contains the knowledge base you want to delete.
3. Select the knowledge base you want to delete, then select the three-dot icon right to the knowledge base name. Select **Delete knowledge base** to delete the knowledge base.



## Publish knowledge base

- [Composer v2.x](#)
- [Composer v1.x](#)

After you have [created your QnA Maker resource in Azure](#) and finished [creating the QnA Maker knowledge base](#), you can proceed to publish it to the [QnA Maker](#) service.

### NOTE

When you create a bot using the QnA Maker template, the information for your QnA Maker database is imported into Composer. When you follow the steps here to publish your knowledge base, Composer uses that to create a database for you in the [QnA Maker](#) service. If you have an existing bot and wish to add an existing QnA Maker database, you will need to follow the steps given in the [Add a QnA Maker knowledge base to your bot](#) article.

1. Select **Configure** from the Composer menu. Then select **Development resources** and scroll down to **Azure QnA Maker**.
2. Enter your **QnA Maker Subscription key**, shown in the screenshot below:

### Azure QnA Maker

QnA Maker is an Azure Cognitive services that can extract question-and-answer pairs from a website FAQ. [Learn more](#). Use an existing key from Azure or create a new key. [Learn more](#).

QnA Maker Subscription key [?](#)

**Set up QnA Maker**

3. Go back to the **Create page**. Select **Start bot**.
4. After publishing the knowledge base successfully, you can navigate to your [QnA Maker portal](#) to view the knowledge base just published.

## Test

- [Composer v2.x](#)
- [Composer v1.x](#)

1. Select **Start bot** in the top right. Then select **Open Web Chat**.
2. Type a question in Web Chat to have the bot respond with the matching answer from the knowledge base.

 Restart Conversation - new user ID |   

Welcome to your bot.

### About this guide

Just now

#### About this guide

This guide is designed to get you up and running with the key features of your new Surface Pro 4 and Surface Pen. You'll find lots more info online at [Surface.com](http://Surface.com): Go to <http://www.microsoft.com/surface/support>. The information on [Surface.com](http://Surface.com) is also available through the Surface app on your Surface Pro 4. For more info, see The Surface app later in this guide.



Type your message



## Additional information

- [Manage QnA Maker resources.](#)
- [Add a QnA Maker knowledge base to your bot in Composer.](#)
- [.qna file format.](#)

# Multilingual support

6/18/2021 • 3 minutes to read

**APPLIES TO:** Composer v1.x and v2.x

Bot Framework Composer provides multilingual support for bot development in different languages, with English as the default. With simple changes to the settings in Composer, you can author [.lg](#) and [.lu](#) files in your preferred language, and give your bot the ability to talk to users in different languages.

This article shows how to build a [basic bot](#) in English ([en-us](#)) and walks through the process to author the bot in Chinese ([zh-cn](#)).

## NOTE

If your bot has LUIS or QnA integrations, you'll also need to consider additional constraints of [LUIS supported languages](#) and [QnA supported languages](#).

## Prerequisites

- [Install Composer](#)

## How does multilingual support work?

Composer creates copies of your source language files so that you can add manual translation. If you build a bot with just a single dialog, you can access your bot's source code (for example, in the directory:

`C:\Users\UserName\Documents\Composer\CoolBot` ) and see the following file structure:

```
/coolbot
  coolbot.dialog
  /language-generation
    /en-us
      common.en-us.lg
      coolbot.en-us.lg
    /language-understanding
      /en-us
        coolbot.en-us.lu
```

When [adding languages](#), Composer creates copies of the language files. For example, if you add Chinese ([zh-cn](#)), your bot's file structure will look like the following:

```
/coolbot
  coolbot.dialog
  /language-generation
    /en-us
      common.en-us.lg
      coolbot.en-us.lg
    /zh-cn
      common.zh-cn.lg
      coolbot.zh-cn.lg
  /language-understanding
    /en-us
      coolbot.en-us.lu
    /zh-cn
      coolbot.zh-cn.lu
```

#### NOTE

Both `en-us` and `zh-cn` are `locales`. A `locale` is a set of parameters that defines the user's language, region and any special variant preferences that the user wants to see in their user interface.

After adding the languages, you can add manual translations with your source language files as reference. When you are done with the translation process, you must set the `locale` in the **Default language** field. This tells your bot in which language it must talk to the users. However, this `locale` setting will be overwritten by the client's (for example, Bot Framework Emulator) `locale` setting.

In the next sections, we will use a basic bot in English and walk through the steps to author bots in multiple languages.

## Build a basic bot

To show how multilingual support works in Composer, we build a simple bot in English for demo purposes. If you already have a bot, you can skip to the [update language settings](#) section.

This bot named **MultilangBot** consists of a main dialog called **MultilangBot**, a prebuilt **Greeting** trigger, and an **Intent recognized** (LUIS) trigger named **Joke** with the following trigger phrases:

- Hey bot, tell me a joke.
- Tell me something funny.
- I want to hear something interesting.

The **Send a response** action added to the trigger node is defined as the following response:

- Here is a joke for you: Parallel lines have so much in common. It's a shame they'll never meet.

The **MultilangBot** is shown as follows:

When test the bot in the Emulator you get the following responses:

## Update language settings

The first step to author bots in other languages is to add languages. You can add as many languages as you need in the **Project Settings** page.

1. In the **Project Settings** page, select **Manage bot languages** from the **Bot language** section.
2. In the pop-up window, there are three settings that need to be updated:
  - a. The first setting is the language to copy resources from. You can leave this as **English**.
  - b. The second setting is the preferred bot authoring languages. You can select multiple languages. Let's select **Chinese (Simplified, China)**. Hover your mouse over the selection you'll see the `locale`.
  - c. The final setting is a check box. When checked, your selected language will be the active authoring language. In cases you have multiple sections in the second part, the first selected language becomes the active authoring language. Let's check the box and select **Done**.

This language will be copied and used as the basis (and fallback language) for the translation.

English (United States)

To which language will you be translating your bot?

Chinese

- Chinese
- Chinese (Simplified)
- Chinese (Simplified, China) zh-cn
- Chinese (Simplified, Singapore)

- When done, switch to the newly created language and start the (manual) translation process.

You'll then see the language being added to the following language list:

English (United States) DEFAULT LANGUAGE

Chinese (Simplified, China)

[Manage bot languages](#)

You'll also see the locale changed from `en-us` to `zh-cn` in the Composer title bar.

Bot Framework Composer | Demo (zh-cn)

## Author your bot in the selected language

When you are done updating the language settings, you can start authoring your bot in your selected authoring language.

1. In the **Bot Responses** page, select **Show code** on the upper right corner of the screen to manually translate the responses in your selected authoring language.

Bot Responses

Chinese (Simplified, China)

```
[import](common/lg)
# SendActivity_Welcome
- ${WelcomeUser()}

# SendActivity_L9ImDy()
- 这是给你准备的笑话：平行直线有很多共同点，可惜它们永远无法相见。
```

English (United States)

```
[import](common/lg)
# SendActivity_Welcome
- ${WelcomeUser()}

# SendActivity_L9ImDy()
- Here is a joke for you: Parallel lines have so much in common.
```

### NOTE

Make sure you select all the required dialogs and manually translate all responses.

2. In the **User Input** page, select **Show code** on the upper right corner of the screen to manually translate

the user input in your selected authoring language.

## User Input

[Hide code](#)

The screenshot shows the Bot Framework Emulator interface with two parallel conversations. On the left, under the 'MultilingualBot' section, there are dropdown menus for 'Chinese (Simplified, C...' and 'English (United States)'. Below these are two separate conversation windows. The left window (Chinese) contains a '# Joke' card with three items: '- 嘿，讲个笑话', '- 给我说点有趣的东西', and '- 我想听点有意思的'. The right window (English) also contains a '# Joke' card with three corresponding items: '- Hey bot, tell me a joke.', '- Tell me something funny.', and '- I want to hear something interesting.'.

### NOTE

Make sure you select all required dialogs and add manual translations for all user input.

## Test

Before testing the bot, set the added language as your default language.

### Bot language

List of languages that bot will be able to understand (User input) and respond to (Bot responses). To make this bot available in other languages, click 'Manage bot languages' to create a copy of the default language, and translate the content into the new language.

The screenshot shows the 'Manage bot languages' settings. It lists two languages: 'English (United States)' (DEFAULT LANGUAGE) and 'Chinese (Simplified, China)'. To the right of the language names are two buttons: 'Set it as default language' (which is highlighted with a red box) and 'Remove'.

[Manage bot languages](#)

Testing in Emulator:

The screenshot shows the Bot Framework Emulator interface during a conversation. The user message '你好！今天我能为你做点什么？' is shown in a light gray box. The bot's response '讲个笑话' is shown in a blue box with a timestamp 'A minute ago'. A sidebar on the right provides instructions: 'Click on a log item in the panel below to inspect activity.' and 'You can also inspect the JSON responses from your LUIS and QnA Maker services by selecting a "trace" activity. [Learn More](#)'. At the bottom, there is a message input field 'Type your message' with a paperclip icon and a send button.

Testing in Web Chat:

The screenshot shows the Bot Framework Composer interface. On the left, there's a sidebar with various icons and a navigation tree for the bot. The main workspace displays a workflow diagram. An 'Intent recognized' node (labeled 'Joke') has a downward arrow pointing to a 'Send a response' action. A callout box highlights this action with the text: '这是给你准备的笑话：平行直线有很多共同点，可惜它们永远无法相见。'. To the right, a sidebar titled 'MultilingualBot' shows a recent conversation entry: 'Restart Conversation - new user ID'. The bottom of the screen features a toolbar with icons for search, refresh, and other functions.

# Add user authentication

5/28/2021 • 7 minutes to read

**APPLIES TO:** Composer v1.x and v2.x

In Bot Framework Composer, you can use the **OAuth login** action to enable your bot to access external resources using permissions granted by the end user. This article explains how to use basic OAuth to authenticate your bot with an external service such as GitHub.

**NOTE**

It is not necessary to deploy your bot to Azure for the authentication to work.

## Prerequisites

- Microsoft Azure subscription.
- A basic bot built using Composer.
- Install [ngrok](#).
- A service provider your bot is authenticating with such as [GitHub](#).
- Basic knowledge of [user authentication within a conversation](#).
- [Composer v2.x](#)
- [Composer v1.x](#)

## Create an Azure Bot resource

If you already have an Azure Bot resource, you can skip to the [Configure OAuth connection settings in azure](#) step. Otherwise, follow the steps described in [Create an Azure Bot resource](#).

Make sure to copy and save the Azure Bot resource **app ID** and **password**. You'll need these values in this [Configure the OAuth connection settings in Composer](#) step.

## Configure OAuth connection settings in Azure

The following steps show how to configure OAuth connection settings to allow a bot to access an external service, GitHub in the example.

1. In the Azure portal, go to the Azure Bot resource.
2. In the left pane, in the **Settings** section, select **Configuration**.
3. In the right pane, select **Add OAuth Connection Settings**.

Messaging endpoint

https URL

Enable Streaming Endpoint

Microsoft App ID (Manage) \* [Manage](#)

Schema Transformation Version [Edit](#)

V1.1

Application Insights Instrumentation key [Edit](#)

Application Insights API key [Edit](#)

Application Insights Application ID [Edit](#)

Name	Service Provider
No results	

[Add OAuth Connection Settings](#)

- In the **New Connection Settings**, enter the required information to configure the OAuth connection. Note that the options will differ depending on the service to access. The figure below shows the settings for configuring Github OAuth connection.

#### New Connection Setting x

Name \*  ✓

Service Provider \*

Client id \*

Client secret \*

Scopes

#### NOTE

Record the **Name** of your connection. you will need to enter this value in Composer exactly as it is displayed in this setting.

- Enter the values of **Client ID**, **Client Secret**, and optionally **Scopes**. To obtain these values in this example of GitHub, follow the steps described below.

- Go to [GitHub developer's setting webpage](#) and select **New OAuth App** on the upper right corner. This will redirect you to the GitHub OAuth App registration website where you fill in the values as instructed in the following:

**Application name:** a name you would like to give to your OAuth application, e.g. **Composer**

**Homepage URL:** the full URL to your application homepage, e.g. <http://microsoft.com>

**Authorization callback URL:** the callback URL of your application, e.g.

<https://token.botframework.com/.auth/web/redirect> . Read more [here](#).

- Select **Register application**. Then you will see the **Client ID**, **Client Secret** values generated in the application webpage as shown below.

The screenshot shows the 'Client ID' and 'Client Secret' fields highlighted with red boxes. Below the fields are two buttons: 'Revoke all user tokens' and 'Reset client secret'.

- c. Save the **Client ID** and **Client Secret** values.
6. Go back to Azure and paste the values you saved in the **New Connection Settings**. These values configure the connection between your Azure resource and GitHub.
7. Optionally, enter *user, repo, admin* in **Scopes**. This field specifies the permission you want to grant to the bot.
8. Select **Save**.
9. You can test the connection. In the right pane, select the connection you created, *GitHubLogin* in the example.
10. Select **Test Connection**.

The screenshot shows the 'GitHubLogin' connection settings. The 'Test Connection' button is highlighted with a red box. Other fields shown include Name (GitHubLogin), Service Provider (GitHub), Client id, Client secret, and Scopes.

If the test succeeds, you get the connection token back.

The screenshot shows the test results for the 'GitHubLogin' connection. It displays the token value (redacted) and provides 'Show Token' and 'Copy Token' buttons.

Now, with the **Name**, **Client ID**, **Client Secret**, and **Scopes** of your new OAuth connection setting in Azure, you are ready to configure your bot.

## Configure OAuth connection settings in Composer

1. In the Composer menu, select **Project Settings**.
2. In the right pane, select **Development Resources** tab.
3. At the bottom of the window, in the *Microsoft App ID* section, in the fields of **Microsoft App Id** and **Microsoft App Password**, enter the **app ID** and **app password** values from your Azure Bot registration.

## Microsoft App ID

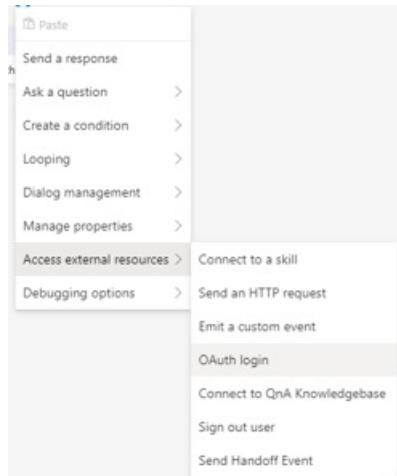
A Microsoft App ID is required for your local Azure resources. If you've created an App ID, enter it here. If you haven't created one, you can do so by clicking the link below.

Microsoft App Id ⓘ

Microsoft App Password ⓘ

4. In the left menu, select **Design** and add the OAuth Login action to your dialog.

- Select **OAuth login** from the **Access external resources** menu.



- The **Connection Name** in the **Properties** panel must be set to the same value you used in Azure for the **Name** of the connection setting.
- You will also need to configure at least the **Text** and **Title** values, which configure the message that will be displayed alongside the login button, as well as the **property** field, which will bind the results of the OAuth action to a variable in your bot's memory.

**OAuth login**

OAuth Input

Collect login information before each request.

[Learn more](#)

Connection name \* ⓘ

abc	GitHubLogin
-----	-------------

Text ⓘ

abc	Please login
-----	--------------

Title ⓘ

abc	Login
-----	-------

Timeout ⓘ

123	900000
-----	--------

Token property ⓘ

abc	dialog.token
-----	--------------

Your bot is now configured to use this OAuth connection!

## Use the OAuth results in your bot

When you launch the bot in the Emulator and trigger the appropriate dialog, the bot will present a login card. Selecting the login button in the card will launch the OAuth process in a new window.



Type your message

You'll be asked to login to whatever external resource you've specified. Once complete, the window will close automatically, and your bot will continue with the dialog.

The results of the OAuth action will now be stored into the property you specified. To reference the user's OAuth token, use `<scope.name>.token` -- so for example, if the OAuth prompt is bound to `dialog.oauth`, the token will be `dialog.oauth.token`.

To use this to access the protected resources, pass the token into any API calls you make with the [HTTP Request](#) action. You can refer to the token value in URL, body or headers of the HTTP request using the normal LG syntax, for example:  `${dialog.oauth.token}` .

## Next

- Learn how to [send an HTTP request and use OAuth](#).

# Send an HTTP request

5/28/2021 • 4 minutes to read

**APPLIES TO:** Composer v1.x and v2.x

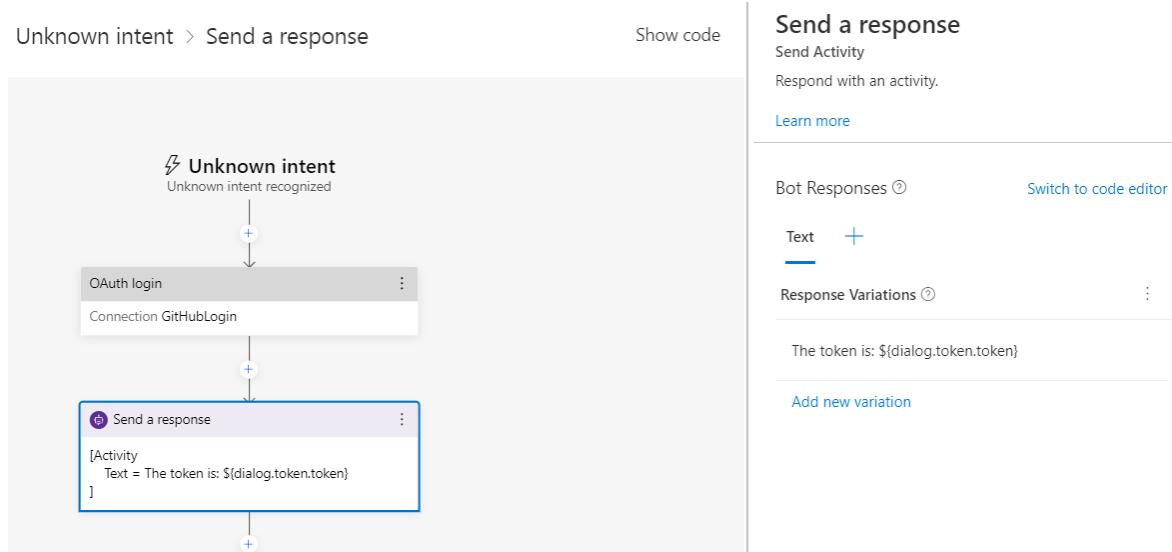
This article will teach you how to send an HTTP request using OAuth for authorization. It's not necessary to deploy your bot to Azure for this to work.

## Prerequisites

- [A basic bot you built using Composer](#)
- A target API for your bot to call
- Basic knowledge of [how to send an HTTP request without OAuth](#)
- Basic knowledge of [how to use OAuth in Composer](#)
- [Composer v2.x](#)
- [Composer v1.x](#)

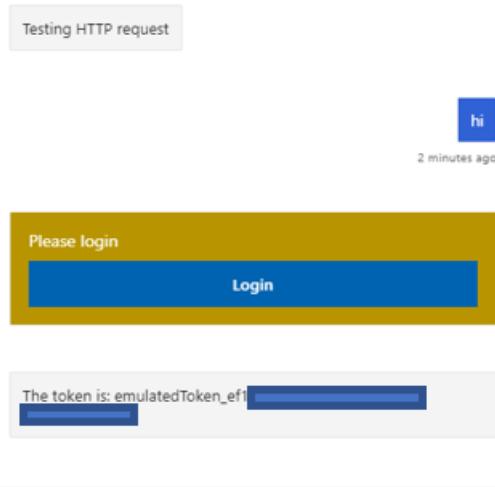
## Set up OAuth in Composer

1. In Composer, create a bot using the *Empty bot* template. Let's call it *testhttp*.
2. Optionally, in the **Greetings** trigger, set the message to *Testing HTTP request*.
3. Follow the steps described in the article [Add user authentication](#) to set up OAuth in your bot to access GitHub. The **Unknown intent** trigger is modified as shown in the figure below.



The **Send a response** message contains `$(dialog.oauth.token)` that is the token returned by GitHub after the connection with the bot has been succeeded.

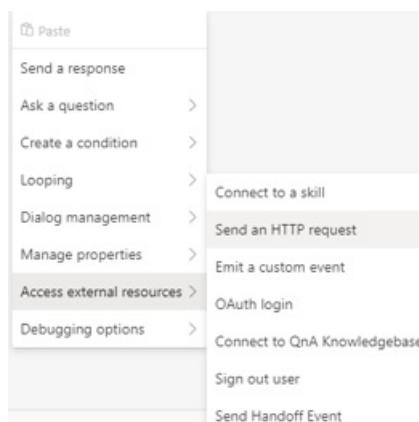
1. Select **Start Bot**, then **Test in Emulator**.
2. In the emulator, enter a short message. The bot will display a **Login** card.
3. Select **Login** and follow the required steps. GitHub will send back the connection token.



Now, with the OAuth setup ready and token successfully obtained, you are ready to add the HTTP request in your bot.

## Add a *Send an HTTP request* action

1. In Composer, select **Design** from the menu on the left.
2. Select the **Unknown intent** trigger.
3. In the canvas, select the + icon, under *Send a response*.
4. From the drop-down menu select **Access external resources** and then **Send an HTTP request**.



5. In the **Properties** panel, set the method to `GET` and set the URL to your target API. For example, a typical GitHub API URL such as `https://api.github.com/users/your-username/orgs`.
6. Add headers to include more info in the request. For example we can add two headers to pass in the authentication values in this request.
  - a. In the first line of header, add `Authorization` in the **Key** field and `bearer ${dialog.token.token}` in the **Value** field. Press Enter.
  - b. In the second line of header, add `User-Agent` in the **Key** field and `Vary` in the **Value** field. Press Enter.
7. Finally, set the **Result property** to `dialog.api_response` and **Response type** to `json`.

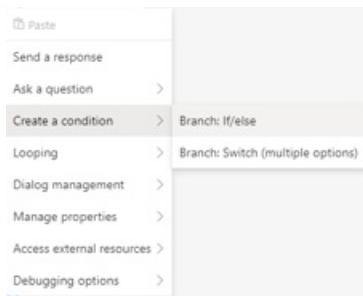
## NOTE

The HTTP action sets the following information in the **Result** property: statusCode, reasonPhrase, content, and headers. Setting the **Result property** to `dialog.api_response` means you can access those values via `dialog.api_response.statusCode`, `dialog.api_response.reasonPhrase`, `dialog.api_response.content` and `dialog.api_response.headers`. If the response is json, it will be a deserialized object available via `dialog.api_response.content`.

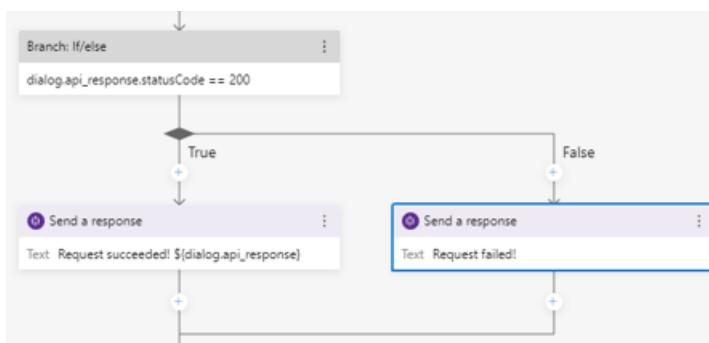
## Test

Add an [If/else branch](#) to test the response of this HTTP request.

1. In the canvas, select the + icon, under *Send an HTTP request*.
2. From the drop-down menu select **Create a condition** and then **Branch:If/else**.



3. In the **Properties** panel, set **Condition** to `dialog.api_response.statusCode == 200`.
4. Add two **Send a response** actions to be fired based on the testing results (true/false) of the condition. This means if `dialog.api_response.statusCode == 200` is evaluated to be `true`, send a response `called with success! ${dialog.api_response}`, else send a response `calling api failed.`



5. Restart your bot and test it in the Emulator. After logging in successfully, you should be able to see the response content of the HTTP request.

# Migrating a bot built with the v4 SDK and waterfall dialogs to Bot Framework Composer

5/20/2021 • 11 minutes to read

**APPLIES TO:** Composer v2.x

Bots built using the Bot Framework v4 SDK, without using Bot Framework Composer, use waterfall dialogs and often have externally managed LUIS models used for Language Understanding and intent recognition. This article will demonstrate some methods that can be used to migrate assets and functionality from an existing bot to a bot built with Composer, avoiding the need to completely re-develop the bot.

This article demonstrates how to migrate the [Core Bot Sample](#) to a new bot project built Composer.

## NOTE

This article demonstrates some of the potential approaches to the migration of a bot using a C# scenario. It is possible that these principles and methods shown will need to be altered or extended depending on the specific bot you are migrating.

## Prerequisites

- A running version of the [Core Bot Sample](#), with a deployed LUIS natural language model. Instructions for building the sample (including deploying the language model) can be found in the sample's [README](#).
- [Install Bot Framework Composer](#)

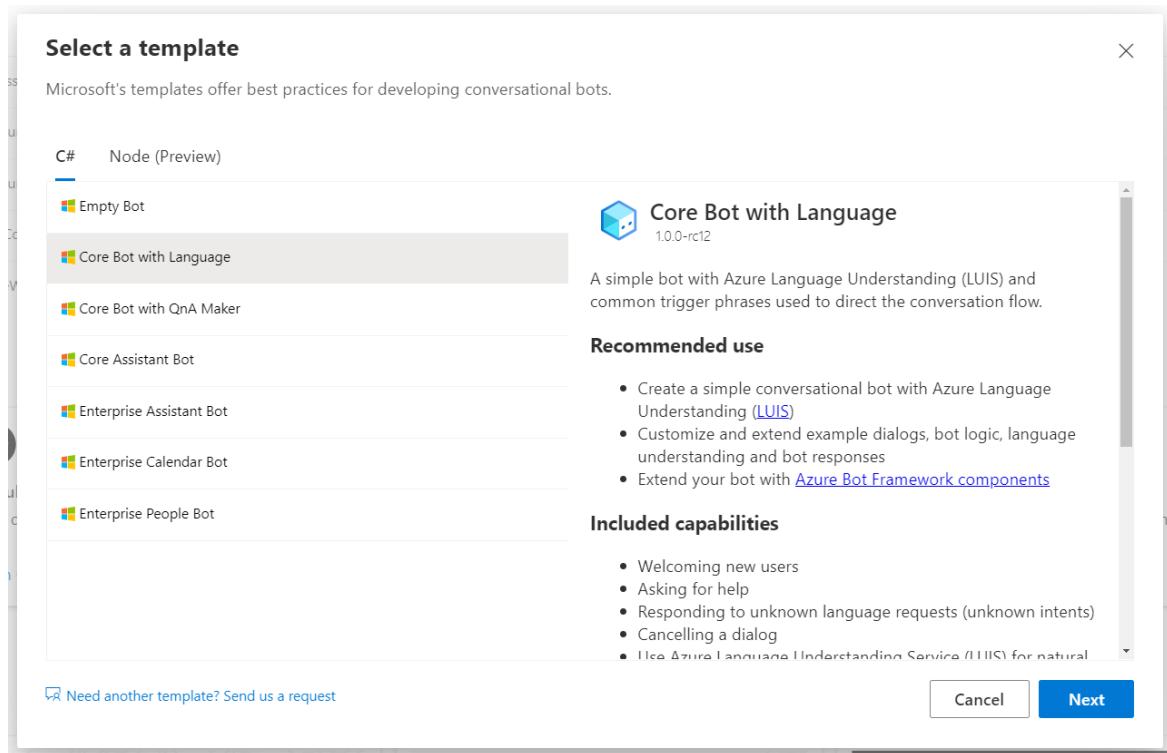
## Create a Composer bot using an appropriate template

When migrating a bot, an important first task is to create a new target bot in Composer and choose a complimentary template to start with. The core bot sample we are migrating has the following capabilities.

- Uses LUIS to implement core AI capabilities.
- Implements a multi-turn conversation using dialogs.
- Handles user interruptions for such things as "help" or "cancel".

This article uses the **Core Bot with Language** template, which has support for Language Understanding and interruption handling.

1. Open Composer.
2. Select **New (+)** on the homepage.
3. Select the **Core Bot with Language** template under the **C#** section. This template creates a bot containing only a root dialog and a initial greeting dialog. Then select **Next**.



4. Fill in the **Name** for the bot as "CoreBotWithLanguage" and the **Description** as "A Core Bot with Language". Then select **Azure Web App** from the **Runtime type**, and choose a location for your bot on your machine.
5. Select **OK**. It will take a few moments for Composer to create your bot from the template.

## Set up Language Understanding

1. Navigate to **Project Settings** in the left hand navigation and then to the **Development resources** tab.
2. Select the **Set up Language Understanding** button. The **Set up Language Understanding** window will appear, shown below:

## Set up Language Understanding

X

To understand natural language input and direct the conversation flow, your bot needs a language understanding service. [Learn more](#)

- Use existing resources
- Create and configure new Azure resources
- Generate instructions for Azure administrator

Next

Cancel

The following sections, on the [How to add LUIS](#) article, explain the different options for setting up language understanding. Below is a link to each set up option and when to choose each:

- [Use existing resources](#): You already have existing Azure resources.
- [Create and configure new Azure resources](#): You don't have existing Azure resources and need to create and configure new ones.
- [Generate instructions for Azure administrator](#): You need an administrator to create Azure resources for you.

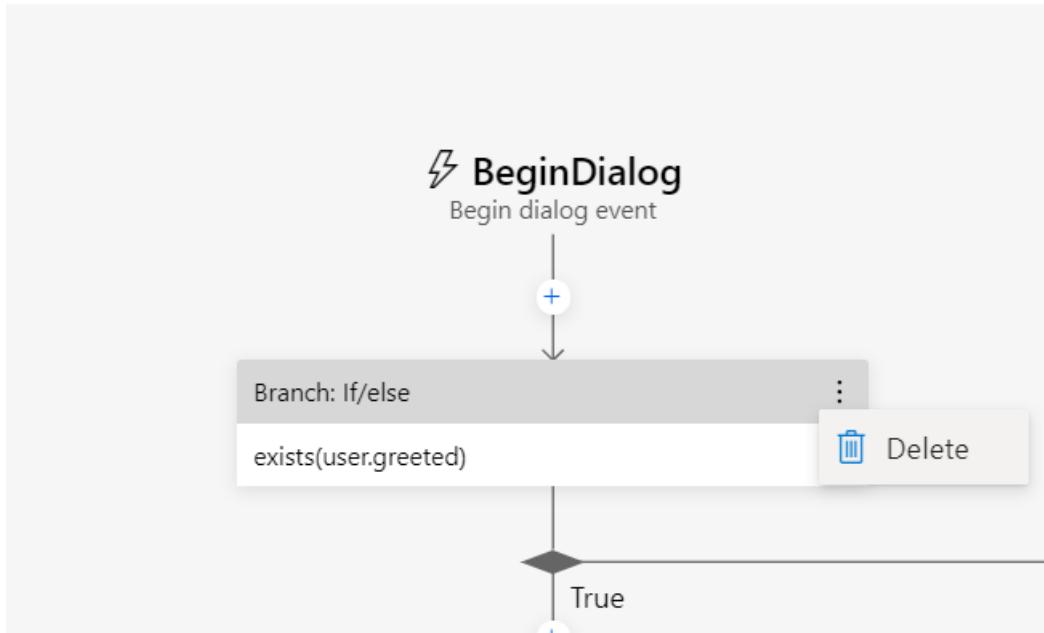
## Update the bot welcome message

The Core Bot Sample sends an Adaptive Card to welcome the user within the `DialogAndWelcomeBot.cs` class. For some bot capabilities such as this, you do not need to migrate the code itself. Instead we can use built in Composer capabilities to send the card for us and the template you have used to create your Composer bot already has a `WelcomeDialog` dialog, which sends a greeting to the user that we can update.

1. In Composer, select the `BeginDialog` trigger within `WelcomeDialog` in the left hand navigation tree.
2. The template we have used has existing actions within the `WelcomeDialog` to greet the user with a different message, depending on if this is their first time engaging with the bot. For this exercise though, we will replicate the existing functionality of the Core Bot Sample and send the same Adaptive Card as a greeting to users, regardless of if they have engaged with the bot previously.

Select the three dots next to the **Branch: If/else** action currently on the canvas and select **Delete** to remove the current welcome actions.

## WelcomeDialog > BeginDialog



3. Use the + button on the canvas and select **Send an response** action.
4. With your new **Send an response** action selected, click the + next to the **Text** tab and select **Attachments**.
5. Click **Add new attachment** and select **Adaptive Card** under the **Create from template** sub menu.
6. Populate the input field for the attachment content using the Adaptive Card JSON from the [welcomeCard.json file](#), found within the [cards](#) folder in Core Bot Sample project.

The screenshot shows the configuration for the 'Send a response' action. It includes fields for 'Attachments' (set to '\$(`{"\$schema": "http://adaptivecards.io/schemas/adaptive-card.json", "type": "AdaptiveCard", "version": "1.0", "body": [ { "type": "Image", "url": "https://adaptivecards.io/images/logo.png" } ]}')') and 'Text' (empty). To the right, there is a preview of the adaptive card JSON:

```
1 - ${`{`  
2 - "$$schema": "http://adaptivecards.io/schemas/  
adaptive-card.json",  
3 - "type": "AdaptiveCard",  
4 - "version": "1.0",  
5 - "body": [  
6 - | {  
7 - | "type": "Image",  
8 - | "url": "https://adaptivecards.io/images/logo.png"  
9 - ]}`}`}
```

Your attachment content should look like this. Note the need to wrap the Adaptive Card JSON with `- $ { }`, as we are using the Language Generation syntax.

```

- ${{{
  "$schema": "http://adaptivecards.io/schemas/adaptive-card.json",
  "type": "AdaptiveCard",
  "version": "1.0",
  "body": [
    {
      "type": "Image",
      "url": "https://encrypted-tbn0.gstatic.com/images?
q=tbn:ANd9GcQtB3AwMUeNoq4gUBGe60cj8kyh3bXa9ZbV7u1fVKQoyKFHdkqu",
      "size": "stretch"
    },
    {
      "type": "TextBlock",
      "spacing": "medium",
      "size": "default",
      "weight": "bolder",
      "text": "Welcome to Bot Framework!",
      "wrap": true,
      "maxLines": 0
    },
    {
      "type": "TextBlock",
      "size": "default",
      "isSubtle": true,
      "text": "Now that you have successfully run your bot, follow the links in this Adaptive Card to expand
your knowledge of Bot Framework.",
      "wrap": true,
      "maxLines": 0
    }
  ],
  "actions": [
    {
      "type": "Action.OpenUrl",
      "title": "Get an overview",
      "url": "https://docs.microsoft.com/en-us/azure/bot-service/?view=azure-bot-service-4.0"
    },
    {
      "type": "Action.OpenUrl",
      "title": "Ask a question",
      "url": "https://stackoverflow.com/questions/tagged/botframework"
    },
    {
      "type": "Action.OpenUrl",
      "title": "Learn how to deploy",
      "url": "https://docs.microsoft.com/en-us/azure/bot-service/bot-builder-howto-deploy-azure?view=azure-
bot-service-4.0"
    }
  ]
}}}

```

## Add a trigger for the Booking intent

The new Bot Framework Composer bot already contains an intent trigger for `Cancel`, along with an associated pre-built dialog to handle the conversation with the user when that trigger is fired, so we do not need to migrate the existing `CancelAndHelpDialog.cs` class. However, we do need to add an intent triggers for the remaining intents used by the Core Bot Sample.

1. Navigate to the LUIS application, within the LUIS.ai portal, you have deployed as part of the Core Bot Sample. Here we can see there are two additional custom intents that we need to add to our new Composer bot. They are `BookFlight` and `GetWeather`.

Name	Examples	Features
BookFlight	14	+ Add feature
Cancel	4	+ Add feature
GetWeather	4	+ Add feature
None	5	+ Add feature

- When creating internet triggers within Composer, we need to provide trigger phrases using the Language Understanding format, containing the definition for any LUIS entities that are used as part of those phrases. To get the appropriate Language Understanding content for the new intents in Composer, you can export the Core Bot Sample LUIS application into .LU format.

To do this, within the LUIS portal, click **Manage** in navigation at the top and then **Versions** in the left hand navigation. Select the latest version of the application available and click **Export, Export as LU**. Save the file to a location on your computer.

Name	Last Modified
1.0 (Active)	5/12/2021
1.1 (Draft)	Not yet

- In Composer, select the three dots next to the root dialog in your bot. Then select **+ Add a trigger**.

- In the trigger creation screen, select **Intent recognized** from the drop-down list.
- Enter `BookFlight` in the **What is the name of this trigger?** field.
- Open the .LU file you exported from LUIS in a text editor and locate the `#BookFlight` intent and

associated utterances. Select and copy the utterances into the **Trigger phrases** field.

7. The trigger phrases you have copied use LUIS entities (e.g. the phrase

book a flight from {@From=new york} has an entity to detect the name of the airport the user will travel from). At the bottom of the exported .LU file you will find the entity definitions used within the model and you need to include any of these entity definitions that are used as part of your new trigger. In this case, the BookFlight trigger uses a composite list entity (to provide the airports from which as a user can fly from / to) and a pre-built datetime entity to detect if the user specifies a date as part of their incoming message.

```
> # PREBUILT Entity definitions

@ prebuilt datetimeV2

> # Phrase list definitions

> # List entities

@ list Airport =
- Paris :
- paris
- cdg
- London :
- london
- lhr
- Berlin :
- berlin
- txl
- New York :
- new york
- jfk
- Seattle :
- seattle
- sea

> # RegEx entities

> # Composite entities

@ composite From = [Airport]
@ composite To = [Airport]
```

Copy the LU definitions of these entities and paste them at the bottom of the **Trigger phrases** field. Then click **Submit** to create your new trigger.

## Create a trigger

What is the type of this trigger?

Intent recognized



What is the name of this trigger?

BookFlight

Trigger phrases

+ Add entity ▾   ➔ Insert entity ▾

- 1 - book a flight
- 2 - book a flight from {@From=new york}
- 3 - book a flight from {@From=seattle}
- 4 - book flight from {@From=london} to {@To=paris} on feb 14th
- 5 - book flight to {@To=berlin} on feb 14th
- 6 - book me a flight from {@From=london} to {@To=paris}
- 7 - flight to {@To=paris}
- 8 - flight to {@To=paris} from {@From=london} on feb 14th



Cancel

Submit

8. Now, repeat step 7, this time creating a trigger called `GetWeather`. This time you only need to copy the trigger phrases from the `#GetWeather` intent within the .LU file as these phrases do not contain any intent usage.

## GetWeather

Intent recognized

Actions to perform when specified intent is recognized.

Trigger phrases [?](#)

+		Add entity	▼	Insert entity	▼
1	-	what's the forecast for this friday?			
2	-	what's the weather like for tomorrow			
3	-	what's the weather like in new york			
4	-	what's the weather like?			

Intent name: [#GetWeather](#)

Your new Composer bot now contains the intent triggers needed to mirror the capabilities in the Core Bot Sample.

The screenshot shows the Microsoft Bot Framework Composer interface. The top navigation bar is blue with the title "CoreWithLanguage". The left sidebar has several icons: Home, People, Tools, Chat, Cloud, and Lists. The main area displays the intent structure:

- CoreWithLanguage
  - CoreWithLanguage
    - Greeting
    - Cancel
    - Help
    - Error occurred
    - Unknown intent
    - GetWeather
    - BookFlight
  - CancelDialog
    - BeginDialog
  - HelpDialog
    - BeginDialog
  - WelcomeDialog
    - BeginDialog

# Make existing BookingDialog dialog available for use in Composer

You now need to make the pre-existing waterfall dialog, from the Core Bot Sample for handling the `BookFlight` intent, available to use within Composer. The process of making an existing waterfall dialog available to use within Bot Framework Composer has the following stages.

- Copy the dialog class file, along with any other files that the dialog depends on, into your Composer bot runtime.
- Create a wrapper dialog class for the dialog, which allows you to use Composer's built in **Begin a new dialog** action to pass in options for your dialog.
- Register the new dialog wrapper in your bot's `Startup.cs` file.

To complete this process for the `BookingDialog` in Core Bot Sample, complete the following steps.

1. Navigate to **Project Settings** in the left hand navigation within Composer and note the **Bot project location** for your Composer bot. Navigate to this location within file explorer and open the Visual Studio .sln file within that folder. This will open your bot runtime project in Visual Studio.
2. Create a new folder within your project in Visual Studio called `MigratedDialogs`.
3. Open file explorer and navigate to the folder where your Core Bot Sample is located. Copy the following files from the Core Bot Sample and paste them into the new `MigratedDialogs` folder within your Composer bot Visual Studio project.
  - `Dialogs\BookingDialog.cs`
  - `Dialogs\DateResolverDialog.cs`
  - `FlightBookingRecognizer.cs`
4. Within your Visual Studio project, open each of the 3 files and update the namespace to match the namespace of your Composer bot. e.g. If you root name space for your Composer bot is `CoreBotWithLanguage`, you would update the namespace for each of the 3 files to be `CoreBotWithLanguage.MigratedDialogs`.
5. Create a new class, within the `MigratedDialogs` folder, called `BookingDialogWrapper.cs` using the code below as the contents.

```
public class BookingDialogWrapper : ComponentDialog
{
    public BookingDialogWrapper()
    {
        this.Dialogs.Add(new BookingDialog() {Id = "BookingDialogWrapper"});
    }

    public override async Task<DialogTurnResult> BeginDialogAsync(DialogContext outerDc, object options
= null,
        CancellationToken cancellationToken = new CancellationToken())
    {
        return await outerDc.BeginDialogAsync("BookingDialog", options.CastTo<BookingDetails>());
    }

    public override async Task<DialogTurnResult> ResumeDialogAsync(DialogContext dc, DialogReason
reason, object result = null, CancellationToken cancellationToken = default(CancellationToken))
    {
        // End the current dialog and return result to parent.
        return await dc.EndDialogAsync(result, cancellationToken).ConfigureAwait(false);
    }
}
```

When migrating other dialogs, you would need to

- Replace the dialog Id (`BookingDialogWrapper` in the example above) with an appropriate Id.
- Replace the dialog Id of the migrated dialog being called (`BookingDialog` in the example above) with the Id as defined within the existing waterfall dialog you are migrating.
- If your dialog uses a specific type for dialog options, you need to ensure that the `options` object is correctly cast to that type (`BookingDetails` in the example above).

6. Open the `Startup.cs` class in your Composer bot runtime solution and register both the waterfall dialog being migrated, as well as the new wrapper dialog, as singletons within the `ConfigureServices` method. The bot runtime will then automatically look for any `Dialog` singletons and add them to the available dialogs collection.

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddControllers().AddNewtonsoftJson();
    services.AddBotRuntime(Configuration);

    services.AddSingleton<Dialog, BookingDialog>();
    services.AddSingleton<Dialog, BookingWaterfallDialog>();
}
```

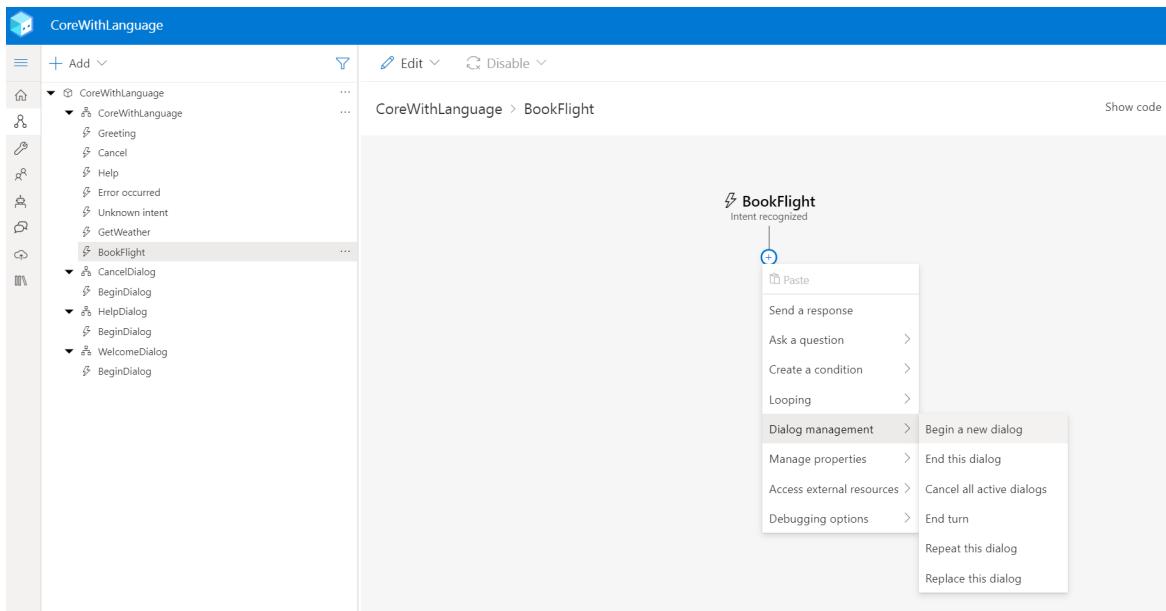
#### TIP

In the example above, because the `BookingDialog` has options that are passed into it, a wrapper was required in order to ensure the proper casting between the JSON object that Composer passes from the **Begin a new dialog** action and the expected type, in this case `BookingDetails`, happens. If you are migrating a waterfall dialog that does not use dialog options, you can skip the step of creating a wrapper class and simply register the waterfall dialog class itself in `Startup.cs` and call it by its Id using the **Begin a new dialog** action.

## Calling the migrated dialog from Bot Framework Composer

Now that you have added the `BookingDialog` class and all of its dependencies into your Composer bot runtime and registered it correctly, you can now call that dialog as easily as you would call another dialog you have authored with Composer.

1. Select the **BookFlight** in the navigation tree of your Composer bot.
2. Use the + button on the canvas and select **Begin a new dialog** action, found under the **Dialog management** sub menu. This will add a new **Begin a new dialog** action to your canvas.



- We now need to tell our new action which dialog to call. With the new action selected, select the **Dialog name** drop down in the property pane and select **Write an expression**. Then enter `BookingDialogWrapper` (the id defined within our new dialog wrapper class) into the text field.

## Start your bot and test the booking dialog

You are now ready to test your bot with your new triggers and migrated dialog.

- Click the **Start bot** button at the top of the Composer window.
- Once the bot has started click **Test in webchat** to open the Composer web chat testing pane.
- Type `book a flight` and send the message. The bot should then correctly fire the `BookFlight` intent trigger, start the `BookingDialogWrapper` and prompt you with the first question `Where would you like to travel to?`.

book a flight

Just now

Where would you like to travel to?

seattle

Just now

Where are you traveling from?

london

Just now

When would you like to travel?

tomorrow

Just now

Please confirm, I have you traveling to: seattle  
from: london on: 2021-05-13. Is this correct?

Yes

No



Type your message



# Upgrade Virtual Assistant bots to use Composer

5/26/2021 • 7 minutes to read

APPLIES TO: Composer v2.x

## Virtual Assistant

The [Virtual Assistant Solution Accelerator](#) built on top of the Bot Framework SDK enabled more sophisticated assistant-style conversational experiences to be built across a wide-range of customers and partners. These [capabilities](#) included core utility intents, multi-language support, speech, skill dispatch, feedback, authentication, telemetry along with deployments scripts, CI/CD and analytics dashboards.

## Integration into Composer and the SDK

The popularity of the Virtual Assistant solution accelerator combined with traction around [Bot Framework Composer](#) for authoring conversational experiences and customer feedback led us to focus investments to deeply integrate these capabilities directly into the core product, resulting in the following new capabilities available as part of Build release in May 2021:

- Infuse core capabilities such as activity routing, interruption, multi-language support, speech and telemetry directly into the SDK and reduce the need for developers to be responsible for implementation and ongoing support.
- Investment into the [adaptive runtime](#) and [bot components](#) to further reduce complexity around bot infrastructure and provide an easy to add new capabilities to a bot.
- A new creation experience within Composer offering a range of [templates](#) with a range of conversational capabilities aligned to popular scenarios. The [Core Assistant template](#) offers similar capabilities to the previous Virtual Assistant template but with more control and configuration around which capabilities you need. Unlike with the prior Virtual Assistant approach, you can start with a simple QnA template based experience and add capabilities progressively over time through our component model to suit your needs.
- A new [Enterprise Assistant template](#) that builds on the Core Assistant template to provide a sophisticated enterprise assistant with [Calendar](#) and [People \(formerly WhoBot\)](#) capabilities.
- A new provisioning experience within Composer that goes beyond deployment script automated in Virtual Assistant to provide Azure provisioning automation based on the needs of your bot directly within Composer along with handoff to an Azure administrator if a developer doesn't have the requisite Azure permissions.
- Virtual Assistant conversational capabilities are now available as part of [templates and packages](#).
- Any bot built using Composer can quickly be made available as a skill unlike the Virtual Assistant approach which required a skill template. Skills can easily be connected to a Composer based bot through the new skill connection experience which automates all the key steps.
- Introduced the [Orchestrator](#) capability to increase skill dispatch quality and is seamlessly enabled when adding a skill to a bot project. This replaces the prior Dispatch capability.
- Provided (via the Bot Builder Community) implementation of human-handoff capabilities for both [LivePerson](#) and [ServiceNow](#) with the latter providing a key capability required by many Enterprise Assistant scenarios.

## Support for Virtual Assistant bots

Following our investments to bring the Virtual Assistant capabilities into the product, our assistant focus moving forward will be on Composer along with the associated templates and packages. However, it's important to note that any Virtual Assistant based bot leverages the v4 Bot Framework SDK—which Composer itself relies on and

is in active development and support—meaning that existing bots are unaffected.

With the introduction of [native skills registration and configuration in Composer](#), the `botskills` CLI will be deprecated on December 31, 2021. This automates various steps to register a skill with a bot created using the Bot Framework SDK, including manifest retrieval, skill configuration, and dispatch training. For additional technical information on how skills are registered by Composer, see [Implement a skill consumer](#) in the Bot Framework SDK documentation.

Similarly, the introduction of [Orchestrator](#) means the [Dispatch CLI](#) will be deprecated on December 31, 2021. As detailed below, existing Bot Framework SDK and Virtual Assistant bots can make use of Orchestrator in place of Dispatch and benefit from the improvements.

The `Microsoft.Bot.Solutions` package provides foundational capabilities to Virtual Assistant bots such as multi-language and speech support along with message handing. These capabilities are all now part of the SDK and Bot Framework Composer, so `Microsoft.Bot.Solutions` will be deprecated on December 31, 2021. Source code will remain available on the GitHub repo if you wish to make changes, for example, updating to the latest SDK version. In the meantime, we will update this library to the latest SDK for each upcoming release.

We are committed to ensuring that existing customers and partners with existing Bot Framework bots, including those built using the Virtual Assistant solution accelerator, have the ability to benefit from our Composer tooling, whilst enabling existing investments to be taken forwards. You can [open an issue](#) in the GitHub repo with any questions or to report issues.

## Approach 1: migrate to a core assistant bot and connect to existing skills

Through our experience with the Virtual Assistant work, we saw that developers lightly customized the root Virtual Assistant to suit their needs, mostly language generation and core dialogs such as greetings. Skills were then used to implement specific domain logic which were then connected back to the root bot.

For developers who want to benefit from [Bot Framework Composer](#) and reduce technical complexity of their solutions, we recommend following the steps below to create a new root bot using Composer based on the [Core Assistant template](#) and connect your existing skills using Composer. This will enable you to quickly benefit from a new Composer based root bot along with Orchestrator for more robust skills dispatch.

### Create a new root bot

1. Create a new bot using the [Core Assistant bot](#) template to act as your new root bot replacing your current Virtual Assistant based bot.
2. Create a new publishing profile, a link to this is available within the rocket ship menu inside Composer. You can choose to create new resources or import existing resources, choosing the latter will enable you to provide existing resource information that you may have available as part of your existing development or test environment.
3. Test the core template capabilities.

### Add additional QnA Maker knowledge bases

If you have any additional QnA Maker knowledge bases that are used as part in your existing Virtual Assistants and have the content represented in `.qna` file format, you can bring this content into Composer and make use of the integrated QnA authoring experience.

- Select **Knowledge Base** in the Composer menu, then the **Create new KB** button.
- In the **Add QnA Maker knowledge base** dialog box, select the **Create Custom Knowledge base** button, enter a valid knowledge base name then select **Create**.
- Once created, you can select **Show Code** in the upper right side of the screen. Once in the code view, you can paste in the contents from an existing `.qna` file.

- Alternatively, you can use the [Connect to QnA Knowledgebase](#) action to connect to an existing deployed knowledge base that is being managed by other users through the QnA Maker portal directly.

## Customize bot responses

In the **Bot Responses** page in Composer you can migrate any LG customization you have made to the Virtual Assistant, such as greetings, adaptive cards, and so on to the new bot, replacing existing examples provided out of the box.

## Connect to skills

Skills created using the Virtual Assistant Solution Accelerator are standard Bot Framework Skills albeit with some additional helper capabilities as part of the provided template. These skills and their associated skill manifests can be connected directly to bots created in Composer with the skill manifest providing training data for Orchestrator to be trained for skill dispatch.

Follow the steps in [How to connect to a skill](#) for each skill you have connected to your virtual assistant.

## Leverage custom waterfall dialogs in your new root bot

If you have extended your root bot with custom waterfall dialogs which you don't wish to migrate to Composer built dialogs, you can invoke these directly from your newly created Composer bot following the steps outlined in the how to [Migrate a bot to Composer](#) article.

This will enable waterfall dialogs created in the Bot Framework SDK to be invoked directly from Composer triggers and dialogs, alongside Composer built dialogs.

## Custom middleware

The Virtual Assistant solution accelerator includes a number of middleware components that provide support for capabilities such as speech and multi-language support. These capabilities are now built into Composer and are no longer needed. If, you have implemented custom-middleware for capabilities not offered by Composer, you can refer to the [extend with code](#) article to learn for how to register custom bot components, such as middleware, with your newly created bot project.

# Approach 2: extend an existing Virtual-Assistant bot with Composer-built skills

There are a number of productivity and simplification benefits from moving the root bot to Composer. However, if you are need to preserve your SDK-first root bot built on the Virtual Assistant template and leverage Composer for new capabilities, you can do this through making a Composer-based bot available as a skill.

Composer's [Export as a skill](#) feature guides you though the process of selecting the dialogs and LU intents to make available to a caller. This skill can then be registered with your Virtual Assistant bot.

### NOTE

The benefits offered by [Orchestrator](#) will not automatically apply unless you transition to use of Orchestrator from Dispatch using [these SDK-first focused instructions](#).

# Extend Composer with extensions

5/20/2021 • 3 minutes to read

**APPLIES TO:** Composer v1.x and v2.x

[Composer Extensions](#) are JavaScript modules. When loaded into Composer, the module is given access to a set of Composer APIs which can then be used by the extension to provide new functionality to the application. You can extend and customize the behavior of Composer by installing extensions which can hook into the internal mechanisms of Composer and change the way they operate. Extensions can also "listen to" the activity inside Composer and respond to it. In this article you will learn the following:

- [Set up multi-user authentication via extensions.](#)
- [Set up customized storage and make storage user-aware.](#)
- [Add an interactable web view to Composer's left navigation list.](#)
- [Add a custom publishing extension and user experience from Composer.](#)
- [Provide an alternate version of the runtime template.](#)

## Prerequisites

- A basic understanding of [Composer extensions](#).
- A clone of the [Bot Framework Composer GitHub repository](#).

## Set up multi-user authentication via extensions

By default, there is no authentication required to access the Composer application and anyone with the URL will be able to use it. Some mechanisms must be used to secure access to the application and the resources available to it!

This can be achieved using Composer's authentication and identity extension endpoint. Composer has adopted [Passport.js](#) as its primary auth mechanism. As a result, it is possible to use any of the 500+ compatible authentication systems with Composer through only a few lines of code.

In your clone of Composer, add the new extension under the `/extensions` folder. Make sure the `package.json` file is properly configured according to the instructions at the link above. Reload the Composer app, and the new extension should take affect automatically, requiring a login.

Read the API documentation for [authentication and identity extensions](#).

## Set up customized storage and make storage user-aware

By default, Composer will read and write bot projects from the local filesystem. All users have the same access to the filesystem.

To change the storage mechanism, or modify it to consider a user identity provided by an authentication extension, it is necessary to create a new extension that uses the storage extension endpoint.

By providing a custom storage implementation, it is possible to restrict a user's access to content using something like user id. It is also possible to completely replace the storage mechanism used by Composer, for example, using a hosted database instead of the local filesystem.

Read the API documentation for [storage extensions](#).

## Add an interactable web view to Composer's left navigation list

Composer supports adding extensions that render custom user interfaces inside the application and allows these user interface to read Composer's active application state, as well as make various modifications to it. By configuring the extension's `package.json` definition to add itself as a contribution point to the Composer's navigation page, it shows up in the application. An example of this is [here](#).

The full code for Package Manager is [here](#).

## Add a custom publishing extension and user experience from Composer

Similar to adding a page to Composer's left navigation, users can add a custom publishing and deployment experience inside Composer's "Publish" page as its own contribution point. Composer's support of publishing to [Power Virtual Agents](#) is an example of this. For an example of the contribution point, see [a sample publish element](#).

For the full code for Composer's Power Virtual Agents publishing extension, see the `pvaPublish` folder in the BotFramework-Composer repo.

## Provide an alternate version of the runtime template

Sometimes it is necessary to modify the code of the runtime used to operate the bots, for example, to install different packages or bundle new features.

This is possible by modifying the runtime template that is generated with Composer. Composer supports C# .NET and JavaScript (preview) solutions located in the root of the project directory when created.

When a user selects **Start bot** or uses Composer's publishing system, Composer will use this structure as the source of truth of what is deployed. Changes made to the template in this folder will automatically be used.

It is also possible to specify one or more alternate runtimes that can be made available as part of a bot project template, or as an option for users to choose in each bot's settings.

Read the API documentation for [providing runtime templates](#).

# Self-hosted Composer

5/12/2021 • 6 minutes to read

Bot Framework Composer is a visual authoring tool for building conversational AI software. Composer is available as [an open-source project](#). While the primary way it is distributed is as a [bundled desktop application](#), it is possible to use Composer in a variety of ways including as a shared, hosted service.

This article covers an approach to hosting Composer in the cloud as a service. It also covers topics related to customizing and extending the behaviors of Composer in this environment.

## Who is this for?

Hosting Composer in the cloud is a technical process that involves setting up and configuring cloud resources and writing code. A high level of technical proficiency will be necessary to execute this process.

However, this document should provide enough background information to users interested in evaluating this approach without having to implement it yourself.

## Prerequisites

- A subscription to [Microsoft Azure](#).
- Knowledge of Linux and familiarity with package management.
- Familiarity with [nginx](#) and configuring an nginx web server and operating in a command line environment.

## Composer application architecture

Composer is a fairly traditional web application and it consists of several major components:

- The "core" Composer application – a [React](#) and [Node](#) webapp.
  - A "backend" webserver that serves the website and provides features like access to bot projects. This component is a *Node.js* application.
  - A "frontend" client that provides the authoring tools and user interface. This component is a *React JS* application.
- A web application that acts as a bot runtime - dotnet or JavaScript webapps.
  - A "bot runtime" application that takes the content authored in Composer and interprets it to provide a running bot that can be interacted with via the Emulator or other methods.

In most configurations, there will be one copy of the core Composer application running which will serve all users. It is possible to restrict access to this service using [Azure Active Directory](#) (AAD) or a similar system.

In addition, there will be one or more bot runtime applications running for testing purposes – these are designed to be managed through Composer. In the default configuration, these processes run "in the background" on the same computer used to host Composer. It is possible to change this behavior using [plugins](#).

In the next sections we will walk you through the steps to host Composer in an [Azure VM](#).

## Create a Virtual Machine

In its default configuration, Composer uses the local file system to read and write content, and also starts and stops processes "in the background" to enable real-time testing of bots. As a result, our recommended hosting environment for a bot is an [Azure VM](#).

This type of host provides several key capabilities:

- Ability to read and write files to the local filesystem.
- Ability to execute processes.
- Ability to have custom networking configuration and/or proxy settings.

When you create an [Azure VM](#) we recommend using ubuntu. In the VM networking configuration, allow inbound connections to port 3000 (this allows connections to Composer and the bot apps). You will need a port range like 3979-3999 to allow for bots to run locally.

**NOTE**

You can choose the type of VM to host Composer, but this article is specific to hosting Composer in a ubuntu VM.

## Get Composer up and running

1. In your VM instance, install the following prerequisites with correct versions.

**TIP**

- [Node.js](#) (12.18.3 or later)
- [NVM](#) (v0.35.1)
- [npm](#) (6.14.6 or later)
- [Yarn](#) (1.22.4 or later)
- [.NET Core](#) (3.1 or later)

- a. Update the packages available.

```
sudo apt update
```

- b. Install node.

```
sudo apt install nodejs
```

- c. Install npm the node package manager.

```
sudo apt install npm
```

- d. Use npm to install yarn.

```
sudo npm install -g yarn
```

- e. Install the nvm node version manager

```
curl -o- https://raw.githubusercontent.com/creationix/nvm/v0.35.1/install.sh | bash
```

- f. Use nvm to install the long term support version of node (currently 12.18.x)

```
nvm install --lts
```

g. Get the updated ubuntu packages for dotnet

```
 wget https://packages.microsoft.com/config/ubuntu/18.04/packages-microsoft-prod.deb -O packages-microsoft-prod.deb
 sudo dpkg -i packages-microsoft-prod.deb
```

h. Install the dotnet SDK

```
 sudo apt-get update; \ sudo apt-get install -y apt-transport-https && \ sudo apt-get update && \ sudo apt-get install -y dotnet-sdk-3.1
```

2. Create a fork of the [Composer repo](#). With a fork Composer you will be able to make small modifications to the codebase and still pull upstream changes.

3. Follow the instructions below to build Composer and run it.

```
 cd Composer
 yarn
 yarn build
 yarn start
```

4. In your VM instance, load Composer in a browser at `localhost:3000` and verify that you can use Composer in it. Outside your VM, load Composer in a browser at `http://<IP ADDRESS OF VM>:3000` and verify that you can use Composer at this URL.

## Set up nginx

Now you have deployed Composer into your VM and it runs at this URL: `http://<IP ADDRESS OF VM>:3000`. Let's make Composer run on port `80` instead of `:3000` (difference between `mycomposer.com:3000` and `mycomposer.com`) using [nginx](#). Nginx is a web server and proxy service. It can sit in front of the Composer service and pass requests into Composer. It can also be used to enable SSL on the domain without binding with Composer, and to proxy the individual bot processes instead of exposing their ports to the Internet.

**TIP**

[HAProxy](#) is also an option you may consider, but this documentation is specific to nginx.

1. Install nginx.

```
# install nginx web server
Sudo apt install nginx
# edit the main config file
sudo vi /etc/nginx/sites-enabled/default
```

2. Edit the default nginx config to proxy all requests to the composer app running at `:3000`.

a. Find the section that says:

```
location / {
    # First attempt to serve request as file, then
    # as directory, then fall back to displaying a 404.
    try_files $uri $uri/ =404;
}
```

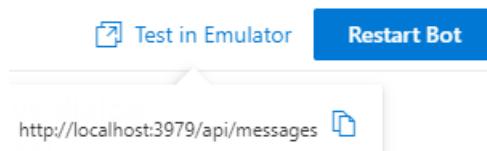
b. Replace the above with the following:

```

location / {
    # First attempt to serve request as file, then
    # as directory, then fall back to displaying a 404.
    proxy_pass http://127.0.0.1:3000;
    proxy_http_version 1.1;
    proxy_set_header Upgrade $http_upgrade;
    proxy_set_header Connection "Upgrade";
    proxy_set_header Host $host;
}

```

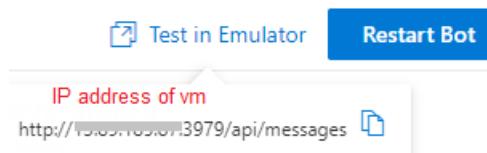
- Now you can load `http://<ip address of VM>/` and you should see Composer. No port number is required. You should be able to create and edit bots in Composer. You should also be able to start the bot – but the URL for the bot will be "localhost" (mouse over **Test in Emulator**). In the next step we will show you how to fix this by patching the code of Composer in two small places.



## Update *Start Bot* Emulator links

The default URL for the Emulator link is `localhost`. Now that Composer is hosted in your VM, we should update this URL in Emulator.

- In your fork, navigate to the `Composer\plugins\localPublish\src\localPublish\src\index.js` file and update `localhost` to your IP or hostname. There are two places you need to update this.
- Run `yarn build:plugins` to rebuild this plugin file.
- Run `yarn startall` to restart your Composer app.
- Now if you create a bot, click **Start Bot** then click **Test in Emulator**. Emulator should open and connect to your dev bot.



- If you want to allow bots to run and be connected on this instance of Composer, you should open network ports. In your Azure portal, go to the VM's networking tab. Add inbound security rule.

## hosted-aug13 | Networking

Inbound port rules

Priority	Name	Port	Protocol
300	SSH	22	TCP
320	HTTP	80	TCP
65000	AllowVnetInbound	Any	Any
65001	AllowAzureLoadBalancer...	Any	Any
65500	DenyAllInBound	Any	Any

## Add inbound security rule

**Basic**

Source \*

Source port ranges \*

Destination \*

Destination port ranges \*

Protocol  Any  TCP  UDP  ICMP

Action  Allow  Deny

Priority \*

Name \*

Description  
this allows the dotnet runtimes to be hosted on the same VM as composer

## Set up Composer to run after you log out

You can set up Composer to run even after you log out. Follow the steps:

1. Install pm2 process manager.

```
sudo npm install -g pm2
```

2. Start Composer using pm2. This will allow the app to continue running even once you log out.

```
pm2 start npm --name composer - start
```

3. Test to see if Composer is already running by using

```
pm2 list
```

You now have a working copy of Composer in a shared location.

### IMPORTANT

Without additional steps, anyone can access this instance of Composer. Before you leave it running, take measures to secure the access control either by installing an auth plugin covered in this [article](#) or by turning on service-level access controls via the azure portal.

## Next steps

- Extend Composer with plugins.

# Managing packages for your bot

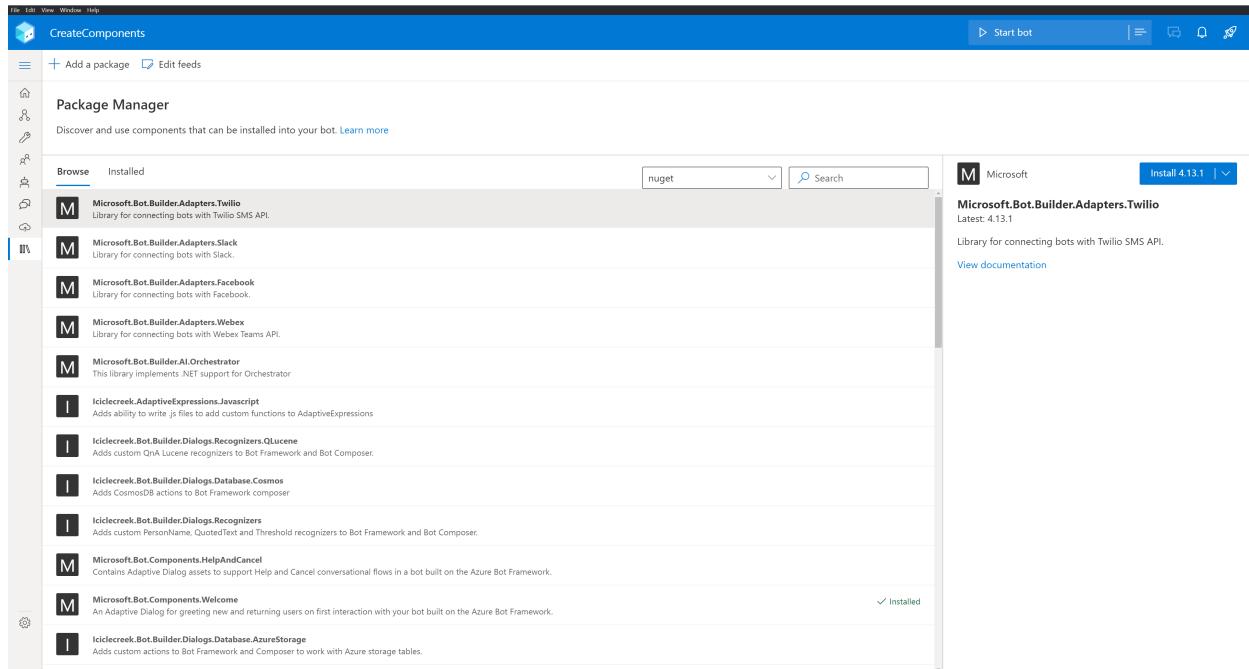
5/20/2021 • 4 minutes to read

APPLIES TO: Composer v2.x

Packages are pieces of bot functionality that can be imported into a bot. They can contain things like dialogs and other declarative assets (`.lu`, `.lg`, `.dialog`, and `.qna` files), and components like custom actions, triggers or adapters.

Typically, they align to a particular vertical or problem space. For example, the Adaptive Cards package (`Microsoft.Bot.Components.AdaptiveCards`) adds custom actions for authoring Adaptive Cards, and custom triggers for responding to Adaptive Card actions.

In Composer, **package manager** is used to add, remove and update packages.



## Prerequisites

- A basic understanding of [packages](#), and how they work in Composer.
- An existing Composer bot to add packages to.

## Add packages to a bot

First, open package manager from the icon on the left navigation rail. From the **Browse** tab, search for packages to add to the bot project. Package manager filters the list of packages to those tagged with `msbot-component` when connected to a public package feed, so not all packages the bot project has a dependency on will be listed. Open the project with a code IDE like Visual Studio to see all package dependencies.

To install a package, select the package to be installed from the list, then click the **Install <version number>** button in the package details pane. Package manager will default to installing the latest stable version of the package. To install a specific version click the down arrow next to the package version number and choosing the required version.

**TIP**

If you receive a "Detected package downgrade:..." error this is likely due to two packages depending on different versions of the `Microsoft.Bot.Builder.Dialogs.Adaptive.Runtime` package. Check to see if there is an update available to that package, or a package that depends on it.

## Update packages

When a new version of a package is available, it can be updated using package manager. From the **Browse** tab select one of the installed packages. If an updated version of the package is available, the new version will be shown on the button on the package details pane.

When updating a package that contains declarative assets that have been altered, installing the new version of the package will replace the customizations with what is contained in the updated package. For example, if the message the Welcome package sends has been customized, and then in the future an updated version is installed, the customization would be lost. Package manager will show a warning when updating packages.

## Remove packages

Packages can be removed from package manager in Composer. When you remove a package, any declarative assets in the package that were copied into the `imported` folder will be deleted, and any instances of custom actions or triggers contained in the package will also be removed. This can potentially leave the bot in a non-functional state - be very cautious when removing packages.

From the **Installed** tab, select a package and then select the **Uninstall** button.

## Connecting to custom feeds

Package manager will connect to the primary public package feed based on your bot project's language (for example, C# bots will be connected to NuGet by default). To connect to other public feeds, private feeds, or even local feeds, click on the **Edit Feeds** button and add the feed to the list.

**TIP**

When working with local NuGet packages, make sure a local feed has been created, and the package is added to it. See the [NuGet documentation](#) on local feeds for more information.

## Package considerations

Packages that contain dialog assets differ from packages you've worked with in the past slightly. Normally files and libraries contained in a package are not intended to be edited directly, however with declarative assets it is very likely that you will want to alter them to meet your needs. To support this, declarative assets in the `exported` folder in packages are *merged* into your bot project, and a copy is created for you to edit. Once you've edited those assets, attempting to upgrade your package will cause a conflict and you'll need to determine manually how to manage merging your edits with the new version of the package.

## Managing packages using CLI tooling

**NOTE**

Managing packages using Package Manager in Composer is the preferred way to work with packages.

Packages can also be managed using command line tooling, which can occasionally be useful for debugging, creating DevOps pipelines, or other advanced scenarios. Composer is performing more actions that just add/update/remove of packages - choosing to manage packages outside of Composer will require manually perform the steps outlined below.

## Add the package

To install packages from the command line, use the normal package installation tool for based on the bot's programming language:

- [JavaScript/Node.js](#)
- [C#/dotnet](#)

Navigate to the bot project folder containing the package.json file and run:

```
cd {BOT_NAME}  
npm install --save [some package]
```

## Merge declarative files

After running one of these commands, the package will be listed in the appropriate place, either the package.json or the .csproj file of the project. Now, use the [Bot Framework CLI tool](#) to extract any included .dialog, .lu and .lg files, as well as to merge any new .schema or .uischema files. Run the following command:

```
bf dialog:merge [package.json or .csproj] --imports /dialogs/imported --output /schemas/sdk
```

The output of the CLI tool will include a list of the files that were added, deleted or updated. Note that **changes to existing files will be overwritten if newer versions are found in a package**.

## Register any components

If the package you're adding contains components (coded extensions registered using the `BotComponents` class), you'll also need to update the `components` array in your `appsettings.json` file. An example is given below:

```
...  
{  
  "name": "Microsoft.Bot.Components.Teams"  
}  
...
```

## Next steps

- [Create packages](#)
- [Extend a bot with code](#)

# Creating packages

6/11/2021 • 3 minutes to read

**APPLIES TO:** Composer v2.x

Packages are bits of bots that you want to reuse and/or share. They can contain declarative assets, schema files, and/or components.

At a high level, the steps for creating a package are:

1. Create the declarative files (use Composer to create them).
2. Create the components (use your favorite IDE to create them).
3. Package your files (use NuGet for C# runtime bots, and npm for bots using the JavaScript runtime).
4. Publish your package to a package feed (public, private, or local).

## Prerequisites

- A basic understanding of creating packages with [NuGet](#) or [npm](#)
- A basic understanding of [packages in Composer](#)
- [A bot built using Composer](#).
- A set of files to package, like the ones created in the [custom action article](#). Alternatively, use the [sample code](#).

## Complete sample

The complete code for this sample can be found in the Bot Framework samples repository on GitHub:

- [C#/dotnet sample](#)

## Declarative files in packages

### TIP

Use Composer to create the declarative files, then add them to a folder named *exported* in the package project.

You can include declarative files (`.dialog`, `.lu`, `.lg`, or `.qna`) in your packages. Your declarative files **must** be in a folder named *exported* at the root of your package. Typically, declarative files are not included by default (for NuGet, you'd add them in either the `.csproj` or `.nuspec` file) when you create your package.

### Example `.csproj` file

This example `.csproj` file demonstrates including declarative files in the *exported* folder for a package.

```

<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <OutputType>Library</OutputType>
    <TargetFramework>netcoreapp3.1</TargetFramework>
    <PackageId>Microsoft.Bot.Components.Samples.MemberUpdates</PackageId>
    <Description>This library implements .NET support for custom triggers for conversation member updates.</Description>
    <Summary>This library implements .NET support for custom triggers for conversation member updates. OnMembersAdded and OnMembersRemoved.</Summary>
    <ContentTargetFolders>content</ContentTargetFolders>
    <PackageTags>msbot-component;msbot-trigger</PackageTags>
  </PropertyGroup>
  <ItemGroup>
    <Content Include="**/*.schema" />
    <Content Include="**/*.uischema" />
    <None Include="exported/**/*.*" Pack="true" PackagePath="exported" />
    <None Include="README.md" Condition="Exists('README.md')" Pack="true" PackagePath="" />
    <PackageReference Include="Microsoft.Bot.Builder.Dialogs.Adaptive.Runtime" Version="4.13.1" />
  </ItemGroup>
</Project>

```

## Components in packages

The components in a package are the same as what are created when [extending your bot with code](#). Ensure the `BotComponent` class is used to register any components.

### Example `BotComponent` class

To dynamically register your action with the adaptive runtime, define a `BotComponent` by inheriting from the `Microsoft.Bot.Builder.BotComponent` class. For example, this registers a simple custom action called `MyCustomAction`:

```

using Microsoft.Bot.Builder;
using Microsoft.Bot.Builder.Dialogs.Declarative;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.DependencyInjection;

namespace MyBot
{
    public class MyBotComponent : BotComponent
    {
        public override void ConfigureServices(IServiceCollection services, IConfiguration configuration)
        {
            // Component type
            services.AddSingleton<DeclarativeType>(sp => new DeclarativeType<MyCustomAction>
            (MyCustomAction.Kind));
        }
    }
}

```

## Package tags

In order for the adaptive runtime and package manager to correctly install a package, it must be tagged appropriately. All packages published to a public feed must be tagged with `msbot-component` to be displayed correctly in package manager.

Additionally, tag the package with one or more of the tags below depending on its contents.

- `msbot-content`
- `msbot-middleware`

- msbot-action
- msbot-trigger
- msbot-adapter

## Walkthrough: Create and publish a NuGet package

This walkthrough uses the [sample code](#) and the [dotnet CLI](#) to create a NuGet package.

### Create your package

1. Open a command prompt and clone the Bot Framework Samples repository, and change into the correct folder.

```
git clone https://github.com/microsoft/BotBuilder-Samples.git  
cd BotBuilder-Samples/composer-samples/csharp_dotnetcore/packages/DialogAndTriggerPackage
```

2. Create your package.

```
dotnet pack
```

3. Verify the package was created in the `/bin/debug` folder.

### Publishing your package

You can publish your package to a local feed, or to a hosted feed (private or public). This walk through creates a local feed and adds the package to it.

From a command prompt in the `/bin/debug` folder:

```
nuget add ".\Microsoft.Bot.Components.Samples.DialogAndTriggerPackage.1.0.0.nupkg" -Source  
"c:\someplace\feed"
```

This creates a new folder, instantiates a local NuGet feed in that folder, and adds the package to it.

#### NOTE

Make sure the package version is updated with each package and publish iteration. NuGet will behave unexpectedly if the package is republished with the same version.

### Test the package in Composer

1. Open a bot in Composer to test the package against. Alternatively, create a new *Empty Bot* for testing.
2. Open the *package manager* page, then click the **Edit feeds** button.
3. Click the **Add a new feed** button.
4. Give the feed a name, select *NuGet* in the **Type** dropdown, and put the location of your feed in the **URL** field - `c:\someplace\feed` (the `-Source` argument used previously).
5. Click **Done**.
6. Back in *package manager*, select the newly added feed in the drop down. The package will be listed.
7. Select the package, and click the **Install** button.

## Learn more

- [NuGet documentation](#)
- [npm documentation](#)
- [Packages in Composer](#)

# Create custom actions

6/11/2021 • 5 minutes to read

APPLIES TO: Composer v2.x

In Bot Framework Composer, [actions](#) are the main contents of a [trigger](#). Composer provides different types of actions, such as [Send a response](#), [Ask a question](#), and [Create a condition](#). Besides these built-in actions, additional actions can be added through [packages](#) or by [creating components](#) that includes custom actions.

This article explains how to create a custom action that multiplies two inputs together.

## Prerequisites

- A basic understanding of [actions](#) in Composer.
- A basic understanding of [extending a bot with components](#)
- [A bot built using Composer](#).
- The latest version of the [Bot Framework CLI](#).

## Complete sample

The complete code for this sample can be found in the Bot Framework samples repository on GitHub:

- [C#/dotnet sample](#)

## Setup the Bot Framework CLI tool

The Bot Framework CLI tools include the `bf-dialog` command for working with `.schema` files. If the Bot Framework CLI tool is not already installed, open an elevated command prompt and run the following command to install the Bot Framework tools:

```
npm i -g @microsoft/botframework-cli
```

## Setup the component project

To create a custom action (or any component), first setup a new project, and add the necessary package dependencies for working with adaptive dialogs and the Bot Framework SDK.

1. Locate the `<myBot>.sln` file for the bot, and open it in an editor (like Visual Studio or Visual Studio Code).
2. Add a new project named *MultiplyDialog* to your solution. In Visual Studio right-click on the solution in the *Solution Explorer* and select **Add > New Project**. Use the *Class Library* project template.
3. Add a reference to the *Microsoft.Bot.Builder.Adaptive.Runtime* package. Use the same version as the bot depends on.

```
<PackageReference Include="Microsoft.Bot.Builder.Dialogs.Adaptive.Runtime" Version="4.13.1" />
```

4. Add a project reference from the bot project to the component project. Right-click on the `<myBot>` project and select **Add > Project Reference**. Choose the *MultiplyDialog* project and click **OK**.
5. Build the entire solution to restore all packages and validate the dependency tree.

## Create the custom action

Actions in Composer are special implementations of the `Dialog` base class. This allows each action in the trigger to be pushed onto the dialog stack, and executed in turn.

In the new project, rename the `Class1.cs` file to `MultiplyDialog.cs`, and update it's contents to look like the below:

```
using System;
using System.Runtime.CompilerServices;
using System.Threading;
using System.Threading.Tasks;
using AdaptiveExpressions.Properties;
using Microsoft.Bot.Builder.Dialogs;
using Newtonsoft.Json;

public class MultiplyDialog : Dialog
{
    [JsonConstructor]
    public MultiplyDialog([CallerFilePath] string sourceFilePath = "", [CallerLineNumber] int sourceLineNumber = 0)
        : base()
    {
        // enable instances of this command as debug break point
        RegisterSourceLocation(sourceFilePath, sourceLineNumber);
    }

    [JsonProperty("$kind")]
    public const string Kind = "MultiplyDialog";

    [JsonProperty("arg1")]
    public NumberExpression Arg1 { get; set; }

    [JsonProperty("arg2")]
    public NumberExpression Arg2 { get; set; }

    [JsonProperty("resultProperty")]
    public StringExpression ResultProperty { get; set; }

    public override Task<DialogTurnResult> BeginDialogAsync(DialogContext dc, object options = null,
CancellationToken cancellationToken = default(CancellationToken))
    {
        var arg1 = Arg1.GetValue(dc.State);
        var arg2 = Arg2.GetValue(dc.State);

        var result = Convert.ToInt32(arg1) * Convert.ToInt32(arg2);
        if (this.ResultProperty != null)
        {
            dc.State.SetValue(this.ResultProperty.GetValue(dc.State), result);
        }

        return dc.EndDialogAsync(result: result, cancellationToken: cancellationToken);
    }
}
```

## Create the schema file

The `.schema` file for the component is a partial schema that will be merged into the main `.schema` file for the bot. Although it is possible to edit the main `sdk.schema` file for the bot directly, doing so is not recommended. Merging partial schema files will isolate changes, allow for easier recovery from errors, and enable easier packaging of your component for reuse.

Create a new file in the project named `MultiplyDialog.schema` and update the contents to the below:

## IMPORTANT

The name of the `.schema` file must match the `Kind` variable defined in the `MultiplyDialog.cs` file exactly, including casing.

```
{  
    "$schema": "https://schemas.botframework.com/schemas/component/v1.0/component.schema",  
    "$role": "implements(Microsoft.IDialog)",  
    "title": "Multiply",  
    "description": "This will return the result of arg1*arg2",  
    "type": "object",  
    "additionalProperties": false,  
    "properties": {  
        "arg1": {  
            "$ref": "schema:/definitions/integerExpression",  
            "title": "Arg1",  
            "description": "Value from callers memory to use as arg 1"  
        },  
        "arg2": {  
            "$ref": "schema:/definitions/integerExpression",  
            "title": "Arg2",  
            "description": "Value from callers memory to use as arg 2"  
        },  
        "resultProperty": {  
            "$ref": "schema:/definitions/stringExpression",  
            "title": "Result",  
            "description": "Value from callers memory to store the result"  
        }  
    }  
}
```

## Create the `BotComponent` class

The adaptive runtime will dynamically discover and inject components at startup time.

1. Create a new `MultiplyDialogBotComponent.cs` file in the project and update the contents to

```
using Microsoft.Bot.Builder;  
using Microsoft.Bot.Builder.Dialogs.Declarative;  
using Microsoft.Extensions.Configuration;  
using Microsoft.Extensions.DependencyInjection;  
  
public class MultiplyDialogBotComponent : BotComponent  
{  
    public override void ConfigureServices(IServiceCollection services, IConfiguration configuration)  
    {  
        // Anything that could be done in Startup.ConfigureServices can be done here.  
        // In this case, the MultiplyDialog needs to be added as a new DeclarativeType.  
        services.AddSingleton<DeclarativeType>(sp => new DeclarativeType<MultiplyDialog>  
(MultiplyDialog.Kind));  
    }  
}
```

2. In the `appsettings.json` file of the bot project (located at `<mybot>\settings`) to include the `MultiplyDialogBotComponent` in the `runtimeSettings/components` array.

```
"runtimeSettings": {  
  "components": [  
    {  
      "name": "CustomAction.MultiplyDialog"  
    }  
  ]  
}
```

3. Build the entire solution to validate everything was added correctly.

## Merge schema files

### NOTE

This step only needs to be performed when a new `.schema` file is added or updated.

The final step is to merge the partial schema file from the *MultiplyDialog* project into the main  *sdk.schema* file for the bot. This makes the custom action available for use in Composer.

1. Navigate to the *schemas* folder in the *myBot* project. This folder contains a PowerShell script and a bash script. Use an elevated PowerShell terminal to execute the PowerShell script. You will need to either copy/paste the contents of the script, or ensure your *execution-policy* allows for running unsigned scripts.
2. To validate the script executed successfully, search for *MultiplyDialog* inside the `MyBot\schemas\ sdk.schema` file and validate that the partial schema from the *MultiplyDialog.schema* file is included in  `sdk.schema`.

### NOTE

Alternatively, you can click-to-run the `update-schema.sh` file inside the `MyEmptyBot\schemas` folder to run the bash script.

## Test

Open the bot project in Composer to test the added custom action. If the project is already loaded, return to *Home* in Composer, and reload the project.

1. Open the bot in Composer. Select a trigger add the custom action to.
2. Select **+** under the trigger node to see the actions menu. Then choose **Custom Actions > Multiply**.
3. On the *Properties* panel on the right side, enter two numbers in the argument fields: *Arg1* and *Arg2*. Enter "dialog.result" in the *Result* property field.
4. Add a *Send a response* action. Enter "The result is: \${dialog.result}" in the Language Generation editor.
5. Select **Start Bot** to test the bot in Web Chat. When triggered, the bot will respond with the test result entered in the previous step.

## Additional information

- [Create custom triggers](#)
- [Bot Framework SDK Schemas](#)

# Create custom triggers

6/11/2021 • 5 minutes to read

**APPLIES TO:** Composer v2.x

In Bot Framework Composer, triggers are fired when events matching a condition occur.

This article demonstrates how to create a custom trigger named *OnMembersAdded* that will fire when members are added to the conversation.

## Prerequisites

- A basic understanding of [triggers](#) in Composer.
- A basic understanding of [extending a bot with components](#)
- [A basic bot built using Composer](#).
- The latest version of the [Bot Framework CLI](#).

## Complete sample

The complete code for this sample can be found in the Bot Framework samples repository on GitHub:

- [C#/dotnet sample](#)

## Setup the Bot Framework CLI tool

The Bot Framework CLI tools include the *bf-dialog* command for working with `.schema` files. If the Bot Framework CLI tool is not already installed, open an elevated command prompt and run the following command to install the Bot Framework tools:

```
npm i -g @microsoft/botframework-cli
```

## Setup the component project

To create a custom trigger (or any component), first setup a new project, and add the necessary package dependencies for working with adaptive dialogs and the Bot Framework SDK.

1. Locate the `<myBot>.sln` file for your bot, and open it in an editor (like Visual Studio or Visual Studio Code).
2. Add a new project named *MemberUpdates* to your solution. In Visual Studio right-click on the solution in the *Solution Explorer* and select **Add > New Project**. Use the *Class Library* project template.
3. Add a reference to the *Microsoft.Bot.Builder.Adaptive.Runtime* package. Use the same version as the bot depends on.

```
<PackageReference Include="Microsoft.Bot.Builder.Dialogs.Adaptive.Runtime" Version="4.13.1" />
```

4. Add a project reference from the bot project to the component project. Right-click on the /project and select **Add > Project Reference**. Choose the *MemberUpdates* project and click **OK**.
5. Build the entire solution to restore all packages and validate the dependency tree.

## Create the custom trigger

Custom triggers extend the base `OnActivity` class, adding an expression in order to respond to specific events.

This sample creates a trigger that fires when new members are added to a conversation - specifically it responds to `ConversationUpdate` Activities if the `membersAdded` array is not null or empty.

In the *MemberUpdates* project, rename the *Class1.cs* file to *OnMembersAdded* and update the contents to:

```
using System.Collections.Generic;
using System.Runtime.CompilerServices;
using AdaptiveExpressions;
using Microsoft.Bot.Builder.Dialogs;
using Microsoft.Bot.Builder.Dialogs.Adaptive.Conditions;
using Microsoft.Bot.Schema;
using Newtonsoft.Json;

public class OnMembersAdded : OnActivity
{
    [JsonProperty("$kind")]
    public new const string Kind = "OnMembersAdded";

    [JsonConstructor]
    public OnMembersAdded(List<Dialog> actions = null, string condition = null, [CallerFilePath] string callerPath = "", [CallerLineNumber] int callerLine = 0)
        : base(type: ActivityTypes.ConversationUpdate, actions: actions, condition: condition, callerPath: callerPath, callerLine: callerLine)
    {
    }

    protected override Expression CreateExpression()
    {
        // The Activity.MembersAdded list must have more than 0 items.
        return Expression.AndExpression(Expression.Parse($"count({TurnPath.Activity}.MembersAdded) > 0"),
            base.CreateExpression());
    }
}
```

## Create the schema files

The `.schema` file for the component is a partial schema that will be merged into the main `.schema` file for the bot. Although it is possible to edit the main `sdk.schema` file for the bot directly, doing so is not recommended. Merging partial schema files will isolate changes, allow for easier recovery from errors, and enable easier packaging of your component for reuse.

1. Create a new file in the project named *OnMembersAdded.schema* and update the contents to the below:

### IMPORTANT

The name of the `.schema` file must match the `Kind` variable defined in the `OnMembersAdded.cs` file exactly, including casing.

```
{
    "$schema": "https://schemas.botframework.com/schemas/component/v1.0/component.schema",
    "$role": [ "implements(Microsoft.ITrigger)", "extends(Microsoft.OnCondition)" ],
    "title": "On Members Added",
    "description": "Actions to perform on receipt of an activity with type 'ConversationUpdate' and MembersAdded > 0.",
    "type": "object",
    "required": [
    ]
}
```

2. Create another new file in the project named *OnMembersAdded.uischema*. This file tells Composer where to display the component.

```
{
    "$schema": "https://schemas.botframework.com/schemas/ui/v1.0/ui.schema",
    "form": {
        "order": [
            "condition",
            "*"
        ],
        "hidden": [
            "actions"
        ],
        "label": "Members Added",
        "subtitle": "Members Added ConversationUpdate activity",
        "description": "Handle the events fired when a members have been added to a conversation.",
        "helpLink": "https://docs.microsoft.com/composer/how-to-define-triggers#activities"
    },
    "trigger": {
        "label": "Members Added (ConversationUpdate activity)",
        "submenu": "Member Updates"
    }
}
```

## Create the `BotComponent` class

The adaptive runtime will dynamically discover and inject components at startup time.

1. Create a new *MultiplyDialogBotComponent.cs* file in the project and update the contents to:

```
using Microsoft.Bot.Builder;
using Microsoft.Bot.Builder.Dialogs.Declarative;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.DependencyInjection;

public class MultiplyDialogBotComponent : BotComponent
{
    public override void ConfigureServices(IServiceCollection services, IConfiguration configuration)
    {
        // Anything that could be done in Startup.ConfigureServices can be done here.
        // In this case, the MultiplyDialog needs to be added as a new DeclarativeType.
        services.AddSingleton<DeclarativeType>(sp => new DeclarativeType<OnMembersAdded>
(OnMembersAdded.Kind));
    }
}
```

2. In the *appsettings.json* file of the bot project (located at `<mybot>\settings`) to include the `MultiplyDialogBotComponent` in the `runtimeSettings/components` array.

```
"runtimeSettings": {  
    "components": [  
        {  
            "name": "MemberUpdates"  
        }  
    ]  
}
```

3. Build the entire solution to validate everything was added correctly.

## Merge schema files

The final step is to merge the partial schema file from the *MemberUpdates* project into the main  *sdk.schema* file for the bot. This makes the custom action available for use in Composer.

### NOTE

This step only needs to be performed when a new `.schema` file is added or updated.

1. Navigate to the *schemas* folder in the *myBot* project. This folder contains a PowerShell script and a bash script. Use an elevated PowerShell terminal to execute the PowerShell script. You will need to either copy/paste the contents of the script, or ensure your *execution-policy* allows for running unsigned scripts.
2. To validate the script executed successfully, search for *MultiplyDialog* inside the `MyBot\schemas\ sdk.schema` file and validate that the partial schema from the *MultiplyDialog.schema* file is included in  `sdk.schema`.

### NOTE

Alternatively, you can click-to-run the `update-schema.sh` file inside the `MyEmptyBot\schemas` folder to run the bash script.

## Test

Open the bot project in Composer and you should be able to test your added custom trigger. If the project is already loaded, return to *Home* in Composer, and reload the project.

1. Open the bot in Composer. Select a dialog to add this custom trigger with and select ....
2. Select **+ Add new trigger** to open the triggers menu. Choose the **Member Updates** trigger.
3. Select **Members Added (ConversationUpdate activity)**, then click **Submit**.
4. Add a *Send a response* action to the newly created trigger, and enter some text. For example "Members added".
5. Select **Start Bot** to test the bot in Web Chat. The bot will receive the appropriate Activity when the conversation initiates, and respond with the text entered in the previous step.

## Additional information

- [Bot Framework SDK Schemas](#)
- [Create custom actions](#)

# Create and test a local skill

5/20/2021 • 8 minutes to read

**APPLIES TO:** Composer v1.x and v2.x

This article explains how to create a local skill in an existing bot project, connect the existing bot to the local skill, and test the behavior locally.

## Prerequisites

- Knowledge of bot development in Composer. For information, see the [Introduction to Bot Framework Composer](#).
- Knowledge of skill and skill consumer bots. For information, see [About skills](#).

## Create your root bot

Create an initial bot. It will consume the skill you create in a later step.

- [Composer v2.x](#)
- [Composer v1.x](#)

1. In the Composer **Home** view, create a new empty bot:

- Select **Create new**, select the **C#** tab, select the **Empty Bot** template, and select **Next**.
- Give your bot a name—the bot used in this article is named **RootBot**.
- Select **Azure Web App** for the bot **Runtime type**.
- Select a location for the bot project.
- Then select **Create** to have Composer create the bot.
- This process can take a few moments.

2. The **Get started** panel contains links to more information about creating bots. You can close it for now.

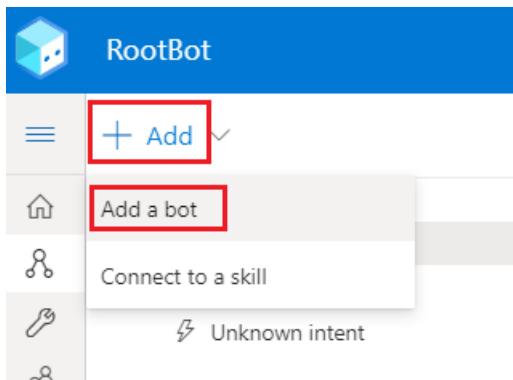
3. Select the **RootBot** dialog and change the recognizer type it uses to the **Regular expression recognizer**.

In a later step, you will add an **Intent recognized** trigger to allow the user to start the skill.

## Create the skill

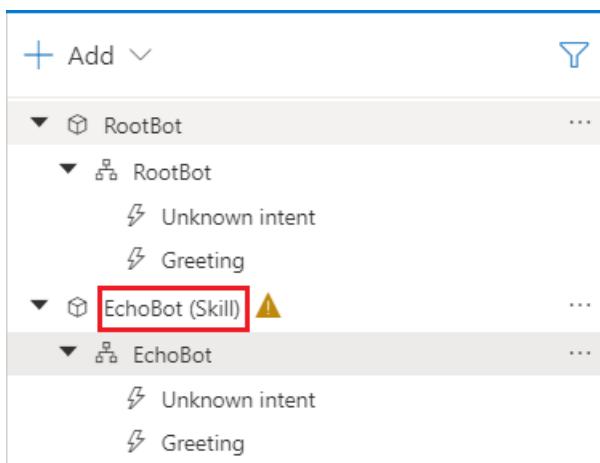
- [Composer v2.x](#)
- [Composer v1.x](#)

1. At the top of the navigation pane, select **Add (+)**, then **Add a bot**.



2. In the **Add a bot** wizard:

- Select **Create a new bot** and select **Next**.
- On the **Create a skill in your bot** page, select the **C#** tab, select the **Empty Bot template**, and select **Next**.
- On the **Create a bot project** page, give your bot a name—the skill bot used in this article is named `EchoSkill`.
- Select **Azure Web App** for the bot **Runtime type**.
- Select the location for the bot project, and select **Create** to have Composer create the bot. This process can take a few moments.
- On the **Enable Orchestrator** page, select **Skip**. Orchestrator is not required for the bot described in this article.



3. Select the **EchoSkill** bot's **EchoSkill** dialog.

- By default, the dialog has **Auto end dialog** set to `true`. Change this to `false` to let the skill loop. Otherwise, it will exit when the dialog's **Unknown intent** trigger comes to an end.
- Change the recognizer type it uses to the **Regular expression recognizer**.

## Add logic to the skill

- [Composer v2.x](#)
- [Composer v1.x](#)

### Update the skill's Greeting trigger

1. Select the skill's **Greeting** trigger.

2. Delete the **Loop**: for each item action.
3. Add a **Send a response** action in its place.

- Change the response to:

I will repeat what you enter, until you enter \*\*stop\*\*.

4. When the skill first starts, this will help the user know how to exit the skill.

### Update the skill's Unknown intent trigger

Update the unknown intent trigger to echo the user's utterance.

1. Select the **Unknown intent** trigger of the EchoSkill dialog.
2. On the authoring canvas, select the **Send a response** action.
3. In the properties pane, change the **Responses** under **Bot Responses** to:

You said, \${turn.activity.text}

### Add a trigger to let the user exit the skill

Since the echo skill is designed to loop, you need to give the user a way to exit from the skill and return to the root bot.

1. Go to the EchoSkill dialog and add an **Intent recognized** trigger.

Name it **stop** and set the regEx pattern to **stop**.

2. To the new **stop** trigger, add a **Send a response** action to acknowledge the user's input. Set the bot response to:

Thanks for using the echo skill

3. Then add a second action. Select **End this dialog** action from the **Dialog management** sub menu.

This trigger will end the dialog and return control to the root bot.

Now, your local skill is ready! Let's connect this skill to another bot in Composer.

## Consume the skill in the root bot

Update the root bot to start the skill when the user enters "echo".

- [Composer v2.x](#)
- [Composer v1.x](#)

1. Go to the **RootBot** dialog and add an **Intent recognized** trigger.

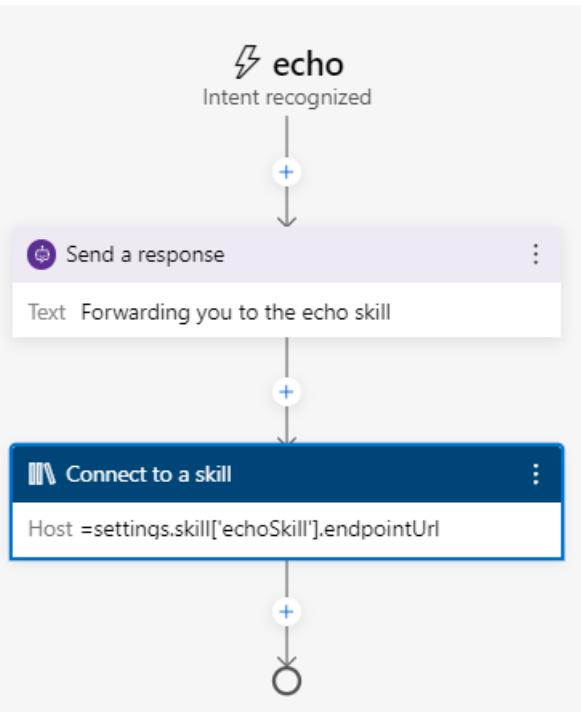
Name it **echo** and set the regEx pattern to **echo**.

2. To the new **echo** trigger, add a **Send a response** action to acknowledge the user's input. Set the bot response to:

Forwarding you to the echo skill

3. Add an **Access external resources** > **Connect to a skill** action.
4. In the **Properties** pane for the **Connect to a skill** action, set the following properties:
  - a. In the **Skill Dialog Name** field, select the skill you want to connect to, the **EchoSkill**.
  - b. Note that by default the **Skill Endpoint** field is set to **Local Composer**. Composer will connect to the skill running locally on your machine. This also means Composer doesn't need a manifest to call the skill.
  - c. Leave **Activity processed** set to `true`. When the root bot calls the skill, it will send the activity you specify.
  - d. Switch the **Activity** to **Show code** and use this template:

```
[Activity
  type = conversationUpdate
]
```



5. Finally, update the **Unknown intent** trigger in the **RootBot** dialog.

- Select the **Send a response** action and change the response to:

```
Enter **echo** to access the echo skill
```

- This will help the user tell the root bot responses apart from the echo skill's.

You now can test the skill bot and its connection with the consumer bot.

#### NOTE

You don't need an Azure registration resource for the local bot-to-bot connection.

## Test the bots locally

Start both bots locally. This will allow you to test that you can access the echo skill from the root bot and that the

echo skill can end and return control to the root bot.

- [Composer v2.x](#)
- [Composer v1.x](#)

1. Select **Start all** in the upper-right corner of the screen. This tells Composer to start all local bots in the project.
2. Once the bots have started, the **Local bot runtime manager** opens. Use **Start and stop local bot runtimes** (≡) to show or hide the manager.
3. In the **Local bot runtime manager**, test the RootBot in Web Chat.
  - a. Interact with the root bot, then enter `echo` to have the root bot start the skill.

Welcome to the EchoBot sample

hi

Just now

Enter **echo** to access the echo skill

echo

Just now

Forwarding you to the echo skill

You said 'echo'

- b. Interact with the skill, then enter `stop` to have the echo skill end and return control to the root bot.

You said 'echo'

hi

Just now

You said 'hi'

stop

Just now

Thanks for using the echo skill

## Further reading

- To learn to publish a skill, see how to [Publish a skill](#).
- To learn to connect your bot to a remote skill, see how to [Connect to a remote skill](#).

# Connect to a remote skill

5/20/2021 • 11 minutes to read

**APPLIES TO:** Composer v1.x and v2.x

The Bot Framework supports *skills*, which are bots that encapsulate bits of conversational logic that can then be used to extend other bots. The bot that contains the skill is a *skill bot* and the bot that accesses the skill is a *skill consumer*, also referred to as a *root bot*, if it's the initial bot the user interacts with. You can build a user-facing bot and extend it by consuming your own or third-party *skills*, without the need to access the skill's source code.

To develop and test a root bot that accesses a deployed skill, the root bot needs an endpoint that the skill can send replies to. The root bot will forward the skill's replies to the user.

This article describes how to connect a root bot running locally to an already deployed skill. The local root bot uses an Azure Bot Channels Registration and ngrok to provide an endpoint in the cloud that forwards to a local endpoint on your machine.

If you don't have a skill yet or you're looking for how to create a remote skill with Composer, read the [Export a skill](#) article.

## IMPORTANT

Connecting to a skill in Composer is a technical process that involves many steps such as setting up Composer and configuring Azure resources. A high level of technical proficiency will be necessary to execute this process.

## Prerequisites

- A subscription to [Microsoft Azure](#).
- [A basic bot built with Composer](#).
- A good understanding of [skills](#) in the Bot Framework SDK.
- Install [ngrok](#).
- A remote skill and The URL for the skill manifest file.
  - The remote skill bot in this article is named "EchoBot".

If you do not yet have a remote skill, first follow the steps in how to [Create and test a local skill](#) and [Publish a skill to Azure](#).

The bots used in this article are a root bot named `RootBot` and a skill bot that's been published named `EchoSkill`.

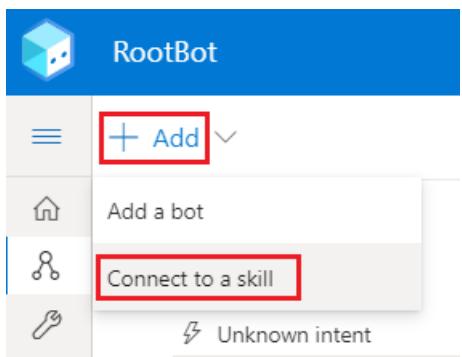
## Connect to the remote skill

- [Composer v2.x](#)
- [Composer v1.x](#)

### Add the remote skill

To add the remote version of the skill that you published earlier:

1. Go to the [Create](#) view. At the top of the navigation pane, select **Add (+)**, then **Connect to a skill**.



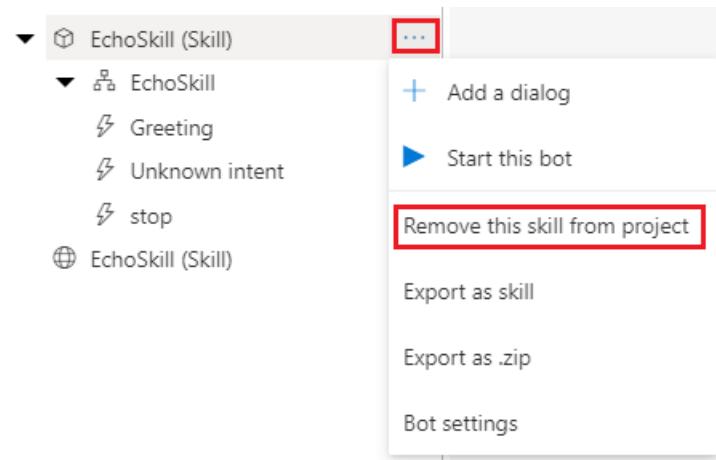
2. On the **Add a skill** page, enter the skill manifest URL for your remote skill and select **Next**.
  - If you get a **Manifest URL can not be accessed** error, check whether your skill is published and its manifest URL is entered correctly.
3. If Composer can access the URL, it displays information from the manifest. Select **Done** to add the remote skill to your root bot.
4. Composer adds an entry for the remote skill to the navigation pane.

EchoSkill (Skill)

### Remove the local skill

To avoid confusion, if your root bot is currently using a local version of your remote skill, remove the local skill from the root bot's project. Composer will not delete any files, and you can continue to maintain and publish the skill from its own project.

1. Go to the **Create** view and select **More options** next to the */local skill*. Then select **Remove this skill from project**.



2. When Composer shows a warning about removing the local skill from the project, select **Yes** to continue. You will update your root bot to access the remote version of the skill in the next step.

### Update the root bot to reference the remote skill

To update the root bot to access the remote skill:

1. Go to the **Create** view.
2. Select the trigger in your root bot that contains the **Connect to a skill** action and select the **Connect to a skill** action.
  - In this article, this is the **echo** trigger in the **RootBot** dialog.
3. In the properties pane for the action:
  - a. For **Skill Dialog Name**, select the remote skill.
  - b. For **Skill Endpoint**, select the endpoint you want to access. If your skill has just one endpoint, the

endpoint is selected by default.

Now your root bot will access the remote skill, using the information in the skill's manifest.

### Provision resources for your root bot

To access a remote skill, your root bot needs a valid app ID and password. If you have published your root bot before, you can skip this step.

1. Go to the **Publish** view and select the **Publishing profile** tab.
2. To add a publishing profile for your root bot, select **Add new**.
3. On the **Create a publishing profile** page, enter a name for the profile and select **Publish bot to Azure** for the **Publishing target**. Then, select **Next**.
4. Follow the steps to create the resources. For more information, see how to [Publish a bot to Azure](#).
5. When Composer displays a provision success message, your resources are ready.

### Configure the root bot

To retrieve the root bot app ID and password:

1. Go to the **Configure** view.
2. Select the root bot.
3. Select the **Development resources** tab.
4. Scroll to the bottom of the pane and select **Retrieve App ID**.
  - a. On the **Retrieve App ID from publishing profile** page, select the profile to use and select **Save App ID**.

Composer will apply these credentials when starting your root locally.

## Test in the Emulator

You can now test your local root bot's ability to connect to the remote skill.

- [Composer v2.x](#)
- [Composer v1.x](#)

With a local root bot and a remote skill, the skill is running in the cloud and your root bot is running on your machine. For the two to communicate, you need to configure a URL for your root bot that the skill can access from the cloud.

### Start the local root bot

The first time you start a bot that connects to a remote skill, Composer displays the ngrok command to use to create a forwarding endpoint.

1. Select **Start bot** to start the root bot.
  - If Composer displays the ngrok command to use. Copy the command for a later step. Then, close the pop-up window.
  - Otherwise, Composer displays the **Local bot runtime manager**.
    - a. Hover over **Test in Emulator**.
    - b. Copy the port number for the root bot. For example, if the messaging endpoint is `http://localhost:3980/api/messages`, the port number is 3980. You need this to create the forwarding endpoint.

### Create a forwarding URL

1. Open a command prompt or terminal.

- If ngrok is not in your path, change to the directory that contains `ngrok.exe`.
- Run the command you copied from Composer earlier. It should look similar to this, where `<port-number>` is the port number for your root bot.

```
ngrok.exe http <port-number> --host-header=localhost
```

- ngrok creates a forwarding URL in the cloud that directs to the port on which your local bot is hosted.
- Copy the https URL from ngrok and leave the command running.
  - The forwarding URL expires when the ngrok session expires or when you end the ngrok process.

## Update the skill host endpoint

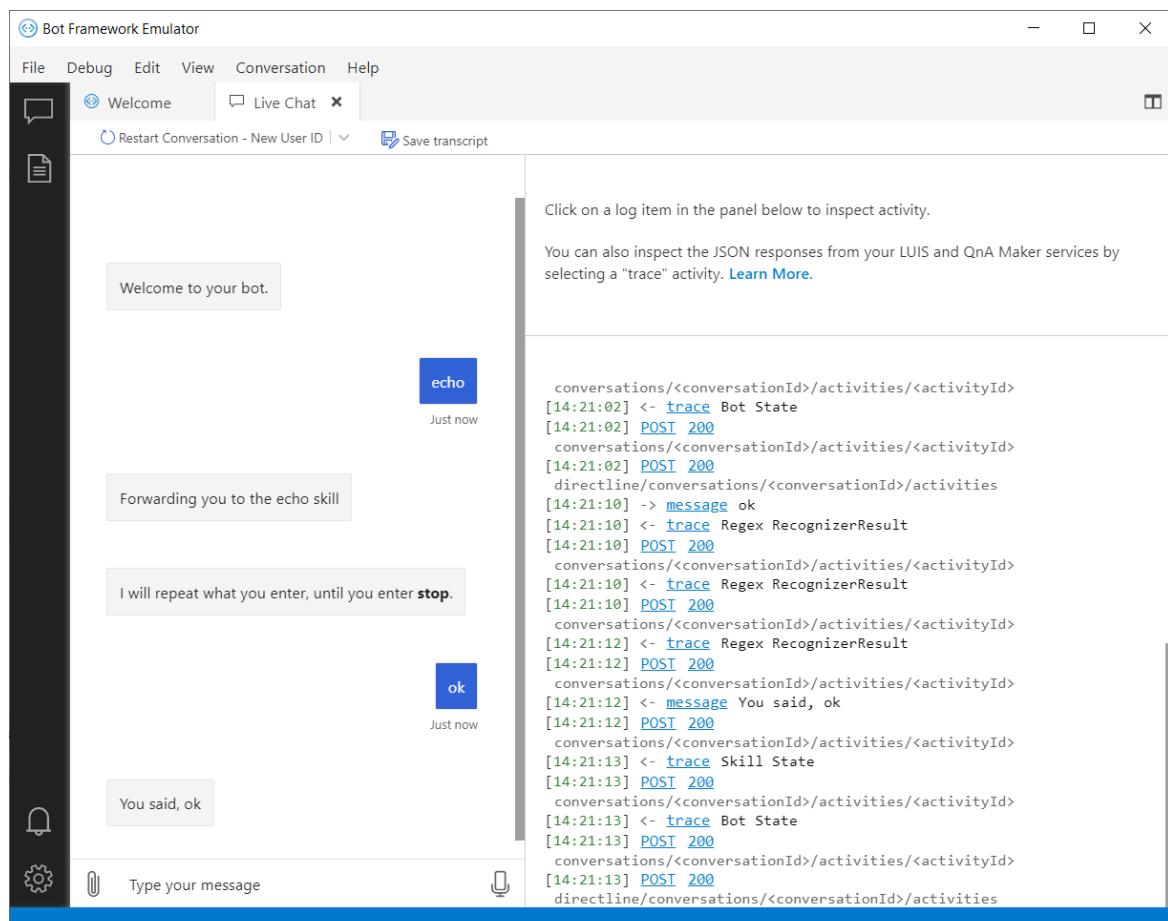
- In Composer, go to the **Configure** view, select your root bot, and select the **Skill configuration** tab.
- Under **Call skills**, change the **Skill host endpoint URL** to use the forwarding URL as its base, instead of the localhost port.

For example, if your local port is `3980` and your forwarding URL is `https://37bab15eb6e2.ngrok.io`, then you would change `http://localhost:3980/api/skills` to `https://37bab15eb6e2.ngrok.io/api/skills`.

Your root bot uses this URL to receive responses from the skill bot.

## Start the root bot and open it in Emulator

- Restart your root bot from Composer.
- From the **Local bot runtime manager**, select **Test in Emulator**.
- Composer opens your root bot in the Bot Framework Emulator.
- Enter text in the Emulator and see the response.



## Further reading

- [About skills](#)
- [Export a skill.](#)

# Single sign on for skills

5/20/2021 • 2 minutes to read

APPLIES TO: Composer v2.x

This article shows how to use the single-sign-on (SSO) feature for skills. To do so, it uses a *consumer bot*, also known as *root bot*, to interact with a *skill bot*.

SSO enables users to sign in to the root bot, and not require signing into each skill bot they use through the root bot. An OAuth input prompt within a skill is allowed to access shared resources on behalf of the root bot's Azure Active Directory (AD) OAuth connection through a token exchange process. In this example, the *token exchange* is processed through the [Bot.Builder.Community.Components.TokenExchangeSkillHandler](#) package installed within the *root bot*.

## IMPORTANT

Skill single sign on in Composer is a technical process that involves many steps such as setting up the Azure AD applications and configuring Azure resources. A high level of technical proficiency will be necessary to execute this process.

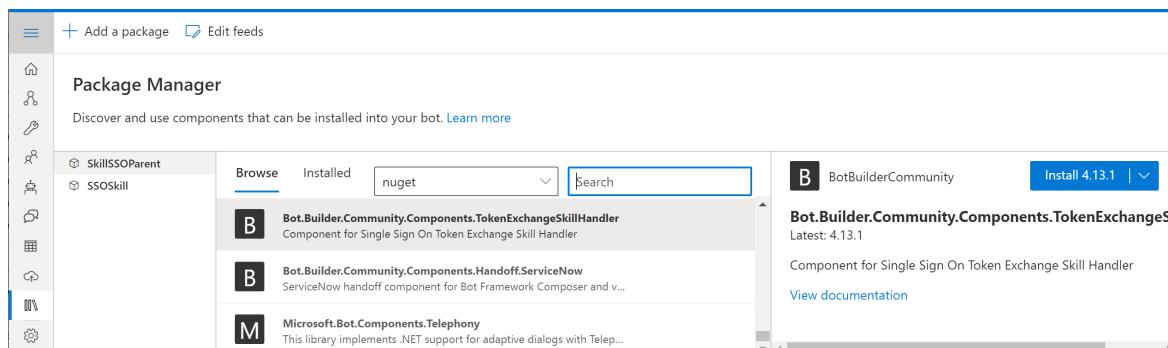
## Prerequisites

- A subscription to [Microsoft Azure](#). If you don't have an Azure subscription, you can create a free account.
- [A basic bot built with Composer](#).
- A good understanding of [skills](#) in the Bot Framework SDK.
- Install [ngrok](#).
- A local or remote skill, such as the one described in how-to [Export a skill](#).
- Single sign on configured *root bot* and *skill bot* described in [Add single sign on to a bot](#), up to the point of the sample.

## Add the TokenExchangeSkillHandler package

Your skill and root bot Azure AD applications must be configured for OAuth token exchange, and the bot's must be configured with correct OAuth input connection settings.

1. Open your root bot project.
2. Add the [Bot.Builder.Community.Components.TokenExchangeSkillHandler](#) package to the *root bot* through the [Composer Package manager](#).



## Configure the TokenExchangeSkillHandler in the root bot

Once the package is installed, you need to configure your root bot.

1. Go to the **Configure** view for your root bot.
2. Switch to the **Advanced Settings View (json)**.
3. Make sure the component is added to the `components` array. For example:

```
"components": [  
    {  
        "name": "Bot.Builder.Community.Components.TokenExchangeSkillHandler",  
        "settingsPrefix": "Bot.Builder.Community.Components.TokenExchangeSkillHandler"  
    }  
,
```

4. To the root of the bot's JSON object, add configuration information for the token exchange handler:

```
"Bot.Builder.Community.Components.TokenExchangeSkillHandler": {  
    "useTokenExchangeSkillHandler": true,  
    "tokenExchangeConnectionName": "YourTokenExchangeConnectionName"  
,
```

5. Republish your root bot.

Now, your root bot can share its OAuth token with the skill.

## Further reading

- [Create a skill](#)
- [Connect to a remote skill](#)
- [Publish a skill to Azure](#)

# Test and debug bots using Bot Framework Web Chat

5/20/2021 • 3 minutes to read

**APPLIES TO:** Composer v1.x and v2.x

Since the Bot Framework Composer v1.4 release, you can use Bot Framework Web Chat within the Composer environment to test and debug your bots. This article will show you how to use Web Chat in Composer to test a bot locally, how to save a transcript from your Web Chat test for debugging, and how to use the [Web Chat Inspector](#) to view trace activities and bot state.

## NOTE

With the current Web Chat integration:

- You can test a local [root bot](#) directly.
- You can test a local [skill](#) indirectly, through the root bot.

However, you need to test remote bots and skills in the Emulator.

## Prerequisites

- Install [Bot Framework Composer](#) 1.4 and above.
- [A basic bot built using Composer](#).

## Run and test a bot locally

1. Select the **Start bot** button on the top right to start your bot's local runtime.



2. Once your bot is running the **Local bot runtime manager** opens and will show your bot's status as *running*.

### Local bot runtime manager

Start and stop local bot runtimes individually.

Bot	Status	
RootBot	Running	<a href="#"> Open Web Chat</a>
SkillBot (Skill)	Running	<a href="#"> Test in Emulator</a>

## NOTE

If you see a *Windows Security Alert*, your firewall has blocked some features of this app. Select **Allow access**.

3. Select **Open Web Chat** to the right of your root bot or select the **Open Web Chat** () icon from the toolbar. This will open the **Web Chat** window where you interact with your root bot. You can test

different types of message activities, including text, cards and suggested actions.

The screenshot shows the Microsoft Bot Framework Web Chat interface. At the top, it says "CardBot". Below that is a toolbar with icons for "Restart Conversation - new user ID" (a blue circle with a white arrow), a dropdown menu, a file icon, and a copy icon. The main area shows a message from the bot: "Welcome to Card Samples Bot." followed by a blue rectangular card with the word "Hello!" in white. Below the card, the timestamp "A minute ago" is shown. A sidebar on the left lists options for displaying cards: 1. HeroCard, 2. HeroCardWithMemory, 3. ThumbnailCard, 4. SigninCard, 5. AnimationCard, 6. VideoCard, 7. AudioCard, 8. AdaptiveCard, and 9. AllCards. At the bottom, there's a text input field with a paperclip icon and the placeholder "Type your message", and a send button with a right-pointing arrow.

4. Select **Restart Conversation** to restart a conversation. Use the drop-down menu to the right of the button to choose whether to start the conversation with the same user ID or a new user ID.

The screenshot shows the Microsoft Bot Framework Web Chat interface. At the top, it says "CardBot". Below that is a toolbar with icons for "Restart Conversation - new user ID" (a blue circle with a white arrow), a dropdown menu (highlighted with a red box), "Restart Conversation - same user ID" (a blue circle with a white arrow), and "Restart Conversation - new user ID" (a blue circle with a white arrow). The dropdown menu is open, showing the three options listed above.

## Save a transcript for debugging

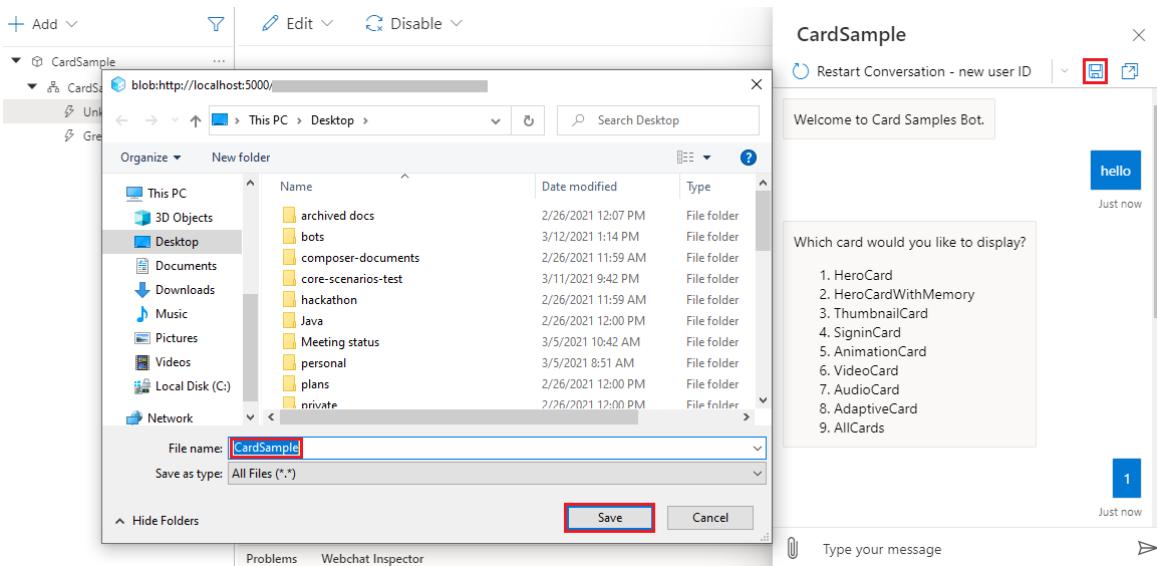
A bot transcript file is a specialized JSON file. It preserves the interactions between a user and your bot, including the contents of a message, interaction details such as the user id, channel id, channel type, channel capabilities, time of the interaction, and more. All of this information can be used to help find and resolve issues when testing and debugging your bot.

This section walks you through the process to save a transcript from your **Web Chat** test and use the transcript to debug your bot.

1. Select the **Save chat transcripts** () icon in the upper-right of the **Web Chat** window.

The screenshot shows the Microsoft Bot Framework Web Chat interface. At the top, it says "CardBot". Below that is a toolbar with icons for "Restart Conversation - new user ID" (a blue circle with a white arrow), a dropdown menu, a file icon () highlighted with a red box, and a copy icon.

2. Choose a location to save the transcript file.



3. Double-click the saved transcript (blob) file to open it in the Emulator. With your transcript file loaded, you're now ready to debug interactions that you captured between a user and your bot.
4. Select any event or activity from the conversation window on the left. In the example shown below, the user's first interaction when they sent the message "Hi" was selected.

The JSON window shows the following message object:

```

root:
  > channelData:
    text: "hi"
    textFormat: "plain"
    type: "message"
    channelId: "emulator"
  > from:
    locale: "en-us"
    timestamp: "2021-03-12T20:11:07.650Z"
  > entities:
    id: "21d0369f-4d61-4d75-92e4-23f59c99fc61"
  > recipient:
    localTimestamp: "2021-03-12T12:11:07-08:00"
  > conversation:
    serviceUrl: "http://localhost:5000"
  
```

The terminal window shows the following log entries:

```

[12:13:21] -> conversationUpdate
[12:13:21] <- message Welcome to Card Samples Bot.
[12:13:21] <- trace Bot State
[12:13:21] -> message hi
[12:13:21] <- message Which card would you like to display? 1. HeroC...
[12:13:21] <- trace Bot State
[12:13:21] -> message 1
[12:13:21] <- message application/vnd.microsoft.card.hero
[12:13:21] <- trace Bot State
[12:13:21] -> message 2
[12:13:21] <- message Which card would you like to display? 1. HeroC...
[12:13:21] <- trace Bot State
  
```

Looking at some of these values from the JSON window on the top right, you will see some detailed information such as the:

- Plain text sent contained "Hi".
- Interaction type was message.
- Time the message was sent.

- User ID and information.

This detailed information allows you to follow the step-by-step interactions between the user's input and your bot's response, which is useful for debugging situations.

## Web Chat Inspector

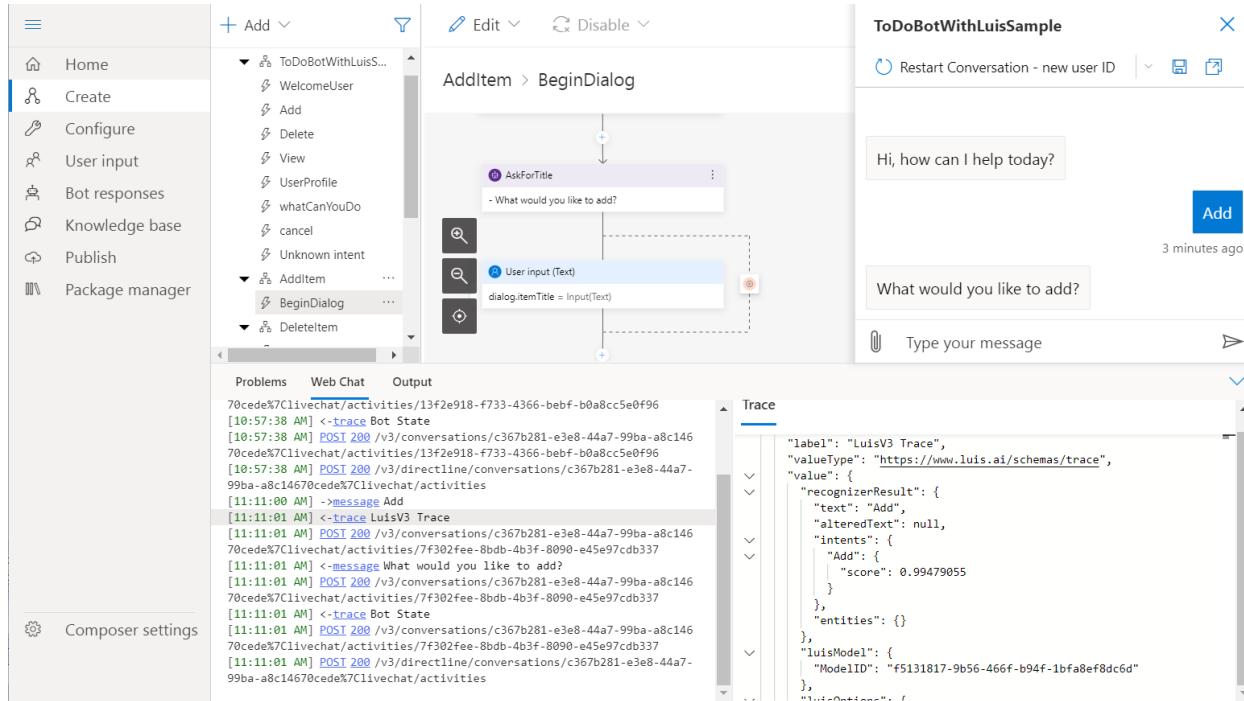
- [Composer v2.x](#)
- [Composer v1.x](#)

You can view bot state, LUIS, QnA and other trace activities in the **Web Chat Inspector** panel, which is located in the Diagnostic pane at the bottom of the Composer window.

The **Problems** panel logs authoring errors and warnings such as incorrect syntax or a missing configuration file.

The **Web Chat Inspector** panel logs errors and warnings that happen between Composer and the bot. For example, if you enter an invalid app ID or password, you will see the errors in the **Web Chat Inspector** panel.

The **Output** panel shows the standard output/error stream, when the bot is running, from the dotnet or JavaScript runtime. This includes start up information and runtime errors.



# Debugging and validation

5/20/2021 • 4 minutes to read

**APPLIES TO:** Composer v1.x and v2.x

This article introduces the validation functionality provided in Bot Framework Composer. The validation functionality helps you identify syntax errors and provide suggested fixes when you author [.lg](#) files, [.lu](#) files, and [expressions](#) during the process of developing a bot using Composer. With the help of the validation functionality, your bot-authoring experience will be improved and you can easily build a functional bot that can run correctly.

## Prerequisites

- [Install Bot Framework Composer](#).
- A basic understanding of [language generation](#).
- A basic understanding of [language understanding](#).
- A basic understanding of [adaptive expressions](#).

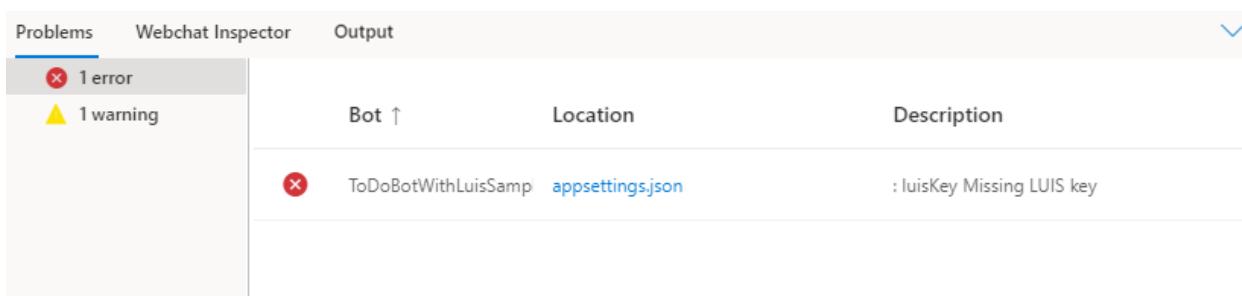
## Diagnostics

When there are errors in Composer, the button to start the bot will be disabled.



Selecting the error or warning icon will navigate you to the **Problems** pane at the bottom of Composer where you can view a full list of the errors and warnings.

- [Composer v2.x](#)
- [Composer v1.x](#)

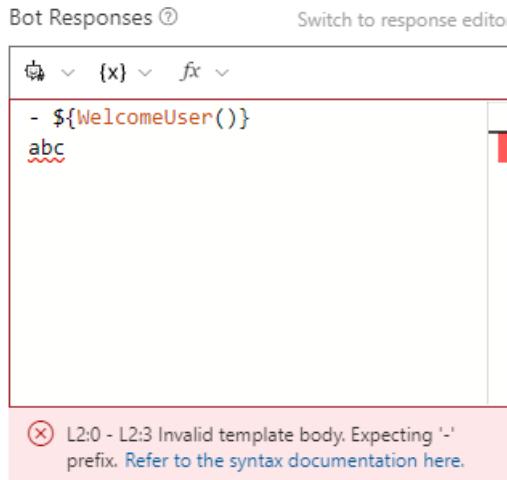


Starting in version 2.0 of Composer there is also an **Output** pane that shows the standard output/error stream, when the bot is running, from the dotnet or JavaScript runtime. This includes start up information and runtime errors.



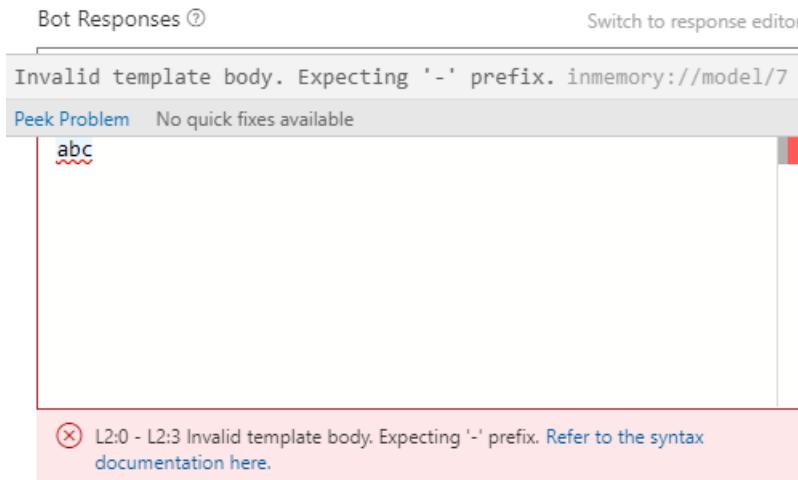
## .lg files

When you author an [.lg file](#) that has syntax errors, a red wavy line will show under the error in the **code editor** under **Bot Responses** on the right, and the border of the box will appear red. Note that you are unable to switch to the **response editor** until you fix the template errors.



In the example .lg template above, *abc* is invalid. Do the following to diagnose and fix the error:

1. Read the error message beneath the editor and click [Refer to the syntax documentation here](#) to refer to the syntax documentation.
2. Hover your mouse over the erroneous part and read the detailed error message with suggested fixes.



### NOTE

If you don't find the error message helpful, read the [.lg file format](#) article and use the correct syntax to compose the language generation template.

Select **Bot Responses** on the Composer menu on the left side and select **Show code**. You will find the error is also saved and updated here, seen below.

[Hide code](#)

The screenshot shows a code editor window for an LG template. The code is as follows:

```
[import](common.lg)

# SendActivity_Welcome()
- ${WelcomeUser()}
abc

# SendActivity_003038
- You said '${turn.activity.text}'
```

A red rectangle highlights the character 'c' in the word 'abc'. A red message bar at the bottom states: "✖ L5:0 - L5:3 Invalid template body. Expecting '-' prefix. Refer to the syntax documentation here."

- The tiny red rectangle on the right end of the editor helps you to identify where the error is. This is especially helpful when you have a long list of templates.
- The error message at the bottom of the editor indicates the line numbers of the error. In this example, **line5:0 - line5:3** means the error locates in the fifth line of the editor from the first character (indexed 0 )to the fourth character (indexed 3).

Hover your mouse over the erroneous part you will see the detailed error message with suggested fixes.

The screenshot shows the same LG template editor. A tooltip has appeared over the character 'c' in 'abc'. The tooltip contains the text: "Invalid template body. Expecting '-' prefix. inmemory://model/12". Below the tooltip, a message says "Peek Problem No quick fixes available".

In this example, the error message indicates a - is missing in the template. After you add the - sign in the lg template, you will see the error message disappear.

The screenshot shows the LG template editor with the corrected code:

```
[import](common.lg)

# SendActivity_Welcome()
- ${WelcomeUser()}
- abc

# SendActivity_003038
- You said '${turn.activity.text}'
```

If you go back to the **code editor** you will see the change is updated and error disappear as well. You will also be able to switch back to the **response editor**.

## WARNING

Switching from the **code editor** to the **response editor** will delete any templates that aren't activity response templates. Be cautious when switching between the two editors.

## .lu files

When you create an **Intent recognized** trigger and your .lu file has syntax errors, a red wavy line will show under the error in the **Trigger phrases**, and the border of the box will appear red.

### Create a trigger

What is the type of this trigger?

Intent recognized

What is the name of this trigger (LUIS)

Greeting

Trigger phrases

- Hi  
- Hello  
Hey

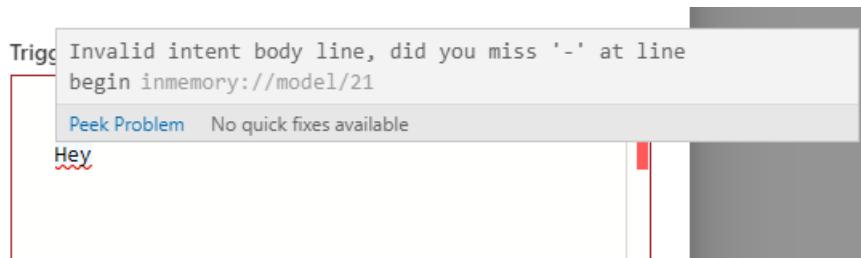
( line 4:0 - line 4:3 Invalid intent body line, did you miss '-' at line begin [Refer to the syntax documentation here.](#)

Cancel

Submit

Do the following to diagnose and fix the error:

1. Selecting the [Refer to the syntax documentation here](#) link in the error will open the [.lu file format](#) article.
2. Hover your mouse over the erroneous part and read the detailed error message with suggested fixes.



## NOTE

If you don't find the error message helpful, read the [.lu file format](#) article and use the correct syntax to compose the language understanding template.

## Expressions

When you send a response with an invalid [expression](#), a warning will appear at the top right of Composer. If you are in the **code editor** view, you will see a red wavy line underneath the erroneous expression. The box around the expression will appear yellow, as seen below.

The screenshot shows the Composer interface with a code editor window. The code is as follows:

```
[Activity
| Text = ${binary(value)}
]
```

A red wavy underline is under the word "binary". A yellow border surrounds the entire code block. Below the editor, a tooltip provides the error message:

L2:11 - L2:27 Property 'Text=\${binary(value)}':  
Error occurred when parsing expression  
'binary(value)'. binary does not have an  
evaluator, it's not a built-in function or a  
custom function. Please add unknown  
functions to setting's customFunctions  
field. [Refer to the syntax documentation here](#).

To diagnose and fix the error, read the error message and select [Refer to the syntax documentation here](#) to be taken to the syntax documentation. For more information see [Adaptive expressions syntax](#).

Once the expression warning is fixed, the red wavy line under the expression will disappear, and the border around the editor will appear black.

## Custom functions

For users who have custom functions defined in a custom runtime, additional configuration is required in the `customFunctions` field of the **Bot Settings** page. If you do not have the `customFunctions` setting configured properly, Composer will identify custom functions as errors, and the **Start Bot** button (or the **Restart Bot** button) will be disabled.

To configure the `customFunctions` setting:

1. Select **Project Settings** from Composer menu then toggle **Advanced Settings View (json)**.
2. In the `customFunctions` field, add the names of the custom functions. For example:

```
"customFunctions": [
    "AskBot.stripTags"
]
```

### TIP

You can set the `customFunctions` field to `functionPrefix.*` to support validation of custom functions with a specific prefix, for example: `AskBot.*`.

# Capture your bot's telemetry

5/20/2021 • 3 minutes to read

**APPLIES TO:** Composer v1.x and v2.x

Bot Framework Composer enables your bot applications to send event data to a telemetry service such as [Application Insights](#). Telemetry offers insights into your bot by showing which features are used the most, detects unwanted behavior and offers visibility into availability, performance, and usage. In this article you will learn how to implement telemetry into your bot using Application Insights.

## NOTE

You may want to look at the related Bot Framework SDK article [Add telemetry to your bot](#).

## Prerequisites

- A subscription to [Microsoft Azure](#).
- [A basic bot built using Composer](#).
- Basic knowledge of [Kusto queries](#).
- How to use [Log Analytics](#) in the Azure portal to write Azure Monitor log queries.
- The basic concepts of [Log queries](#) in Azure Monitor.

## Create an Application Insights resource

Azure *Application Insights* displays data of your application in a Microsoft Azure resource. Creating a new resource is part of setting up *Application Insights* to monitor an application. After creating your new resource, you can get its *instrumentation key* and use it to configure settings in Composer. The *instrumentation key* links your telemetry to the resource.

## TIP

You can learn more about how to create an *Application Insight* resource and get the *Instrumentation key* by reading [this article](#).

## Update settings in Composer

To connect to your *Application Insights* resource in Azure, you need to add the instrumentation key to the `applicationInsights` section of the **Project Settings** page. To do this:

1. Go the **Project Settings** page.
2. Toggle **Advanced Settings View (json)** on the right side of the screen.
3. Find the `applicationInsights` section, then add your Application Insights instrumentation key to the `instrumentationKey` setting.

```

    "feature": {
      "UseShowTypingMiddleware": false,
      "UseInspectionMiddleware": false,
      "RemoveRecipientMention": false
    },
    "MicrosoftAppPassword": "",
    "MicrosoftAppId": "[REDACTED]",
    "cosmosDb": {
      "authKey": "",
      "collectionId": "botstate-collection",
      "cosmosDBEndpoint": "",
      "databaseId": "botstate-db"
    },
    "applicationInsights": {
      "InstrumentationKey": "[REDACTED]"
    },
    "blobStorage": {
      "connectionString": "",
      "container": "transcripts"
    },
    "luis": {
      "name": "",
      "authoringKey": "",
      "authoringEndpoint": "",
      "endpointKey": ""
    }
  }
}

```

## Analyze bot's behavior

After making these changes to include the *instrumentation key*, you can run and interact with your bot to generate telemetry data. To see this telemetry data, navigate to the *Logs* section of your *Application Insights* resource in Azure.

For example, if you run a simple specified `customEvents` as a query, which shows all custom events, but you can narrow down the events or fields you want to see by providing different queries. see [Analyze your bot's telemetry data](#) for additional information on creating custom queries.

timestamp [UTC]	name	itemType	customDimensions	customMeasurements	operation_Name
9/16/2020, 5:47:08.759 PM	ValueRecognizerResult	customEvent	{"AspNetCoreEnvironment": "Development", "conversationId": "8fb415f0-f844-11ea-b426-c97631751c78 livechat", "activityType": "message", "activityId": "a44ae0e0-f844-11ea-b426-c97631751c78 livechat", "score": 1.0}		POST Bot/Post
9/16/2020, 5:47:09.647 PM	BotMessageSend	customEvent	{"AspNetCoreEnvironment": "Development", "conversationId": "8fb415f0-f844-11ea-b426-c97631751c78 livechat", "activityType": "message", "activityId": "a44ae0e0-f844-11ea-b426-c97631751c78 livechat", "score": 1.0}		POST Bot/Post
9/16/2020, 5:47:09.670 PM	QnAMakerRecognizerResult	customEvent	{"AspNetCoreEnvironment": "Development", "conversationId": "8fb415f0-f844-11ea-b426-c97631751c78 livechat", "activityType": "message", "activityId": "a44ae0e0-f844-11ea-b426-c97631751c78 livechat", "score": 1.0}		POST Bot/Post
			AdditionalProperties: {"answers": [{"questions": ["How can I improve the throughput performance for query predictions?"], "answer": "***Answer***:Throughput performance issues indicate you need to scale up for both your App service and your Cognitive Search. Consider adding a replica to your QnA Maker Knowledge Base and the score etc."}], "answers": [{"questions": ["How can I improve the throughput performance for query predictions?"], "answer": "***Answer***:Throughput performance issues indicate you need to scale up for both your App service and your Cognitive Search. Consider adding a replica to your QnA Maker Knowledge Base and the score etc."}]}		
			Answers: [{"questions": ["How can I improve the throughput performance for query predictions?"], "answer": "***Answer***:Throughput performance issues indicate you need to scale up for both your App service and your Cognitive Search. Consider adding a replica to your QnA Maker Knowledge Base and the score etc."}]		
			0: {"questions": ["How can I improve the throughput performance for query predictions?"], "answer": "***Answer***:Throughput performance issues indicate you need to scale up for both your App service and your Cognitive Search. Consider adding a replica to your QnA Maker Knowledge Base and the score etc."}		
			AspNetCoreEnvironment: Development		
	Entities	customEvent	{"answer": "***Answer***:Throughput performance issues indicate you need to scale up for both your App service and your Cognitive Search. Consider adding a replica to your QnA Maker Knowledge Base and the score etc."}		
	Intents	customEvent	{"QnAMatch": {"score": 1.0}}		

As standard you can track a number of events, including bot messages sent or received, LUIS results, dialog events (started / completed / cancelled) and QnA Maker events. Specifically for QnA Maker, you can filter down to events named `QnAMakerRecognizerResult`, which will include the original query, the top answers from the QnA Maker Knowledge Base and the score etc.

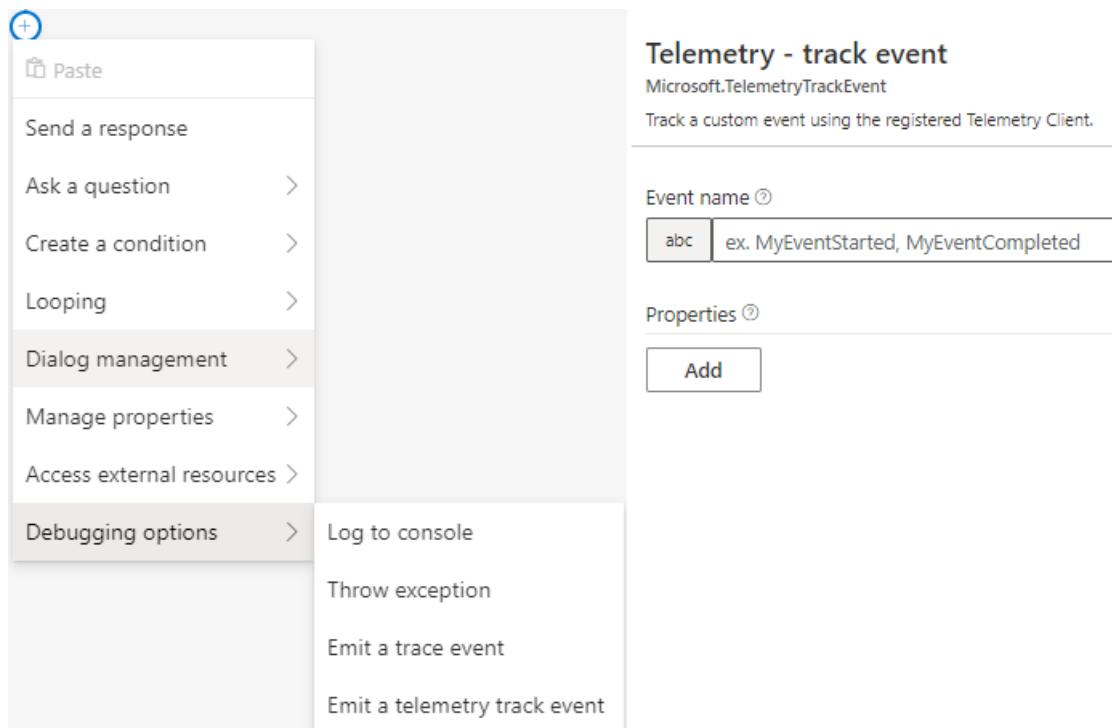
Once you are gathering telemetry from your bot, you can also try using Power BI template, which contains some QnA tabs, to view your data. The template was built for use with the Virtual Assistant template, and you can find details of this [here](#).

## Additional information

In Composer, there are two additional settings in the app settings that you need to be aware of: *logActivities* and *logPersonalInformation*. *logActivities*, which is set to true by default, determines if your incoming or outgoing activities are logged. *logPersonalInformation*, which is set to false by default, determines if more sensitive information is logged. You may see some of the fields blank if you do not enable this.

```
"qna": {  
  "subscriptionKey": "",  
  "knowledgebaseid": "",  
  "endpointKey": "",  
  "hostname": "",  
  "qnaRegion": "westus"  
},  
"telemetry": {  
  "logPersonalInformation": false,  
  "logActivities": true  
},  
"runtime": {  
  "key": "",  
  "name": "",  
  "path": "",  
  "command": "",  
  "customRuntime": true  
},
```

Since Composer 1.1.1 release, Composer features a new action for sending additional events to *Application Insights*, alongside those that you can automatically capture as described above. Wherever you want to track a [custom event](#), you can add the **Emit a telemetry track event** action, which can be found under the **Debugging Options** menu. Once added to your authoring canvas, you specify a custom name for the event, which is the name of the event that will appear in the `customEvents` table referenced above, along with optionally specifying one or more additional properties to attach to the event.



## Further reading

- [Analyze your bot's telemetry data.](#)

# How to provision Azure resources

5/20/2021 • 4 minutes to read

**APPLIES TO:** Composer v2.x

This article describes how to provision Azure resources by IT administrators to satisfy the request submitted by users that do not have provisioning abilities.

## NOTE

The information about generating an handoff request by users without provisioning abilities can be found in the [Publish a bot to Azure](#) article.

## Prerequisites

- A subscription to [Microsoft Azure](#).
- [Node.js](#). Use version 12.13.0 or later.
- Latest version of the [Azure CLI](#).
- PowerShell version 6.0 and later.

## Run the provisioning command

After you receive the **resource provisioning** request from your user, follow the steps described below.

1. Download and unzip the [provisioning helper scripts](#).
2. Open a new command prompt and navigate to the newly create **scripts** folder. For example:

```
cd C:\Users\UserName\Documents\provisionScripts
```

3. Run the following command to install the dependencies. You must run this command every time you install a new version of the scripts.

```
npm install
```

4. Run the provisioning command shared by your requester to provision new Azure resources. The command structure is shown below. The provisioning process will take a few minutes to complete.

```
node provisionComposer.js --subscriptionId=<YOUR AZURE SUBSCRIPTION ID> --name=<YOUR RESOURCE NAME> -  
-appPassword=<APP PASSWORD> --environment=<NAME FOR ENVIRONMENT DEFAULT to dev> --location=<LOCATION>  
--createLuisResource=true --createLuisAuthoringResource=true --createCosmosDb=false --  
createStorage=false createAppInsights=true --CreateQnAResource=true
```

PARAMETER	DESCRIPTION	REQUIRED
<code>subscriptionId</code>	Your Azure subscription ID.	Yes
<code>name</code>	Azure resource name.	Yes

PARAMETER	DESCRIPTION	REQUIRED
<code>appPassword</code>	Azure resource password. It must be at least 16 characters long, contain at least 1 upper or lower case alphabetical character, and contain at least 1 special character.	Yes
<code>environment</code>	Default to <code>dev</code>	Optional
<code>location</code>	Azure resource group region. Default to <code>westus</code> .	Optional
<code>tenantId</code>	ID of your tenant if required.	If default value does not apply you get <i>user was not found</i> error.
<code>createLuisResource</code>	The LUIS prediction resource to create. Region is default to <code>westus</code> and cannot be changed. Default to <code>true</code> .	
<code>createLuisAuthoringResource</code>	The LUIS authoring resource to create. Region is default to <code>westus</code> and cannot be changed. Default to <code>true</code> .	Optional
<code>createQnAResource</code>	The QnA resource to create. Default to <code>true</code> .	Optional
<code>createCosmosDB</code>	The CosmosDB resource to create. Default to <code>true</code> .	Optional
<code>createStorage</code>	The BlobStorage resource to create. Default to <code>true</code> .	Optional
<code>createAppInsights</code>	The AppInsights resource to create. Default to <code>true</code> .	Optional

5. As the Azure resources are being provisioned, you will see a spinning activity indicator for a few minutes.

Once completed, a **publishing profile** in JSON format is displayed.

6. Copy and save this profile. Then send it the user that made the provisioning request.

The process concludes when in Composer the user will publish the bot using the *publishing profile* received in the importing existing resources modality.

## Share resource details

Once completed, a **publishing profile** in JSON format is displayed in the command line terminal.

1. Copy and save this profile.
2. Send it to the user that made the provisioning request.

The following is an example of the JSON file format.

```
{
  "name": "<NAME OF YOUR RESOURCE GROUP>",
  "environment": "<ENVIRONMENT>",
  "hostname": "<NAME OF THE HOST>",
  "luisResource": "<NAME OF YOUR LUIS RESOURCE>",
  "settings": {
    "applicationInsights": {
      "InstrumentationKey": "<SOME VALUE>"
    },
    "cosmosDb": {
      "cosmosDBEndpoint": "<SOME VALUE>",
      "authKey": "<SOME VALUE>",
      "databaseId": "botstate-db",
      "collectionId": "botstate-collection",
      "containerId": "botstate-container"
    },
    "blobStorage": {
      "connectionString": "<SOME VALUE>",
      "container": "transcripts"
    },
    "luis": {
      "endpointKey": "<SOME VALUE>",
      "authoringKey": "<SOME VALUE>",
      "region": "westus"
    },
    "qna": {
      "endpoint": "<SOME VALUE>",
      "subscriptionKey": "<SOME VALUE>"
    },
    "MicrosoftAppId": "<SOME VALUE>",
    "MicrosoftAppPassword": "<SOME VALUE>"
  }
}
}
```

Navigate to your Azure portal, you should be able to see the resources created like the following:

RESOURCE	REQUIRED/OPTIONAL	DESCRIPTION
Application Registration	Required	Required registration allowing your bot to communicate with Azure services.
App Service	Required	Quickly build, deploy, and scale enterprise-grade web, mobile, and API apps running on any platform.
Bot Channels Registration	Required	Build, connect and manage bots to interact with your users from your app or website to Cortana, Skype, Messenger and many other services.
Azure Cosmos DB	Optional	Azure cosmos is a multi-model database service backed up by comprehensive SLAs. Used for bot state retrieving.
Application Insights	Optional	Application performance, availability and usage information at your fingertips. Used for bot chatting data analyzing.

RESOURCE	REQUIRED/OPTIONAL	DESCRIPTION
Azure Blob Storage	Optional	Scalable, durable cloud storage, backup and recovery solutions for any data, big or small. Used for bot transcripts logging.
Luis authoring resource (Cognitive Services)	Optional	Luis is an NLP service that enables you to understand human language in your own application. Used for Luis endpoint hitting.
Luis prediction resource (Cognitive Services)	Optional	Luis is an NLP service that enables you to understand human language in your own application. Used for Luis endpoint hitting.
QnA Maker resource (Cognitive Services)	Optional	Use QnA Maker to build a knowledge base by extracting questions and answers from your content, such as FAQs, documents, etc.

You should then securely share the output JSON with the requester. Once the requester receives the resource details, he or she can use these resources to [import existing Azure resources](#) and [publish](#) their bots.

## Additional information

- [How to manually provision Azure resources for your Composer project](#). Learn more about how to run scripts to provision Azure resources and publish a bot.

# Admin Handoff for LUIS

5/20/2021 • 3 minutes to read

**APPLIES TO:** Composer v2.x

In many cases, such as adding an intent trigger for a LUIS recognizer bot or using a template that requires LUIS, it's required to provide a LUIS Authoring key to Composer to allow for your bot project to launch properly.

Many users won't have the right to create a LUIS service for their bot project as they don't have rights to an Azure Tenant to create objects. The following will walk a user through the process of generating an Admin Handoff request.

## Requesting LUIS creation via Admin Handoff

In Composer there are two ways to initiate the LUIS provisioning for your local development environment.

The first is to go to the **Configure** tab on the navigation on the left column and choose the tab called **Development resources**. Under this section select the **Set up Language Understanding** button on the bottom.

Overview    **Development resources**    Connections    Skill configuration    Localization

### Azure Language Understanding

Language Understanding (LUIS) is an Azure Cognitive Service that uses machine learning to understand natural language input and direct the conversation flow. [Learn more](#). Use an existing Language Understanding (LUIS) key from Azure or create a new key. [Learn more](#)

Application name [?](#)

Empty\_5

Language Understanding authoring key [?](#)

Type Language Understanding authoring key

Language Understanding region [?](#)

Select region

**Set up Language Understanding**

The other way is if you have added any triggers to your bot project or used a template that requires LUIS, you will be able to see a required action for **Set up Language Understanding** in the **Getting Started** experience.



## Get started    Learn more

X

### Required

#### ! Set up Language Understanding

Use machine learning to understand natural language input and direct the conversation flow. [Learn more](#)

Once you initiate the LUIS setup process, you select the option for **Generate instructions for Azure administrator** and select **Next**. You will then be given a screen that will allow you to copy and paste the instructions for your Azure Administrator to complete the process needed to provision your needed resources.

### Generate instructions for Azure administrator

X

If Azure resources and subscription are managed by others, use the following information to request creation of the resources that you need to build and run your bot.

#### Instructions



I am creating a conversational experience using Microsoft Bot Framework project. For my project to work, it needs Azure resources including Language Understanding. Below are the steps to create these resources.

1. Using the Azure portal, please create a Language Understanding resource.
2. Once created, securely share the resulting credentials with me as described in the link below.

Detailed instructions:  
<https://aka.ms/bfcomposerhandoffluis>

Back

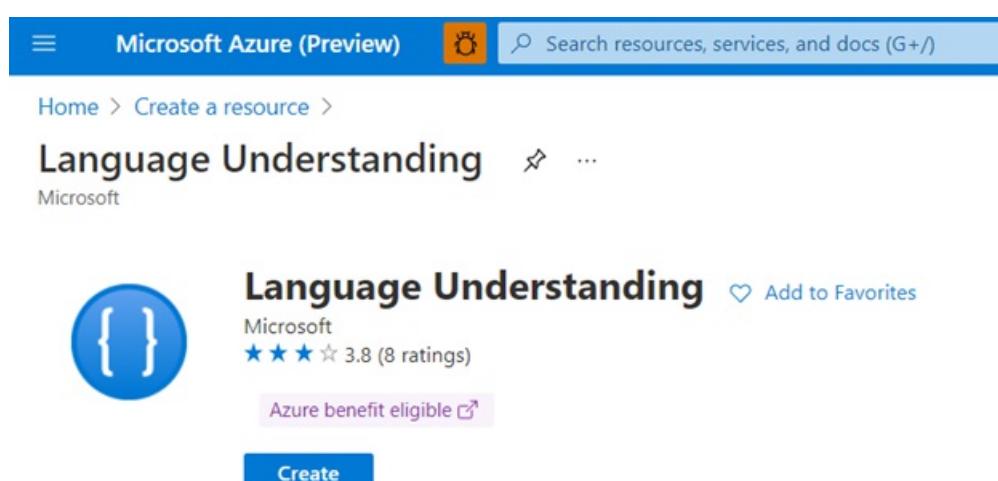
Okay

## Execute the Azure Admin request for a LUIS request from Composer

As an Azure Admin, you may be requested to create a LUIS resource for use with Composer. To complete these actions, you will need to log on to the Azure Portal and create a Language Understanding resource.

### Create LUIS Authoring resource

1. In Azure, search for **Language Understanding**.



The screenshot shows the Microsoft Azure (Preview) portal with the following details:

- Header:** Microsoft Azure (Preview), Search bar: Search resources, services, and docs (G+/)
- Breadcrumbs:** Home > Create a resource >
- Page Title:** Language Understanding
- Resource Type:** Microsoft
- Icon:** A blue circle with white curly braces ({{}}).
- Name:** Language Understanding
- Rating:** ★★★★ 3.8 (8 ratings)
- Status:** Azure benefit eligible
- Actions:** Add to Favorites, Create

2. Select **Create** and choose the **Authoring** option in the **Create options** selection.

3. Ensure that you complete the required items including the **Subscription**, **Resource group**, **Name**, **Authoring location** and **Authoring pricing tier**. Then select the **Review + create** button.

You will then be provided with a summary screen that you will need to verify what you have entered. Select **Create**.

Once the resource has been created you will be presented with a confirmation screen and you will need to select **Go to resource**. This will take you to the LUIS Authoring Service that was created.

#### Provide requestor with resource details

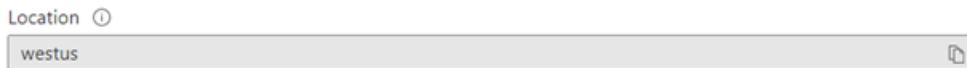
Navigate in the Azure Portal to the LUIS Authoring Service that was created above and collect the following information:

1. Go to **Keys and Endpoints** and select the **Copy** button next to the **KEY 1**.



Copy this key to a secure location that you plan to share with your requestor. This is secure data and should be treated as such.

2. On the **Keys and Endpoint** screen also copy the **Location**.



Copy this to the same secure location as the key.

3. Now securely send this information back to the requestor of the resource.

## Update Composer's local development environment manually with your LUIS information

Once you securely receive the response from your Azure administrator, you will need to complete the following steps to ensure that your local development environment is setup for use in Composer.

1. Navigate to the **Configure** tab on the left column and then select the tab called **Development resources**.
2. Under **Azure Language Understanding** you will need to copy the **Luis authoring key** your administrator provided and select the region that was provided.

This should remove the required item from the **Getting Started** experience and will also remove the errors warning that your LUIS configuration wasn't complete.

Be aware that there is a limit to the number of queries that you can run per month using only an Authoring key. If you happen to get a **403 error** warning that you are out of quota for the month, you can publish your bot to a publishing profile to allow for more queries to be run.

# Admin Handoff for QnA Maker

5/20/2021 • 2 minutes to read

**APPLIES TO:** Composer v2.x

In many cases, such as adding a knowledge base to an existing bot or using a template that requires QnA Maker, it's required to provide a QnA Maker key to Composer to allow for your bot project to launch properly.

Many users won't have the right to create a QnA Maker resource for their bot project as they do not have rights to an Azure Tenant to create resources. The following will walk a user through the process of generating an Admin Handoff request.

## Requesting a QnA Maker creation via Admin Handoff

In Composer there are two ways to initiate the LUIS provisioning for your local development environment.

The first is to go to the **Configure** tab on the navigation on the left column and choose the tab called **Development resources**. Under this section select the **Set up QnA Maker** button on the bottom.

### Azure QnA Maker

QnA Maker is an Azure Cognitive services that can extract question-and-answer pairs from a website FAQ. [Learn more](#). Use an existing key from Azure or create a new key. [Learn more](#).

QnA Maker Subscription key [?](#)

**Set up QnA Maker**

The other way is if you have added a knowledge base to your bot project or used a template that requires a knowledge base, you will be able to see a required action in the **Getting Started** experience for **Set up QnA Maker** and selecting on this item.

Required

**!** Set up QnA Maker

Use Azure QnA Maker to create a simple question-and-answer bot from a website FAQ. [Learn more](#)

Once you initiate the QnA Maker setup process, you select the option for **Generate instructions for Azure administrator** and select **Next**. You will then be given a screen that will allow you to copy and paste the instructions for your Azure administrator to complete the process needed to provision your needed resources.

## Generate instructions for Azure administrator

X

If Azure resources and subscription are managed by others, use the following information to request creation of the resources that you need to build and run your bot.

### Instructions



I am creating a conversational experience using Microsoft Bot Framework project. For my project to work, it needs Azure resources including QnA Maker. Below are the steps to create these resources.

1. Using the Azure portal, please create a QnAMaker resource on my behalf.
2. Once provisioned, securely share the resulting credentials with me as described in the link below.

Detailed instructions:  
<https://aka.ms/bfcomposerhandoffqnamaker>

Back

Okay

## Execute the Azure Admin request for a QnA Maker request from Composer

As an Azure administrator, you may be requested to create a QnA Maker resource for use with Bot Framework Composer. To complete these actions, you will need to log on to the Azure Portal and create a QnA Maker resource.

### Create QnA Maker resource

1. In the Azure portal, search for *QnA Maker*.

The screenshot shows the Microsoft Azure (Preview) portal. At the top, there's a navigation bar with 'Microsoft Azure (Preview)', 'Report a bug', and a search bar. Below the navigation bar, the URL 'Home > Create a resource >' and the service name 'QnA Maker' are visible. A 'Create' button is prominently displayed. The main content area shows the 'QnA Maker' service details, including a rating of 4.2 (171 ratings), an 'Add to Favorites' button, and an 'Azure benefit eligible' link. Below this, there are tabs for 'Overview', 'Plans', 'Usage Information + Support', and 'Reviews'. A detailed description of the QnA Maker service follows.

QnA Maker is a cloud-based API service that lets you create a conversational question-and-answer layer over your existing data. Use it to build a knowledge base by extracting questions and answers from your semi-structured content, including FAQs, manuals, and documents. Answer users' questions with the best answers from the QnAs in your knowledge base -- automatically. Your knowledge base gets smarter, too, as it continually learns from user behavior.

2. Select **Create** and ensure that you don't check the box for **Managed (preview)**. This isn't currently supported in Composer.

**Managed (preview)**

If you select managed, telemetry and compute will be included automatically with your QnA Maker resource. If you do not select managed, you will be prompted to create an App Insights and App Service resources for the required telemetry and compute that you will have to manage for your QnA Maker resource. Read more [here](#).

3. Ensure that you complete the required items. It's suggested that you keep all the resources in the same data center location. Then select the **Review + create** button.
4. You will then be provided with a summary screen to verify what you have entered. Then select **Create**.
5. Once the resource has been created you will be presented with a confirmation screen and you will need to select **Go to resource**. This will take you to the QnA Maker resource that was created.

## Provide requestor with resource details

Navigate in the Azure Portal to the QnA Maker resource that was created above and collect the following information.

1. Go to **Keys and Endpoints** and select the **Copy** button next to **KEY 1**.



2. Copy this to a secure location that you plan to share with your requestor. This is secure data and should be treated as such.
3. Now securely send this information back to the requestor of the resource.

## Update Composer's local development environment manually with your QnA Maker information

Once you securely receive the response from your Azure administrator, you will need to complete the following steps to ensure that your local development environment is setup for use in Composer.

1. Navigate to the **Configure** tab on the left column and then select the tab called **Development resources**.
2. Under **Azure QnA Maker** you will need to copy the **QnA key** your administrator provided and select the region that was provided.

This should remove the required item from the **Getting Started** experience and will also remove the errors warning that your QnA Maker configuration wasn't complete.

# Admin Handoff for Speech

5/20/2021 • 3 minutes to read

**APPLIES TO:** Composer v2.x

In the case that a developer or user would like to interact with their bot using Direct Line Speech, you need to create a Speech resource.

Many users won't have the right to create a Speech resource for their bot project as they do not have rights to an Azure Tenant to create resources. The following will walk a user through the process of generating an Admin Handoff request.

## Requesting a Speech resource creation via Admin Handoff

In Composer, if you would like to enable a bot for speech, you will be required to have created a publishing profile to define the bot environment to which you are publishing. Once you have created a publishing profile and want to enable that environment for Speech, you will need to navigate in Composer to the **Configure** menu on the left navigation menu and then select the **Connections** tab.

Overview   Development resources   **Connections**   Skill configuration   Localization

Add connections to make your bot available in Webchat, Direct Line Speech, Microsoft Teams and more. [Learn more](#).

### Azure connections

Connect your bot to Microsoft Teams and WebChat, or enable DirectLine Speech.

Select publishing profile

### External connections

Find and install more external services to your bot project in [package manager](#). For further guidance, see documentation for [adding external connections](#).

Name	Enabled	Configuration
<a href="#">Add from package manager</a>		

Once you're at the Connections tab, you will need to choose the publishing profile that you want to enable for Speech in the drop-down menu.

You will then be presented with the option to enable Speech for this publishing profile-defined environment by selecting the toggle and following the prompts to create or generate instructions for Azure administrator. Select **Generate instructions for Azure admin** and then select **Next**.

## Set up Speech

X

Use Speech to enable voice input and output for your bot.

- Use existing resources
- Create and configure new Azure resources
- Generate instructions for Azure administrator

Next

Cancel

You will then be given a screen that will allow you to copy and paste the instructions for your Azure Administrator to complete the process needed to provision your needed resources.

## Generate instructions for Azure administrator

X

If Azure resources and subscription are managed by others, use the following information to request creation of the resources that you need to build and run your bot.

Instructions



I am creating a conversational experience using Microsoft Bot Framework project. For my project to work, it needs Azure resources including Speech. Below are the steps to create these resources.

1. Using the Azure portal, please create a Speech resource on my behalf.
2. Once provisioned, securely share the resulting credentials with me as described in the link below.

Detailed instructions:  
<https://aka.ms/bfcomposerhandoffdls>

Back

Okay

## Execute the Azure Admin request for a Speech resource request from Composer

As an Azure Admin, you may be requested to create a Speech resource for use with Composer. To complete these actions, you will need to log on to the Azure Portal and create a Speech resource.

### Create Speech resource

1. In Azure, search for **Speech**.

The screenshot shows the Microsoft Azure (Preview) portal. At the top, there's a navigation bar with 'Microsoft Azure (Preview)', 'Report a bug', and a search bar. Below the navigation bar, the URL 'Home > Create a resource >' is visible. The main content area is titled 'Speech' with a Microsoft logo. It shows a rating of 3.5 (65 ratings). A blue button labeled 'Create' is at the bottom. Below the main title, there's a brief description: 'Transcribe audible speech into readable, searchable text. Add real-time speech translations to your apps and services. Convert text to audio nearly in real time. Quickly build speech-enabled apps and services using the programming languages you already work with. Customize speech systems to optimize quality for specific scenarios.' At the very bottom, there are links for 'Overview', 'Plans', 'Usage Information + Support', and 'Reviews'.

2. Select **Create**.

3. Ensure that you complete the required items. It's suggested that you keep all the resources in the same data center location. Then select the **Create** button.

Once the resource has been created you will be presented with a confirmation screen and you will need to select **Go to resource**. This will take you to the Speech resource that was created.

## Provide requestor with resource details

Navigate in the Azure Portal to the Speech resource that was created above and collect the **Resource** and **Subscription** names.

You will then need to give the user that requested the resource **Contributor** rights on the resource using the **Access control (IAM)** item from the left menu on the Azure Portal. You should also record the name of the resource that was created and the subscription name to share with the requestor.

Now send this information back to the Requestor of the Resource.

## Update Composer's publishing profile environment manually with your Speech information

Once you securely receive the response from your Azure administrator, you will need to complete the following steps to ensure that your environment and publishing profile are setup for use in Composer.

1. Navigate to the **Configure** tab on the left column and then select the tab called **Connections**.
2. Select the publishing profile that you would like to configure for **speech** in the drop-down menu.
3. Enable the Speech toggle and select the **Use existing resources** option.
4. Select your **Directory**, **Subscription**, and **Speech resource** name from the information that was provided by your Azure administrator. Select **Next**.
5. You will get a confirmation screen with the key and region for the Speech service to confirm. Select **Done**.
6. Verify that your toggle switch in Composer is now turned on and use the documentation's **Learn more** link to verify that your Speech service is properly set up and working.

# Publish a bot to Azure

6/11/2021 • 6 minutes to read

**APPLIES TO:** Composer v1.x and v2.x

This article describes how to sign in to Azure, provision Azure resources, import existing Azure resources, and publish a basic bot to *Azure Web App (Preview)*, all from within Bot Framework Composer.

## NOTE

If you choose to run scripts to provision Azure resources and publish your bot, follow the instructions in the `README` file of your bot's project folder, for example, under this directory: `C:\Users\UserName\Documents\Composer\BotName`.

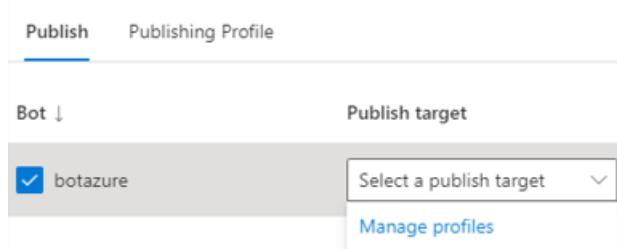
## Prerequisites

- A subscription to [Microsoft Azure](#).
- [A basic bot built using Composer v1.3](#) or later.
- [Supported OS versions](#).

## Sign in to Azure

1. In the Composer menu, select **Publish**.
2. In the **Publish your bots** pane, select the **Publish** tab.
3. Select the bot you want to publish.
4. In the **Publish target** section, select **Manage profiles** from the drop-down menu.
  - [Composer v2.x](#)
  - [Composer v1.x](#)

### Publish your bots



1. In the next display pane, select **Add new** in the **Publishing Profile**.
  - [Composer v2.x](#)
  - [Composer v1.x](#)

## Publish your bots

Publish    **Publishing Profile**

Name	Target

[Add new](#)

1. In the **Add new publishing profile** pop-up window, in the **Name** box enter the name of the profile. In the **Publishing target** box, select **Publish bot to Azure**. Then select **Next**.

- [Composer v2.x](#)
- [Composer v1.x](#)

### Add new publishing profile

A publishing profile provides the secure connectivity required to publish your bot. [Learn more](#)

Name

Publishing target

- [Composer v2.x](#)
- [Composer v1.x](#)

1. You will then see the **Sign in** into Azure. Enter the required information. After signing, you must select one of the following options:

- [Create new Azure resources](#): To *provision new* Azure resources and publish a bot.
- [Import existing Azure resources](#): To *import existing* Azure resources and publish a bot.
- [Hand off to an administrator](#): To *request your Azure admin to provision resources* on your behalf and publish a bot.

## Create new Azure resources

This section covers steps to provision Azure resources from within Composer. If you already have your Azure resources provisioned or created manually in the Azure portal, follow the instructions in [Import existing Azure resources](#) instead.

1. Once you are signed in to Azure from Composer, select **Create new resources**. Read the instructions on the right side of the screen and select **Next**.

- [Composer v2.x](#)
- [Composer v1.x](#)

Configure resources to your publishing profile X

How would you like to provision Azure resources to your publishing profile?

Create new resources      Select this option when you want to provision new Azure resources and publish a bot. A subscription to Microsoft Azure is required.

Import existing resources

Hand off to admin

STEP 1 Sign in to Azure

STEP 2 Select tenant and subscription, enter resource group name and resource name, and select regions

STEP 3 Review and create new resources. Once provisioned those resources will be available in your Azure portal.

[Learn More](#)

1. In the **Configure resources** pop-up window, provide the requested information.

## Configure resources

### Azure details

Select your Azure directory and subscription, enter resource group name.

Azure Directory * ②	<input type="text"/>
Subscription * ②	<input type="text"/> Select one
Resource group * ②	<input type="text"/> Create new
	<input type="text"/> New resource group name

### Resource details

Enter resource name and select region. This will be applied to the new resources.

Name * ②	<input type="text"/> New resource name
Region * ②	<input type="text"/> Select one
LUIS region * ②	<input type="text"/> Select one

[Learn More](#)

1. Select **Next**.

- [Composer v2.x](#)
- [Composer v1.x](#)

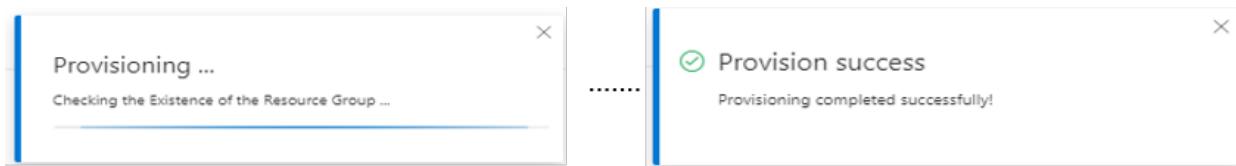
1. In the **Add resources** window, select the resources to create. Then select **Next**.

### IMPORTANT

If you are provisioning LUIS resources, make sure you select both the LUIS authoring resource and the LUIS prediction resource. If you already have a LUIS authoring or prediction resource configured locally, the provisioning process will create new ones for you.

1. In the **Review & create** window, review the resources that will be created. Select **Create**.

2. You will see messages in the upper right corner of your Composer. These messages describe the status of the provisioning process. This will take a few minutes to complete.



1. After you see the **Provision success** message on the upper right corner of the Composer screen, the provisioning process is completed. Go to your Azure portal to see the new resources created.
2. Follow the instructions covered in the [publish](#) section to deploy your bot.

## Import existing Azure resources

1. Select **Import existing resources**. Read the instructions on the right side of the screen and select **Next**.

## Add new publishing profile

A publishing profile provides the secure connectivity required to publish your bot.

- Create new resources
- Import existing resources
- Hand off to admin

### Import existing resources

Select this option to import existing Azure resources and publish a bot.

Edit the JSON file in the Publish Configuration field. You will need to find the values of associated resources in your Azure portal. A list of required and optional resources may include:

- Microsoft Application Registration
- Azure Hosting
- Microsoft Bot Channels Registration
- Azure Cosmos DB
- Application Insights
- Azure Blob Storage
- Microsoft Language Understanding (LUIS)
- Microsoft QnA Maker

[Learn More](#)

2. In the pop-up window **Importing existing resources**, modify the JSON file content based on your resources information. Then select **Import**.

#### NOTE

If you use LUIS, you should author and publish LUIS in the same region. Read more in the [Authoring and publishing regions and associated keys](#) article.

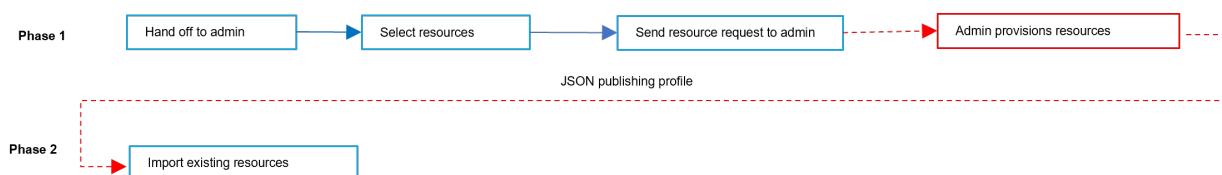
3. After the previous step is completed, follow the instructions described in the [publish](#) section below to deploy your bot to Azure.

## Hand off to an administrator

When you don't have proper permissions to use Azure or you want to generate resources from any other cloud service, you can ask an administrator to create the required resources on your behalf.

1. Select **Hand off to admin**. Read the instructions on the right side of the screen and select **Next**.
2. In the **Add resources** page, select the resources you want the administrator to provision. Select **Next**.
3. In the **Share resource request** page, copy the instructions and securely share them with your Azure admin. You will need your Azure admin to run the provisioning command and create the resources for you.
4. Your Azure admin should securely send you back a **JSON publishing profile** file. Save it.
5. Follow the steps described in [import existing resources](#) and replace the content of the related JSON file with the content you saved, sent to you by the administrator.

The figure below shows the two phases of the hand off process. During the first phase you prepare the hand off request and send it to an admin for provisioning resources in the cloud. After the resources have been provisioned, not shown in the figure, the admin sends back the related **JSON publishing profile**. During the second phase, you use the received profile and import the provisioned resources by performing the steps described in [import existing resources](#).

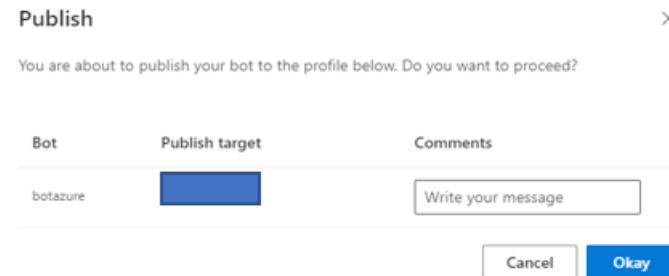


1. After the previous step is completed, follow the instructions described in the [publish](#) section below to deploy your bot to Azure.

# Publish

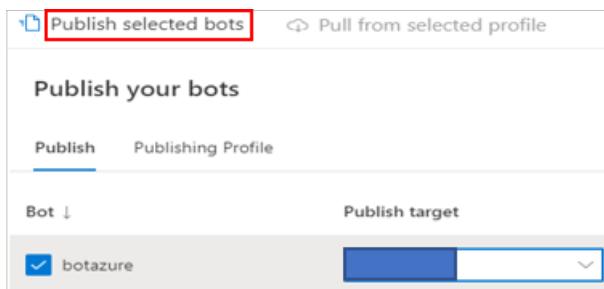
1. Select **Publish** from the Composer menu. In the **Publish your bots** pane, select the bot to publish, then select a publish profile from the **Publish target** drop-down list.

- [Composer v2.x](#)
- [Composer v1.x](#)



1. In the upper left, select **Publish selected bots**.

- [Composer v2.x](#)
- [Composer v1.x](#)



1. In the **Publish** pop-up window, select **Okay** to start the publishing process. It will usually take a few minutes to complete. After the publishing process completes, you should see a **Success** message.

- [Composer v2.x](#)
- [Composer v1.x](#)

Publish your bots				
Bot	Publish target	Date	Status	Message
<input checked="" type="checkbox"/> botazure	botazureprofile	04-25-2021	✓	Success

1. Test the bot following the steps described in the next section.

## Test your bot

1. Go to your Azure portal
2. Test the published bot by selecting **Test in Web Chat**.

- [Composer v2.x](#)
- [Composer v1.x](#)

The screenshot shows the Azure Bot service interface. On the left, there's a sidebar with options like Overview, Activity log, Access control (IAM), Tags, Settings, Bot profile, Configuration, Channels, Pricing, and Test in Web Chat (which is selected). The main area has a search bar and a 'Test' button. A message card titled 'Welcome to a simple Menu bot' is shown, followed by a message from 'menu' 3 minutes ago: 'Today's menu Pizza Lasagna Spaghetti'. Below this is another message from 'menu' 3 minutes ago: '3 minutes ago'. At the bottom is a text input field with 'Type your message' and a send button.

## Additional information

### View and access your published LUIS app

If you publish a bot with LUIS data, you can view and access the published LUIS apps after you publish the bot to Azure. LUIS apps in the LUIS portal are associated with the LUIS authoring resources you created in the Azure portal.

After publishing the bot with LUIS configuration to Azure, you will see the published LUIS resources (both LUIS authoring resource and LUIS prediction resource) from the Azure portal as follows:

- [Composer v2.x](#)
- [Composer v1.x](#)

<input type="checkbox"/>	Name ↑↓	Type ↑↓	Location ↑↓
<input type="checkbox"/>	luis-bot-02	Azure Bot	Global
<input type="checkbox"/>	luis-bot-02	App Service plan	West US
<input type="checkbox"/>	luis-bot-02	App Service	West US
<input type="checkbox"/>	luis-bot-02-luis	Cognitive Services	West US
<input type="checkbox"/>	luis-bot-02-luis-authoring	Cognitive Services	West US

#### NOTE

The LUIS authoring resource has an `luis-authoring` suffix. The LUIS authoring resource allows you to create, manage, train, test, and publish your LUIS applications. The LUIS prediction resource has a `luis` suffix. You use the prediction resource or key to query prediction endpoint requests. Read more in the [Create LUIS resources](#) article.

To view and access the LUIS apps from the LUIS portal, make sure you select the matching LUIS authoring resource as shown in your Azure portal.

- [Composer v2.x](#)
- [Composer v1.x](#)

Azure subscription \*

X ▾

Authoring resource\* ?

Select an authoring resource ...

luis-bot-02-luis-authoring westus

With the correct LUIS authoring resource selected, you will then see the associated LUIS apps from the LUIS portal.

## Conversation apps

Azure subscription: Visual Studio Enterprise / Authoring resource: luis-bot-02-luis-authoring Choose a different authoring resource.

---

+ New app	Rename	Export	Import logs	Export logs	Delete
○	Name		Last modified	Culture	Endpoint hits
<input type="checkbox"/>	ToDoBotWithLuisSample-1(composer)-ToDoBotWithLuisSample-1.en-us.lu		16/3/2021	en-us	0
<input type="checkbox"/>	ToDoBotWithLuisSample-1(composer)-deleteitem.en-us.lu		16/3/2021	en-us	0
<input type="checkbox"/>	ToDoBotWithLuisSample-1(composer)-userprofile.en-us.lu		16/3/2021	en-us	0
<input type="checkbox"/>	ToDoBotWithLuisSample-1(composer)-additem.en-us.lu		16/3/2021	en-us	0
<input type="checkbox"/>	ToDoBotWithLuisSample-1(composer)-viewitem.en-us.lu		16/3/2021	en-us	0

# Publish a skill to Azure

6/11/2021 • 8 minutes to read

**APPLIES TO:** Composer v1.x and v2.x

Publishing your bot as a skill allows other published bots to access it as a skill. This article describes how to publish a skill as an *Azure Web App*.

Bots must have a skill manifest to be used as a skill.

## Prerequisites

- A subscription to [Microsoft Azure](#).
- A bot built using Composer v1.4 or later.
- A [supported OS version](#) for the current release of Composer.

## Create a local skill or start with an existing bot

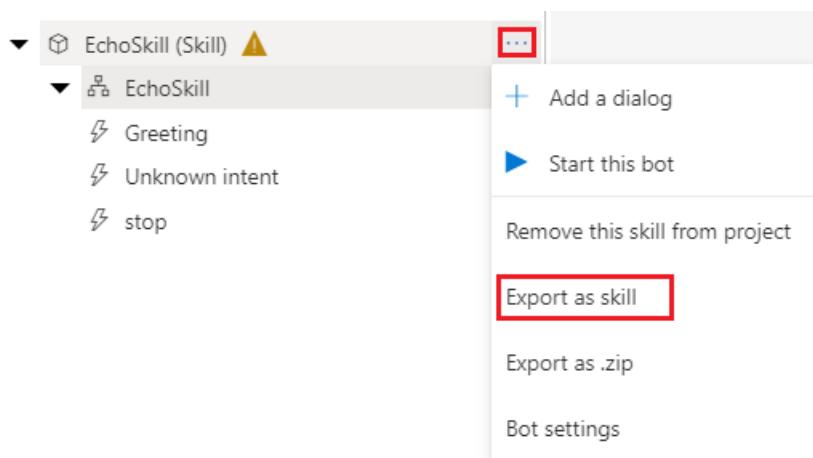
If you don't have a local skill to publish, use the instructions in how to [Create and test a local skill](#) to create one.

You can use the same steps to publish any existing bot as a skill.

## Generate a skill manifest and publish your skill

- [Composer v2.x](#)
- [Composer v1.x](#)

1. Go to the **Create** view. On the **Navigation** pane, select **more options (...)** to the right of your skill bot. Select **Export as a skill** from the menu.



2. In the **Export your skill** dialog, describe your skill. Composer will add this information to your skill's manifest. Then, select **Next**.

These are the properties Composer uses to generate your skill manifest.

CATEGORY	TYPE	REQUIRED	DESCRIPTION
----------	------	----------	-------------

CATEGORY	TYPE	REQUIRED	DESCRIPTION
Manifest Version	String	Required	The skill manifest schema version to use.
Name	String	Required	The name of the skill.
Version	String	Required	The version of the skill the manifest describes.
Publisher Name	String	Required	The name of the skill publisher.
Description	String	Optional	A human-readable description of the skill.
Privacy Url	String	Optional	The URI of the privacy description for the skill.
Copyright	String	Optional	The copyright notice for the skill.
License	String	Optional	The license agreement for the skill.
Icon Url	String	Optional	The URL of the icon to show for the skill.
Tags	String	Optional	A set of tags for the skill. If present, each tag must be unique.

**TIP**

For the complete tables describing the full schema for v2.0 and v2.1 of the Bot Framework skill manifest, read the [How to write a v2.0 skill manifest](#) and [how to write a v2.1 skill manifest](#) articles from the [Bot Framework SDK documentation](#).

3. In the **Select dialogs** dialog, select the dialogs that will be accessible to consumer bots.

- Each dialog name is used as the ID of the associated action in the manifest. If your skill contains multiple dialogs, you can choose which ones a skill consumer can use to start a task.
- The **Description** will be the description associated with the skill.
- Select **Next**.

4. In the **Select triggers** dialog, select the triggers to include that can start a task.

- Select the **Greeting** trigger, as you are expecting a `conversationUpdate` activity as the initial activity sent by a root bot to the skill.
- Select **Next**.

5. In the **Which bots can connect to this skill?** dialog, select **Add allowed callers**.

- For this skill, enter an asterisk (\*) to allow any bot to call this bot as a skill. To restrict callers to specific bots, you would instead add the app ID of each allowed caller.
- Select **Next**.

6. Your bot needs to be provisioned and should have a publishing profile created in order to make it a skill.

If you do not have a publishing profile created you will see a **Create a publish profile to continue** dialog (You will not see this dialog if you have already have one or more publishing profiles created for your bot and you can skip to the next step to **confirm skill endpoints**). Click on Create a new publish profile and follow the instructions in the [Publish a bot](#) article to publish your skill to Azure, using the following options:

- Publish to Azure.
- Create new resources.
- No optional resources are required for the echo skill.

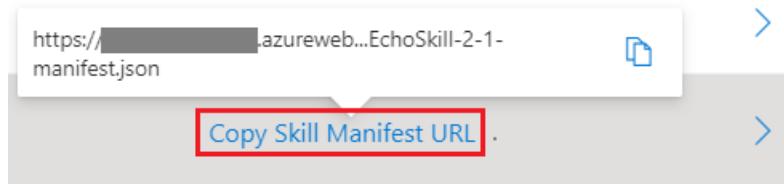
At this point, your skill is provisioned but not yet published.

1. Next, In the **Confirm skill endpoints** dialog, select the publishing profile you created earlier. Then, select **Generate and Publish**.

2. In the **Publish** dialog, enter an optional publishing comment, and select **Okay**.

3. Composer publishes your skill to Azure.

- This process can take a minute or two.
- Once complete, copy the manifest URL. You will use the manifest URL to connect to the deployed skill.



Now your remote skill is ready!

- You can test your skill as a bot directly in Azure. To do so, use Azure's **Test in Web Chat** feature.
- Read the [how to connect to a skill](#) article and learn how to connect a bot to this remote skill.

## Further reading

- Learn [how to create a skill](#).
- Learn [how to connect to a remote skill](#).

# Create a CI/CD pipeline to publish your bot (Preview)

5/20/2021 • 4 minutes to read

APPLIES TO: Composer v2.x

Composer developers can create a CI/CD pipeline to easily deploy new versions of their bots. Using Composer, Azure DevOps, and git, developers can seamlessly deliver their software updates. Note that this process assumes that you have already provisioned the resources in the [prerequisites].

## Prerequisites

To create a CI/CD pipeline, you will need:

- An [Azure DevOps project](#) where you will configure the Azure pipeline
- An [Azure subscription](#) where you will be deploying your bot
- A git repository with the source for the bot you are trying to deploy
- An [Azure resource group](#) with the following resources provisioned and configured:
  - An [Azure App Service plan](#) for your web apps
  - An [Azure App Service](#) to deploy your bot
  - An Azure Bot
  - (Optional) A LUIS Authoring and Perdition resource
  - (Optional) A QnA Maker resource

The sample YAML files referenced in this article can be found on [GitHub](#).

FILE	DESCRIPTION
<a href="#">buildAndDeploy.yaml</a>	The main YAML file used when the Azure DevOps pipeline is triggered. It maps Azure Dev Ops variables (see <a href="#">Define variables</a> for additional information) into YAML <a href="#">runtime parameters</a> and then sequentially calls the YAML templates in the templates folder to build and deploy the bot.
<a href="#">templates/installPrerequisites.yaml</a>	Installs the tools needed to run the pipeline, like npm, BF CLI, and .NET core)
<a href="#">templates/buildAndDeployModels.yaml</a>	Builds, trains and deploys LUIS and QnA models. This template also creates the [projectDirectory]/generated folder that is needed by the bot.
<a href="#">templates/buildAndDeployDotNetWebApp.yaml</a>	Builds the dotnet bot app, prepares the zip package, and deploys it to Azure. It also configures the app settings for the app in Azure once it is deployed.

## Configuration steps

### Provision your Azure environment

You first need to provision Azure resources to publish your bot. Take note of the configuration settings so you can later configure the CI/CD pipeline. To make this easier, copy and paste the [Pipeline parameters table](#)

provided in a separate document and use it as a template.

## Add YAML files to your bot project source code

Download the YAML files provided and add them to your bot's source code in git under the build folder. Then merge the changes into your main branch.

File	Commit Message	Time Ago
BasicAssistant	First pass at migration (#1)	23 hours ago
build/yaml	First pass at migration (#1)	23 hours ago
.editorconfig	First pass at migration (#1)	23 hours ago
.gitignore	First pass at migration (#1)	23 hours ago
LICENSE	Initial commit	yesterday
README.md	First pass at migration (#1)	23 hours ago

### TIP

This is a good time to update the `BotProjectDirectory` and `BotProjectName` settings in your Pipeline parameters table with your `.csproj` name and the relative path to its location in your source tree.

## Create and configure your CI/CD pipeline

### Configure an Azure Service Connection for your Azure DevOps project

#### IMPORTANT

Ensure that your user has 'Owner' or 'User Access Administrator' permissions on the Azure Subscription before you try to create the service connection.

Your pipeline uses a service connection to interact with the resource group where the Azure resources for your bot are located. Create a service connection and note the name in the [Pipeline parameters table](#) under the `AzureServiceConnection` setting.

To configure a Service Connection, go to the [Project Settings](#) section in Azure DevOps, and select [Service connections](#), as seen below:



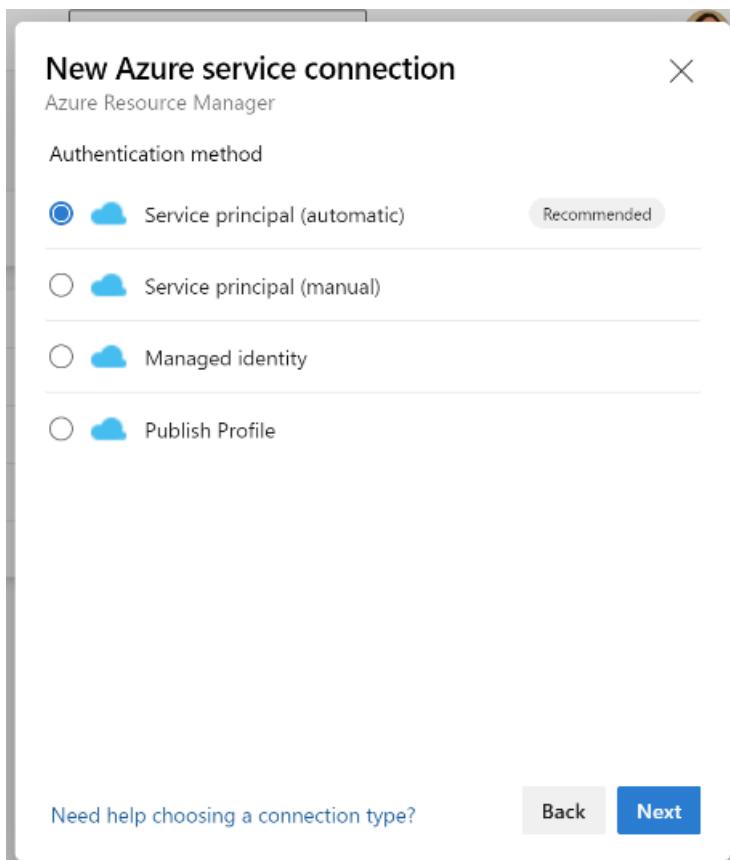
The screenshot shows the 'Project Settings' menu in the Azure DevOps interface. On the left is a vertical sidebar with icons for various settings: General (selected), Parallel jobs, Settings, Test management, Release retention, Service connections (highlighted in blue), XAML build services, Repos, Artifacts, Storage, and Test. The main area displays the selected 'Service connections' section.

Then select the New Service Connection button and create an Azure Resource Manager resource:

The screenshot shows the 'New service connection' dialog box. It has a search bar at the top labeled 'Choose a service or connection type' and 'Search connection types'. Below the search bar is a list of connection types with radio buttons:

- Azure Classic
- Azure DevOps Service Connection
- Azure Repos/Team Foundation Server
- Azure Resource Manager
- Azure Service Bus
- Bitbucket Cloud

Select **Service principal (automatic)** and grant it permissions to the Azure resource group where you created your bot resources.



See [Service connections](#) in the Azure DevOps documentation for additional information.

#### Create your pipeline

In your Azure DevOps project pipelines section select new pipeline.

Select your source code provider and repository in the first two tabs and in the Configure steps above. Scroll down to the bottom of the page and select **Existing Azure Pipelines YAML file**:

- Starter pipeline**  
Start with a minimal pipeline that you can customize to build and deploy your code.
- Existing Azure Pipelines YAML file**  
Select an Azure Pipelines YAML file in any branch of the repository.

[Show more](#)

Then select the `buildAndDeploy.yaml` file you previously added to the repository:

## Select an existing YAML file



Select an Azure Pipelines YAML file in any branch of the repository.

Branch

main



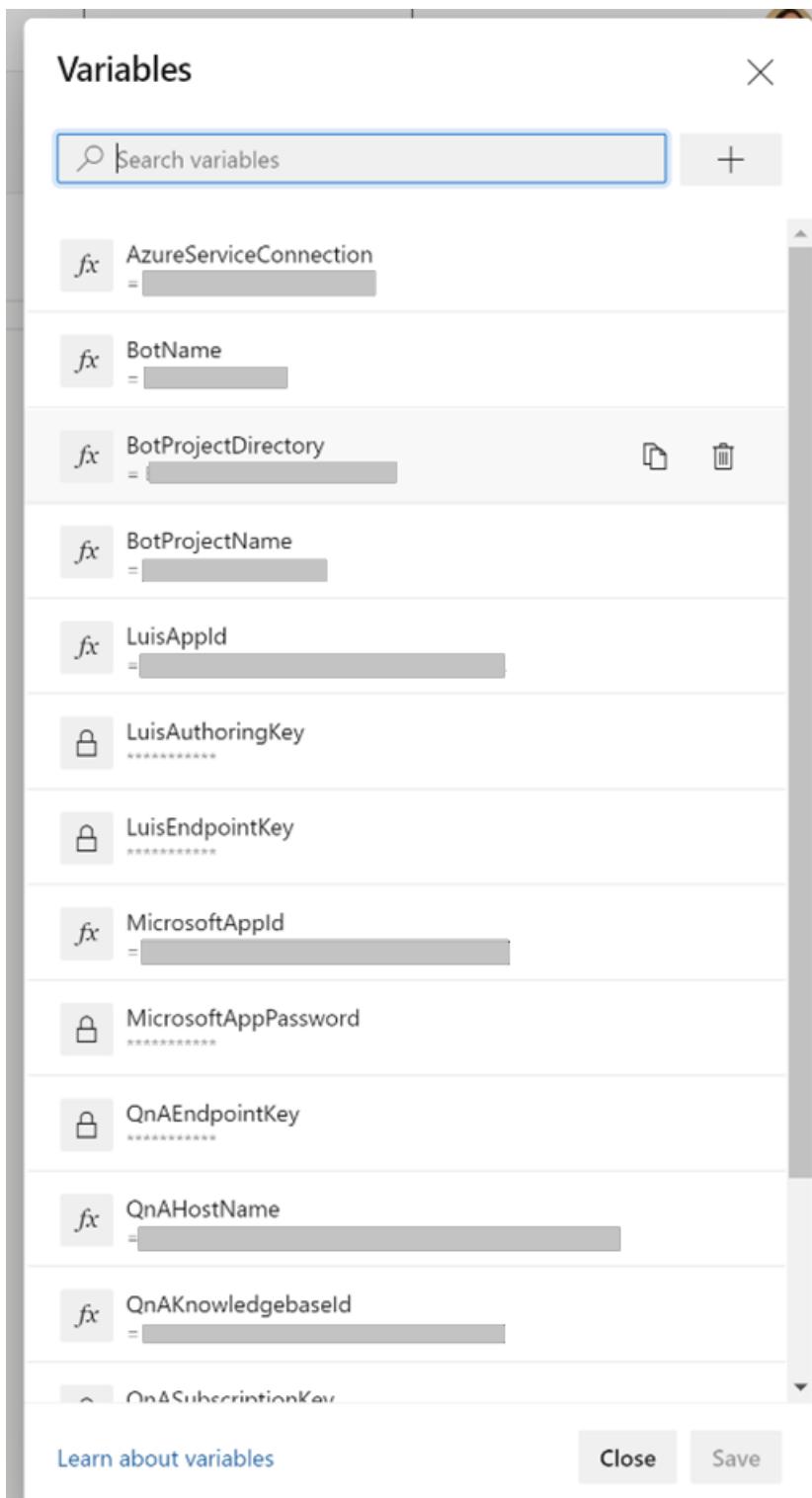
Path

/build/yaml/buildAndDeploy.yaml



Select a file from the dropdown or type in the path to your file

In the **Review** tab, select the **Variables** button and create the variables with the values defined in the [Pipeline parameters table](#). Make sure you check the **Keep this value secret** option for sensitive parameters.



Then select **Run**. This saves and executes your pipeline for the first time, seen below:

The screenshot shows the Azure DevOps pipeline interface. On the left, there's a tree view under 'Jobs' with a node 'Job' expanded, showing various steps like 'Initialize job', 'Checkout gabog/ComposerCI...', 'Nuget Security Analysis', etc., each with its duration. On the right, a detailed log window titled 'Job' displays the command history:

```

1 Pool: Azure Pipelines
2 Image: ubuntu-latest
3 Agent: Hosted Agent
4 Started: Yesterday at 11:11 AM
5 Duration: 3m 4s
6
7 ▶ Job preparation parameters
8 ▶ fx Parent pipeline used these runtime parameters
9 └─ 1 artifact produced

```

#### Configure your LUIS prediction resources (first time only)

Depending on your bot, the pipeline will create one or more LUIS apps when you run it for the first time. Once that is done, make sure you set the prediction resource in the **Manage section** for your LUIS applications:

The screenshot shows the Azure portal interface for managing LUIS applications. At the top, there's a navigation bar with 'DASHBOARD', 'BUILD', and 'MANAGE' tabs, where 'MANAGE' is selected. Below that, it says 'Your survey!'. The main content area is titled 'Azure Resources' and contains a sub-section 'Prediction Resources'. It includes a link 'Add prediction resource'.

#### Run and test your deployed bot

You can now run and test your bot from the configured channels.

#### Appendix: Pipeline parameters

CONFIGURATION SETTING	DESCRIPTION	EXAMPLE
AzureServiceConnection	The name of the Azure DevOps service connection used by the pipeline to interact with the Azure resources	MyCICDBotDeployment
BotProjectDirectory	The relative path to your bot's source code	BasicAssistant/BasicAssistant
BotProjectName	The name of the <code>.csproj</code> for your bot code	BasicAssistant.csproj
ResourceGroupName	The Azure resource group for your deployment resources	MyTestBot-RG
WebAppName	The App Service name for your bot	MyCICDBotApp

CONFIGURATION SETTING	DESCRIPTION	EXAMPLE
BotName	Your bot's name	MyCICDBot
MicrosoftAppId	The App ID for your bot	11111111-1111-1111-1111-111111111111
MicrosoftAppPassword	The App password for your bot	YourPasswordHere
LuisAppId	The LUIS AppId for your root dialog app	11111111-1111-1111-1111-111111111111
LuisAuthoringKey	The authoring key for your LUIS authoring resource (from the Azure portal)	1234567890abcdefgijk1234567890a
LuisEndpointKey	The endpoint key for your LUIS prediction resource (from the Azure portal)	1234567890abcdefgijk1234567890a
QnASubscriptionKey	The QnA Maker subscription key (from the Azure Portal)	1234567890abcdefgijk1234567890a
QnAHostName	Your QnA maker host URL. You can obtain this value from your QnA KB Deployment details section in the Settings page.	<a href="https://mycicdbotapp-qna-qnahost.azurewebsites.net/qnamaker">https://mycicdbotapp-qna-qnahost.azurewebsites.net/qnamaker</a>
QnAEndpointKey	The endpoint key used to access QnA. You can obtain this value from your QnA KB Deployment details section in the Settings page.	1234567890abcdefgijk1234567890a
QnAKnowledgebaseId	Your QnAMaker KB ID. You can obtain this value from your QnA KB Deployment details section in the settings page	11111111-1111-1111-1111-111111111111

Here is an example of how the pipeline variables would look in Azure DevOps:

The screenshot shows the 'Variables' tab in the Azure DevOps pipeline editor. At the top, there are navigation links: 'YAML', 'Variables' (which is selected), 'Triggers', 'History', 'Save & queue', 'Discard', 'Summary', 'Queue', and '...'. Below this is a table titled 'Pipeline variables'.

**Pipeline variables**

Name ↑	Value	Settable at queue time
AzureServiceConnection	MyCICDBotDeployment	
BotName	MyCICDBot	
BotProjectDirectory	BasicAssistant/BasicAssistant	
BotProjectName	BasicAssistant.csproj	
LuisAppId	11111111-1111-1111-1111-111111111111	
LuisAuthoringKey	*****	
LuisEndpointKey	*****	
MicrosoftAppId	11111111-1111-1111-111111111111	
MicrosoftAppPassword	*****	
QnAEndpointKey	*****	
QnAHostName	https://mycicdbotapp-qna-qnahost.azurewebsites.net/qnamaker	
QnAKnowledgebaseId	11111111-1111-1111-111111111111	
QnASubscriptionKey	*****	
ResourceGroupName	MyTestBot-RG	
system.collectionId	ae8b0170-8c49-492d-afe5-8666d1955c7e	
system.definitionId	1325	
system.teamProject	GaboSandbox	
WebAppName	MyCICDBotApp	

[+ Add](#)

## Additional information

### Submitting Feedback/Issues

- Go to the [BotFramework-Composer repository](#) issues.
- Select the appropriate button: **Bug**, **Feature Request**, or **Other**.
- Fill out template and submit your issues. Add the **Area: CICD** label.

# Glossary

5/28/2021 • 10 minutes to read

[A](#) | [B](#) | [C](#) | [D](#) | [E](#) | [F](#) | [G](#) | [H](#) | [I](#) | [J](#) | [K](#) | [L](#) | [M](#) |

[N](#) | [O](#) | [P](#) | [Q](#) | [R](#) | [S](#) | [T](#) | [U](#) | [V](#) | [W](#) | [X](#) | [Y](#) | [Z](#)

## A

### Action

*Actions* are the main component of a [trigger](#), they are what enable your bot to take action whether in response to user input or any other event that may occur. Actions are very powerful, with them you can formulate and send a response, create properties and assign them values, manipulate the conversational flow and dialog management and many other activities.

Additional Information:

- See [action](#) in the dialog concept article.

### Activity

An action expressed by a bot, a channel, or a client that conforms to the Activity Schema.

Additional Information:

- Read more [here](#).

### Adaptive dialogs

*Adaptive Dialogs* are a new way to model conversations that take the best of waterfall dialogs and prompts in the dialogs library. *Adaptive Dialogs* are event-based. Using adaptive dialogs simplifies sophisticated conversation modelling primitives like building a dialog dispatcher and ability to handle interruptions elegantly. *Adaptive dialogs* derive from dialogs and interact with the rest of the Bot Framework SDK dialog system.

Additional Information:

- See [adaptive dialogs](#).

### Adaptive expressions

Adaptive expressions are a new expressions language used with the Bot Framework SDK and other conversational AI components, like [Bot Framework Composer](#), [Language Generation](#), [Adaptive dialogs](#), and [Adaptive Cards](#).

Additional Information:

- See [adaptive expressions](#)

### Authoring canvas

A section of the [Design](#) page where users design and author their bot.

## B

### Bot Responses

An option in the Composer Menu. It navigates users to the [Bot Responses](#) page where the [Language Generation \(LG\) editor](#) locates. From there users can view all the LG templates and edit them.

## **Bot project**

A bot project is a top-level container of multiple bots. A bot project must contain a root bot, and it may contain one or more local and/or remote skills.

# C

## **Child dialog**

Every dialog that you create in Composer will be a *child dialog*. Dialogs can be nested multiple levels deep with the [main dialog](#) being the root of all dialogs in Composer. Each child dialog must have a [parent dialog](#) and parent dialogs can have zero or more child dialogs, but a child dialog can and must have only one parent dialog.

Additional Information:

- The [dialog](#) concept article.
- Learn to create a child dialog in the [Tutorial: Adding dialogs to your bot](#)

# D

## **Design**

An option in the Composer Menu. It navigates users to the [Design](#) page where users design and develop their bots.

## **Dialog**

Dialogs are the basic building blocks in Composer. Each dialog represents a portion of the bot's functionality that contains instructions for what the bot will do and how it will react to user input. Dialogs are composed of [Recognizers](#) that help understand and extract meaningful pieces of information from user's input, a [language generator](#) that helps generate responses to the user, [triggers](#) that enable your bot to catch and respond to events and [actions](#) that help you put together the flow of conversation that will occur when a specific event is captured via a Trigger. There are two types of dialogs in Composer: [main dialog](#) and [child dialog](#).

Additional Information:

- The [dialog](#) concept article.

# E

## **Emulator**

The *Bot Framework Emulator* is a desktop application that allows bot developers to test and debug their bots, either locally or remotely. Using the Emulator, you can chat with your bot and inspect the messages it sends and receives. The Emulator displays messages as they would appear in a web chat UI and logs JSON requests and responses as you exchange messages with your bot. Before deploying your bot, you can run it locally and test it using the Emulator.

Additional Information:

- [The latest release of the Bot Framework Emulator](#)

## **Endpoint**

In Bot Framework, an endpoint is a programmatically addressable location where a bot or channel can receive activities.

## **Entity**

An *entity* contains the important details of the user's [intent](#). It can be anything, a location, date, time, cuisine type, etc. An intent may have no entities, or it may have multiple entities, each providing additional details to help understand the needs of the user.

Additional Information:

- See [entities](#) in the Language Understanding concepts article.

## Examples

A section in the Composer [Home](#) page listing all the example bots.

Additional Information:

- Read more about [how to use samples](#).

## Export

The "export" activity will generate a copy of your bot's runtime so that it can be used for other purposes such as debugging in Visual Studio.

F

G

H

## Home

An option in **Menu** and the start page of Composer.

I

## Intent

An *intent* is the task that the user wants to accomplish or the problem they want to solve. Intent recognition in Composer is its ability to determine what the user is requesting. This is accomplished by the [recognizer](#) using either [Regular Expressions](#) or [LUIS](#). When an intent is detected from the user's input, an event is emitted which can be handled using the [Intent recognized](#) trigger. If the intent is not recognized by any recognizers, another event is emitted which can be handled using the [Unknown intent](#) trigger.

Additional Information:

- See [intents](#) in the Language Understanding concepts article.

J

K

L

## Language Generation

Language Generation (LG), is the process to produce meaningful phrases and sentences in the form of natural language. Language generation enables your bot to response to a user with human readable language.

Additional Information:

- The [Language Generation](#) concept article.

## LG editor

Known as the [response editor](#), this is a section of the [Bot Responses](#) page. It is the editor where users can view and edit all the Language generation templates.

There is also an inline response editor in the [Authoring Canvas](#) where your [Bot Responses](#) for the selected trigger or action can be added or updated.

## Language Understanding

Language Understanding (LU) deals with how the bot handles users input and converts them into something that it can understand and respond to intelligently. It involves the use of either a [LUIS](#) or [Regular Expression recognizer](#) along with [utterances](#), [intents](#) and [entities](#).

Additional Information:

- The [Language Understanding](#) concept article.

## Local skill

A local skill is a [skill](#) that does not need to contain a [skill manifest](#). It can be called and shared within the Composer environment without Azure App registration resources.

## LU editor

A section of the [User Input](#) page. It is the language understanding editor where users can view and edit all the Language understanding templates.

## LUIS

A [recognizer](#) type in Composer that enables you to extract [intent](#) and [entities](#) based on [LUIS](#) service.

Additional Information:

- See [how to use LUIS for language understanding](#) in Composer.

# M

## Main dialog

The *main dialog* is the foundation of every bot created in Composer. There is only one main dialog and all other dialogs are children of it. It gets initialized every time your bot runs and is the entry point into the bot.

## Memory

A bot uses memory to store property values, in the same way that programming and scripting languages such as C# and JavaScript do. A bots memory management is contained within the following scopes: user, conversation, dialog and turn.

Additional Information:

- See the [conversation flow and memory](#) concept article.

## Menu

A list of options provided on the left side of the Composer screen from which a user can choose.

# N

## Navigation pane

A section of the Composer screen. It enables users to navigate to different parts of Composer.

## Notifications

An option in the Composer [Menu](#). It navigates users to the [Notifications](#) page that lists all the errors and warnings of the current bot application.

Additional Information:

- See the [validation](#) article.

# O

# P

## **Parent dialog**

A *parent dialog* is any dialog that has one or more [child dialogs](#), and any dialog can have zero or more child dialogs associated with it. A parent dialog can also be a child of another dialog.

## **Prompt**

*Prompts* refer to bots asking questions to users to collect information of a variety of data types (e.g. text, numbers).

Additional information:

- Read more about [prompts](#).

## **Property**

A *property* is a distinct value identified by a specific address. An address is comprised of two parts, the scope and name: *scope.name*. Some examples of typical properties in Composer could include: user.name, turn.activity, dialog.index, user.profile.age.

Additional information:

- Read more about property in the [memory](#) concept article.

## **Properties pane**

A section of the [Design](#) page where users can edit properties.

# Q

## **QnA maker**

A cloud-based Natural Language Processing (NLP) service that easily creates a natural conversational layer over your data. Bot Framework Composer integrates [QnA Maker knowledge base creation and management](#) in addition to the existing LUIS integration for language understanding.

Additional information:

- See the [What is the QnA Maker service](#) article.

## **QnA knowledge base**

QnA Maker imports [your content](#) into a knowledge base of question and answer pairs. After you publish your knowledge base, a client application such as a chat bot can send a user's question to your endpoint. Your QnA Maker service processes the question and responds with the best answer. Users can create and manage their QnA Maker knowledge bases within the context of the Composer environment.

Additional information:

- See [how to create a QnA Maker knowledge base in Composer](#)).

# R

## **Recognizer**

A *recognizer* enables your bot to understand and extract meaningful pieces of information from user's input. There are currently two types of recognizers in Composer: [LUIS](#) and [Regular Expression](#), both emit events which are handled by [triggers](#trigger).

## **Regular Expression**

A *Regular Expression* (regex) is a sequence of characters that define a search pattern. Regex provides a powerful, flexible, and efficient method for processing text. The extensive pattern-matching notation of regex enables your bot to quickly parse large amounts of text to find specific character patterns that can be used to determine user intents, validate text to ensure that it matches a predefined pattern (such as an email address or zip codes), or extract entities from utterances.

## **Remote skill**

A remote skill is a [skill](#) that contains a [skill manifest](#) and is published to a remote host, such as Azure Web App.

## **Response editor**

This is where you can add or make updates to your LG templates.

See also:

- [Language Generation](#)
- [Bot Responses](#)
- [LG editor](#)

## **Root bot**

A root bot is the first and the main bot that is created in your bot's project.

## **Root dialog**

See [main dialog](#).

# S

## **Scope**

When a [property](#) is in *scope*, it is visible to your bot. See [memory](#) concept article to know more about the different scopes of memory.

## **Settings**

An option in the Composer [Menu](#). It navigates users to the [Settings](#) page where users manage settings for their bot and Composer.

## **Skill**

A skill is a bot that can perform a set of tasks for another bot.

- See the [Skills overview](#) article.

## **Skill consumer**

A skill consumer is a bot that can invoke one or more skills.

Additional information:

- See the [Skills overview](#) article.

## **Skill manifest**

A skills manifest is a JSON file that describes the actions the skill can perform, its input and output parameters, and the skill's endpoints.

- See the [Skills overview](#) article.

# T

## **Title bar**

A horizontal bar at the top of the Composer screen, bearing the name of the product and the name of current bot project.

## Toolbar

A horizontal bar under Title bar in the Composer screen. It is a strip of icons used to perform certain actions to manipulate [dialogs](#), [triggers](#), and [actions](#).

## Trigger

*Triggers* are the main component of a dialog, they are how you catch and respond to events. Each trigger has a condition and a collection of [actions](#) to execute when the condition is met.

Additional information:

- See [events and triggers](#) concept article.
- See [how to define triggers](#) article.

## Trigger phrase

In Composer, trigger phrases are example utterances that users define with a LUIS recognizer or a regular expression recognizer. Composer's language processing examines a user's utterance to determine the intent and extract any entities it may contain. A trigger phrase in LUIS is generally referred to as an utterance. A trigger phrase in a regular expression is generally referred to as a pattern.

## U

### User Input

An option in the Composer [Menu](#). It navigates users to the [User Input](#) page where the [Language Understanding editor](#) locates. From there users can view all the Language Understanding templates and edit them.

### Utterance

An *utterance* can be thought of as a continuous fragment of speech that begins and ends with a clear pause. Composer's language processing examines a user's utterance to determine the [intent](#) and extract any [entities](#) it may contain.

Additional Information:

- See [utterances](#) in the Language Understanding concepts article.

## V

## W

## X

## Y

## Z

# Composer FAQ

5/12/2021 • 2 minutes to read

## How do I find my remote skill manifest URL?

A *skill manifest* is a JSON file that describes the actions the skill can perform. A remote skill will expose a skill manifest URL so that the skill can be shared outside the Composer environment. To find the skill manifest URL of your remote skill, follow these steps:

### NOTE

The following instructions apply to a remote skill in Azure. Make sure you have already [published your skill to Azure](#).

1. Navigate to your Azure portal and select the **App Service** resource associated with the published skill.

Name ↑↓	Type ↑↓	Location ↑↓	
<input checked="" type="checkbox"/>  [REDACTED]-composer-skill	Bot Channels Registration	Global	...
<input type="checkbox"/>  [REDACTED]-composer-skill	App Service plan	West US	...
<input type="checkbox"/>  [REDACTED] composer-skill	App Service	West US	...

2. On the App service menu on the left, select **App Service Editor (Preview)** from the **Development Tools** section.

### Development Tools

-  Clone App
-  Console
-  Advanced Tools
-  App Service Editor (Preview)
-  Extensions

### NOTE

The App Service Editor (Preview) is a tool of Microsoft Azure. It provides a web-based editor to Azure Web App.

3. From the **App Service Editor (Preview)** page, select **Go ->**



App Service Editor provides an in-browser editing experience for your App code. [Learn more](#)

[Go →](#)

4. From the App Service Editor's EXPLORE pane, select the file name of your skill's manifest in the manifest folder.

```

1 {
2   "$schema": "https://schemas.botframework.com/schemas/skills/v2.1/skill-manifest.json",
3   "id": "test0114-echoskill-remote-36a6ef8d-30a5-4dca-b065-2a324d5d5ae5",
4   "endpoints": [
5     {
6       "protocol": "BotFrameworkV3",
7       "name": "echo-skill",
8       "endpointUrl": "https://[REDACTED]/api/messages",
9       "msAppId": "[REDACTED]"
10    }
11  ],
12  "name": "test0114-echoskill-remote",
13  "version": "1.0",
14  "publisherName": "Microsoft",
15  "activities": [
16    "test0114-echoskill-remote": {
17      "type": "event",
18      "name": "test0114-echoskill-remote"
19    },
20    "conversationUpdate": {
21      "type": "conversationUpdate"
22    }
23  }
24 }
25

```

This will open the remote skill manifest with a URL like this:

<https://composer-skill.scm.azurewebsites.net/dev/wwwroot/ComposerDialogs/manifests/echo-remote-Skill-2-1-preview-1-manifest.json>

. This is NOT your skill manifest URL.

You should update the URL to the following format:

<https://composer-skill.azurewebsites.net/manifests/echo-remote-Skill-2-1-preview-1-manifest.json> . This is the skill manifest URL of your published skill.

## How do I publish LUIS to different regions?

Composer currently supports publishing LUIS to the following three regions: `westus`, `westeurope`, and `australiaeast` . Make sure the **Luis region** you selected in Composer is consistent with the **Location** of your LUIS authoring resource in the Azure portal.

### TIP

Read more about [LUIS Authoring regions](#) in the LUIS documentation.

The following is an example **Location** of your LUIS authoring resource in the Azure portal.

[Hide Keys](#)

KEY 1

[REDACTED]

KEY 2

[REDACTED]

Endpoint

<https://luis-westeurope-authoring.cognitiveservices.azure.com/>

Location

**westeurope**

The following is an example **Luis region** of your bot project's setting in Composer.

LUIS application name ?

ToDoBotWithLuisSample-0

LUIS authoring key \* ?

LUIS endpoint key ?

Enter LUIS endpoint key

LUIS region \* ?

westeurope



# Overview: Extend Power Virtual Agents bots with Composer

6/18/2021 • 2 minutes to read

**APPLIES TO:** Composer v1.x and v2.x

Extending your Power Virtual Agents bot with Composer makes it easier than ever for developers and business users to build bots together. You can now add complexity to Power Virtual Agents bots using Bot Framework's rich dialog functionality, and access conversational memory and context defined in Power Virtual Agents. Additionally, Composer developers now have access to the no-code graphical interface and NLU models available in Power Virtual Agents.

The purpose of this article is to introduce Composer users to Power Virtual Agents and link to relevant documentation about extending Power Virtual Agents with Composer.

## Prerequisites

- Familiarity with [Power Virtual Agents](#).
- A Power Virtual Agents Production license or [full trial](#). Composer integration is not available to users who only have the [Teams Power Virtual Agents license](#).
- A basic understanding of [dialogs, triggers, conversational flow and memory](#), and [language generation](#).
- Composer [installed as a desktop application](#). You won't be able to extend Power Virtual Agents bots if Composer is [built from source](#).

## Learn about Power Virtual Agents

There are similarities between Power Virtual Agents and Composer, like that both provide a no-code authoring canvas for users to build bots. However, there are key differences in the functionality and look of both technologies. The following documentation is recommended for Composer users to better understand Power Virtual Agents. Read these articles to learn how to:

- [Create](#) a Power Virtual Agents bot.
- Create and edit [topics](#). Topics are similar to dialogs in Composer. Here's a table of other terminology differences between Composer and Power Virtual Agents. Note that these associations are not an exact match:

POWER VIRTUAL AGENTS	COMPOSER
topic	dialog
node	action
trigger phrase	Language Understanding
entity	LU editor

- [Test](#) your Power Virtual Agents bot.

- [Enhance and improve](#) your Power Virtual Agents bot.

## Extend Power Virtual Agents with Composer

Read the following sections in the [Extend you bot with Bot Framework Composer](#) article to learn how to:

- [Access Composer from Power Virtual Agents](#). You should always open Composer from within Power Virtual Agents if you plan to create content for Power Virtual Agents bots. This will ensure that Composer has all the necessary plugins needed for integration with Power Virtual Agents.
- Follow [guidelines for creating bot content in Power Virtual Agents](#).
- Show an [Adaptive card](#) in Power Virtual Agents.
- [Use Power Virtual Agents variables in Composer](#).
- Display [multiple choice options](#) in Power Virtual Agents.
- Use [Bing Search as a fallback](#) in Power Virtual Agents.
- [Test Bot Framework content with Power Virtual Agents](#). Note that Power Virtual Agent bots extended with Composer cannot be tested in the Bot Framework Emulator.

### IMPORTANT

Not all features are supported when you open Composer from Power Virtual Agents. See [Composer features not supported with Power Virtual Agents](#) for more information.

## Additional Information

See the [Power Virtual Agents documentation](#) for more information.

# Memory scopes and properties

6/11/2021 • 6 minutes to read

**APPLIES TO:** Composer v1.x and v2.x

This article lists the properties that Composer sets by default, grouped by *memory scope*.

See [Conversation flow and memory](#) to find out how memory is used in Bot Framework Composer.

Memory properties are managed within the following scopes:

## The scopes

- [Composer v2.x](#)
- [Composer v1.x](#)

SCOPE	DESCRIPTION
Settings	Read-only information from the bot configuration file.
User	Properties associated with the current user. Properties in the user scope don't expire. These properties are in scope while the bot is processing an activity associated with the user.
Conversation	Properties associated with the current conversation. Properties in the conversation scope have a lifetime of the conversation itself. These properties are in scope while the bot is processing an activity associated with the conversation.
Dialog	Properties associated with the active dialog. Properties in the dialog scope are kept until the dialog ends.
This	Properties associated with the current action.
Turn	Properties associated with the current turn.
Dialog context	Properties associated with the dialog stack.
Dialog class	Information about the active dialog.
Class	Information about the current action.

### TIP

- A *turn* consists of the user's incoming activity to the bot and any activities the bot sends back to the user as an immediate response. You can think of a turn as the processing associated with the bot receiving a given activity.
- Both user and conversation state are scoped by channel. The same person using different channels to access your bot appears as different users, one for each channel, and each with a distinct user state.

The next sections describe the properties that Composer automatically makes available to you. You can also define and access your own properties. See [Conversation flow and memory](#) for more about how memory scopes and properties are used in bot.

## Dialog scope

- [Composer v2.x](#)
- [Composer v1.x](#)

VARIABLE	DESCRIPTION
dialog.eventCounter	Counter of emitted events.
dialog.expectedProperties	Currently expected properties.
dialog.lastEvent	Last surfaced entity ambiguity event.
dialog.lastIntent	The last top scoring intent for this dialog.
dialog.lastTriggerEvent	Inside a forms dialog, the forms event which triggered an Ask action. Note, the forms feature is in preview.
dialog.requiredProperties	Currently required properties.
dialog.retries	Number of retries for the current Ask.

## This scope

- [Composer v2.x](#)
- [Composer v1.x](#)

VARIABLE	DESCRIPTION
this.options	The options passed into the active dialog via the options argument of the BeginDialog action.
this.turnCount	The number of previous times the user was prompted for this information. Applies to input actions.
this.value	The value to use as the default value. Applies to input actions. Can be an adaptive expression.

## Turn scope

- [Composer v2.x](#)
- [Composer v1.x](#)

VARIABLE	DESCRIPTION
turn.activity	The current activity for the turn. For detailed information about the structure of an activity, see the <a href="#">Bot Framework Activity schema</a> .
turn.activity.action	
turn.activity.attachments	A list of objects to be displayed as part of this activity.
turn.activity.entities	A list of metadata objects pertaining to this activity. Unlike attachments, entities don't necessarily manifest as user-interactable content elements, and are intended to be ignored if not understood.
turn.activity.from	Indicates which client, bot, or channel generated an activity.
turn.activity.importance	For message activities: contains an enumerated set of values to signal to the recipient the relative importance of the activity. It's up to the receiver to map these importance hints to the user experience.
turn.activity.locale	For message activities: the language code of the <code>text</code> field.
turn.activity.localTimestamp	The datetime and timezone offset where the activity was generated. This may be different from the UTC <code>timestamp</code> where the activity was recorded.
turn.activity.localTimezone	The timezone where the activity was generated.
turn.activity.name	
turn.activity.recipient	Indicates which client or bot is receiving this activity.
turn.activity.semanticAction	For message activities: contains an optional programmatic action accompanying the user request. The semantic action field is populated by the channel and bot based on some understanding of what the user is trying to accomplish.
turn.activity.speak	For message activities: indicates how the activity should be spoken via a text-to-speech system.
turn.activity.suggestedActions	For message activities: contains a payload of interactive actions that may be displayed to the user. Support for suggestedActions and their manifestation depends heavily on the channel.
turn.activity.summary	For message activities: contains text used to replace attachments on channels that don't support them.
turn.activity.text	For message activities: the text content, either in the Markdown format, XML, or as plain text.
turn.activity.timestamp	The exact UTC time when the activity occurred, as recorded by the channel.

VARIABLE	DESCRIPTION
turn.activity.topicName	
turn.activity.type	The activity's type, such as "message", "event", or "conversationUpdate", and so on.
turn.activity.value	For message activities, contains a programmatic payload specific to the activity being sent.
turn.activityProcessed	Indicates whether the original turn activity was consumed. Used when
turn.dialogEvent	The current <i>dialog event</i> for this turn.
turn.interrupted	Indicates whether an interruption occurred.
turn.lastresult	The result from the last dialog that was called.
turn.recognized	The recognized result for the current turn.
turn.recognized.alteredText	The input text as modified by the recognizer, for example for spelling correction.
turn.recognized.entities	All recognized top-level entities.
turn.recognized.intent	The top scoring intent.
turn.recognized.intents	All recognized intents, with the intent as key and the confidence as value.
turn.recognized.score	The score for the top intent.
turn.recognized.text	The input text to the recognizer.
turn.recognizedEntities	Entities recognized from the text.
turn.repeatedIds	For internal use only.
turn.unrecognizedText	The original text, split into unrecognized strings.