

Table of Contents

[Overview](#)

[About Bot Framework](#)

[How it works](#)

[Add intelligence to bots](#)

[Start building bots](#)

[Quickstarts](#)

[Create bot - Azure Bot Service](#)

[Create bot - .NET SDK](#)

[Create bot - Node.js SDK](#)

[Create bot - REST](#)

[Samples](#)

[.NET](#)

[Node.js](#)

[How-To guides](#)

[Design bots](#)

[Principles of bot design](#)

[First interaction](#)

[Conversation flow](#)

[Navigation](#)

[UX elements](#)

[Patterns](#)

[Develop with Azure Bot Service](#)

[Choose a hosting plan](#)

[Bot templates](#)

[Build features](#)

[Publish a bot](#)

[Migrate a Consumption plan bot into a web app bot](#)

[Develop with .NET](#)

[Key concepts](#)

[Messages and activities](#)

[Dialogs](#)

[FormFlow](#)

[Channels](#)

[State data](#)

[Recognize intent with LUIS](#)

[Request payment](#)

[Add Azure Search](#)

[Secure your bot](#)

[SDK release notes](#)

[SDK reference](#)

[Develop with Node.js](#)

[Key concepts](#)

[Dialogs](#)

[Messages](#)

[Channels](#)

[State Data](#)

[Recognize intent from message content](#)

[Recognize intent with LUIS](#)

[Handle user and conversation events](#)

[Support localization](#)

[Use backchannel mechanism](#)

[Request payment](#)

[Add Azure Search](#)

[SDK reference](#)

[Develop with REST](#)

[Build a bot](#)

[Build a client](#)

[Test and debug](#)

[Debug with the emulator](#)

[Debug with Visual Studio Code](#)

[Debug an Azure Bot Service bot](#)

[Test a Cortana skill](#)

[Deploy bots](#)

[Deploy from local git repo](#)

[Deploy from GitHub](#)

[Deploy from Visual Studio](#)

[Manage](#)

[Register a bot](#)

[Enable analytics](#)

[Connect to channels](#)

[Configure a bot](#)

[Preview bot features](#)

[Bing](#)

[Cortana](#)

[Skype for Business](#)

[Direct Line](#)

[Microsoft Teams](#)

[Skype](#)

[Web Chat](#)

[Email](#)

[GroupMe](#)

[Facebook](#)

[Kik](#)

[Slack](#)

[Telegram](#)

[Twilio](#)

[Bot review guidelines](#)

[Troubleshoot](#)

[Troubleshoot general problems](#)

[Troubleshoot authentication](#)

[Reference](#)

[Bot Framework reference](#)

[.NET](#)

[Node](#)

[REST API](#)

[Resources](#)

[FAQ](#)

[Get support](#)

[Guide to identifiers](#)

[App Insights keys](#)

[Upgrade your bot to v3](#)

[User-agent requests](#)

About the Bot Framework

10/4/2017 • 2 min to read • [Edit Online](#)

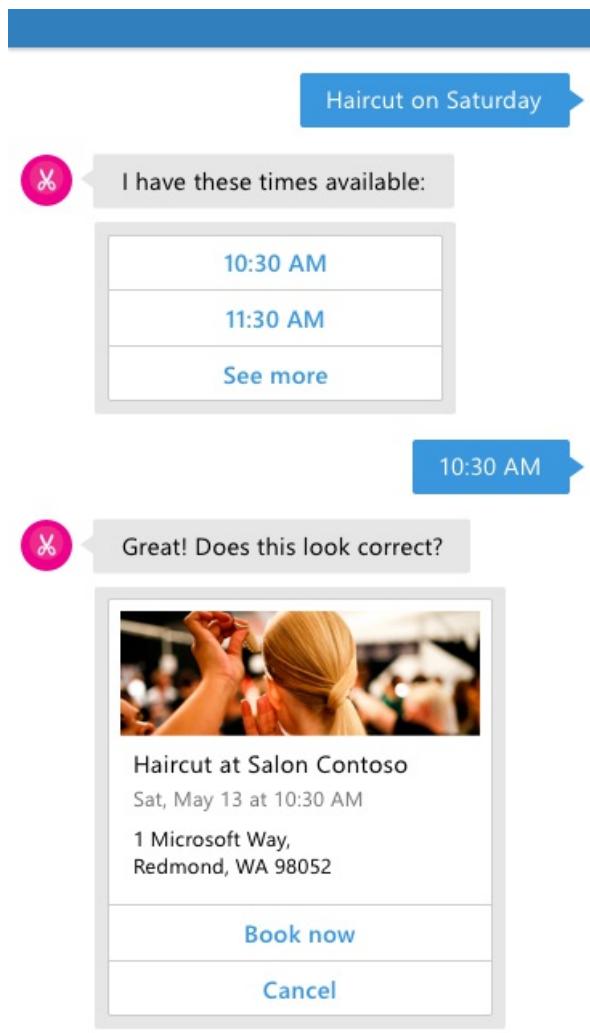
The Bot Framework is a platform for building, connecting, testing, and deploying powerful and intelligent bots. You can quickly [start building bots](#) with the [Azure Bot Service](#). If you prefer building a bot from scratch, the Bot Framework provides the Bot Builder SDK for [.NET](#) and [Node.js](#).

What is a bot?

Think of a bot as an app that users interact with in a conversational way. Bots can communicate conversationally with text, cards, or speech. A bot may be as simple as basic pattern matching with a response, or it may be a sophisticated weaving of artificial intelligence techniques with complex conversational state tracking and integration to existing business services.

The Bot Framework enables you to build bots that support different types of interactions with users. You can design conversations in your bot to be freeform. Your bot can also have more guided interactions where it provides the user choices or actions. The conversation can use simple text strings or more complex rich cards that contain text, images, and action buttons. And you can add natural language interactions, which let your users interact with your bots in a natural and expressive way.

Let's look at an example of a bot that schedules salon appointments. The bot understands the user's intent, presents appointment options using action buttons, displays the user's selection when they tap an appointment, and then sends a thumbnail card that contains the appointment's specifics.



Bots are rapidly becoming an integral part of digital experiences. They are becoming as essential as a website or a mobile experience for users to interact with a service or application.

Why use the Bot Framework?

Developers writing bots all face the same problems: bots require basic I/O, they must have language and dialog skills, and they must connect to users, preferably in any conversation experience and language the user chooses. The Bot Framework provides powerful tools and features to help solve these problems.

Azure Bot Service

The Azure Bot Service accelerates the process of developing a bot. It provisions a resource in Azure and gives you five bot templates you can choose from when you create a bot. You can further modify your bot directly in the browser using the Azure editor or in an Integrated Development Environment (IDE), such as Visual Studio and Visual Studio Code.

Bot Builder

The Bot Framework includes [Bot Builder](#), which provides a rich and full-featured SDK for the .NET and Node.js platforms. The SDK provides features that make interactions between bots and users much simpler. Bot Builder also includes an emulator for debugging your bots, as well as a large set of sample bots you can use as building blocks.

Bot Framework portal

The Bot Framework portal gives you one convenient place to register, connect, and manage your bot. It also provides diagnostic tools and a web chat control you can use to embed your bot in a web page.

Edit anewbot

Bot profile



Icon
[Upload custom icon](#)
30K max, png only

* Display Name [?](#)

* Bot handle [?](#)

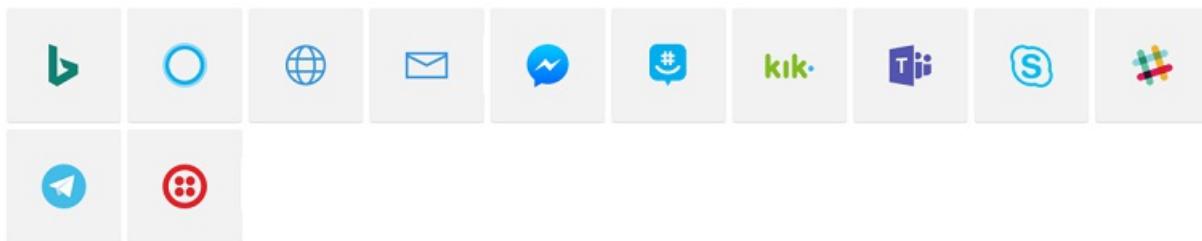
* Long Description [?](#)

Configuration

Messaging endpoint

Channels

The Bot Framework supports several popular channels for connecting your bots and people. Users can start conversations with your bot on any channel that you've configured your bot to work with, including email, Facebook, Skype, Slack, and SMS. You can use the [Channel Inspector](#) to preview features you want to use on these targeted channels.



Next steps

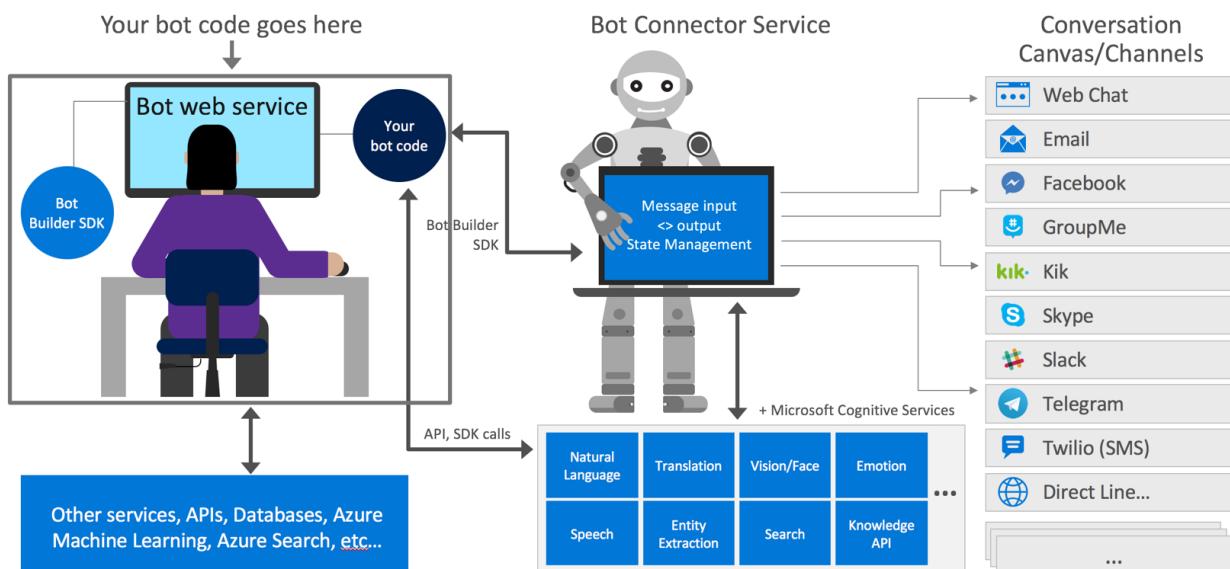
Dive deeper into [the capabilities](#) of the Bot Framework. Get started [building your first bot](#) and learn more about [designing great bots](#).

How the Bot Framework works

10/4/2017 • 6 min to read • [Edit Online](#)

The Bot Framework provides tools and services to help you build, deploy, and publish bots, including the Azure Bot Service, the Bot Builder SDK, the Developer Portal, and the Bot Connector.

Bots are apps that offer a conversational interface as the best solution to the user's needs. Like other apps, bot development starts with your business logic and data. Users can interact with your bot with inline forms and cards, simple menus, or natural language. Choose the model your bot will use to interact with users. You can build your bot with the Azure Bot Service or you can start with the Bot Builder SDK using C# or Node.js. Add artificial intelligence to your bot with Cognitive Services. Register your bot on the Developer Portal and connect it to users through the channels they use, such as Facebook, Kik, and Microsoft Teams. When you are ready to share your bot with the world, deploy it to a cloud service such as Microsoft Azure.



Azure Bot Service

The [Azure Bot Service](#) provides an integrated environment purpose-built for bot development. You can write a bot, connect, test, deploy, and manage it from your web browser with no separate editor or source control required. For simple bots, you may not need to write code at all. It is powered by the Bot Framework and it provides two [hosting plans](#):

- With the App Service plan, a bot is a standard Azure web app you can set to allocate a predefined capacity with predictable costs and scaling.
- With a Consumption plan, a bot is a serverless bot that runs on Azure Functions and uses the pay-per-run Azure Functions pricing.

The Azure Bot Service accelerates bot development with five bot templates you can choose from when you create a bot. You can further modify your bot directly in the browser using the Azure editor or in an Integrated Development Environment (IDE), such as Visual Studio and Visual Studio Code.

Bot Builder

To help you build bots with C# or JavaScript, the Bot Framework includes Bot Builder SDK. The SDK provides libraries, samples, and tools to help you build and debug bots. The SDK contains built-in dialogs to handle user

interactions ranging from a basic Yes/No to complex disambiguation. Built-in recognizers and event handlers help guide the user through a conversation.

The [Bot Builder SDK for .NET](#) leverages C# to provide a familiar way for .NET developers to write bots. It is a powerful framework for constructing bots that can handle both free-form interactions and more guided conversations where the user selects from possible values.

The [Bot Builder SDK for Node.js](#) provides a familiar way for Node.js developers to write bots. You can use it to build a wide variety of conversational user interfaces, from simple prompts to free-form conversations. The conversational logic for your bot is hosted as a web service. The Bot Builder SDK for Node.js uses restify, a popular framework for building web services, to create the bot's web server. The SDK is also compatible with Express and the use of other web app frameworks is possible with some adaptation.

Core concepts

Understanding the core concepts of the Bot Framework will help you build bots that provide the features your users need. These concepts are covered in more detail in the [Develop with .NET](#) and [Develop with Node.js](#) sections of the documentation.

Channel

A channel is the connection between the Bot Framework and communication apps such as Skype, Slack, Facebook Messenger, Office 365 mail, and others. Use the Developer Portal to configure each channel you want the bot to be available on. Use the [Channel Inspector](#) to preview if a particular feature you want to use is supported on the channel you are targeting. The Skype and web chat channels are automatically pre-configured.

Bot Connector

The Bot Connector service connects a bot to one or more channels and handles the message exchange between them. This connecting service allows the bot to communicate over many channels without manually designing a specific message for each channel's schema.

Activity

The Bot Connector uses an *activity* to exchange information between bot and channel. Any communication going back and forth is an *activity* of some type. Some activities are invisible to the user, such as the notification that the bot has been added to a user's contact list.

Message

A *message* is the most common type of *activity*. A message can be as simple as a text string or contain attachments, interactive elements, and rich cards. For example, adding a bot to the user's contact list could trigger the bot to respond with a message containing a string saying "Thank you!" and an image of a happy face.

Dialog

A *dialog* helps organize the logic in your bot and manages [conversation flow](#). Dialogs are arranged in a [stack](#), and the top dialog in the stack processes all incoming messages until it is closed or a different dialog is invoked. For example, a `BrowseProducts` dialog would contain only the logic and UI related to the user browsing the products; clicking the *Order* button would invoke the `PlaceOrder` dialog.

Rich cards

A rich card comprises a title, description, link, and images. A message can contain multiple rich cards, displayed in either list format or carousel format. The Bot Framework supports the following rich cards:

CARD TYPE	DESCRIPTION
Adaptive card	A card that can contain any combination of text, speech, images, buttons, and input fields.

CARD TYPE	DESCRIPTION
Animation card	A card that can play animated GIFs or short videos.
Audio card	A card that can play an audio file.
Hero card	A card that typically contains a single large image, one or more buttons, and text.
Thumbnail card	A card that typically contains a single thumbnail image, one or more buttons, and text.
Receipt card	A card that enables a bot to provide a receipt to the user. It typically contains the list of items to include on the receipt, tax and total information, and other 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.
Video card	A card that can play videos.

Learn more about adding rich cards using the [Bot Builder SDK for .NET](#) and the [Bot Builder SDK for Node.js](#).

Test and debug

The [Bot Framework Emulator](#) is a desktop application that allows developers to test and debug their bots. The Emulator can communicate with a bot running on *localhost* or remotely through a tunnel. As you chat with your bot, the emulator displays messages as they would appear in the web chat UI and logs JSON requests and responses for later evaluation.

You may also use the debugger included in [Visual Studio Code](#). Debugging some languages may require additional extensions and configuration.

Deploy to the cloud

You can host your bot on any reachable service, such as Azure. You can deploy directly from [Visual Studio](#). You can also deploy a bot with continuous deployment from a [git repository](#) or [GitHub](#).

Register a bot

When you finish your bot, [register](#) it on the Developer Portal. The [Bot Framework Portal](#) provides a dashboard interface to perform many bot management and connectivity tasks such as configuring channels, managing credentials, connecting to Azure App Insights, or generating web embed codes. Registering your bot with the Bot Framework generates unique credentials used for authentication.

Connect to channels

You can use the Developer Portal to provide channel configuration information to the target channel(s). Many channels require a bot to have an account on the channel; some also require an application.

To make a bot discoverable, [connect it to the Bing channel](#). Users will be able to find the bot using Bing search and then interact with it using the channels it is configured to support. Note that not all bots should be discoverable. For example, a bot designed for private use by company employees should not be made generally available through Bing.

Make your bot smarter

Connect the Microsoft Cognitive Services APIs to enhance your bot. Smart conversational bots respond more naturally, understand spoken commands, guide the user's search for data, determine user location, and even recognize a user's intention to interpret what the user meant to do. [Learn more](#) about adding intelligence to a bot.

Next steps

- [Design](#)
- [Quickstart](#)
- [Botbuilder SDK for .NET](#)
- [Botbuilder SDK for Node.js](#)

Add intelligence to bots with Cognitive Services

8/7/2017 • 6 min to read • [Edit Online](#)

Microsoft Cognitive Services lets you tap into a growing collection of powerful AI algorithms developed by experts in the fields of computer vision, speech, natural language processing, knowledge extraction, and web search. The services simplify a variety of AI-based tasks, giving you a quick way to add state-of-the-art intelligence technologies to your bots with just a few lines of code. The APIs integrate into most modern languages and platforms. The APIs are also constantly improving, learning, and getting smarter, so experiences are always up to date.

Intelligent bots respond as if they can see the world as people see it. They discover information and extract knowledge from different sources to provide useful answers, and, best of all, they learn as they acquire more experience to continuously improve their capabilities.

Language understanding

The interaction between users and bots is mostly free-form, so bots need to understand language naturally and contextually. The Cognitive Service Language APIs provide powerful language models to determine what users want, to identify concepts and entities in a given sentence, and ultimately to allow your bots to respond with the appropriate action. The five APIs support several text analytics capabilities, such as spell checking, sentiment detection, language modeling, and extraction of accurate and rich insights from text.

Cognitive Services provides five APIs for language understanding:

- The [Language Understanding Intelligent Service \(LUIS\)](#) is able to process natural language using pre-built or custom-trained language models.
- The [Text Analytics API](#) detects sentiment, key phrases, topics, and language from text.
- The [Bing Spell Check API](#) provides powerful spell check capabilities, and is able to recognize the difference between names, brand names, and slang.
- The [Linguistic Analysis API](#) uses advanced linguistic analysis algorithms to process text, and perform operations such as breaking down the structure of the text, or performing part-of-speech tagging and parsing.
- The [Web Language Model \(WebLM\) API](#) can be used to automate a variety of natural language processing tasks, such as word frequency or next-word prediction, using advanced language modeling algorithms.

Learn more about [language understanding](#) with Microsoft Cognitive Services.

Knowledge extraction

Cognitive Services provides five knowledge APIs that enable you to identify named entities or phrases in unstructured text, add personalized recommendations, provide auto-complete suggestions based on natural interpretation of user queries, and search academic papers and other research like a personalized FAQ service.

- The [Entity Linking Intelligence Service](#) annotates unstructured text with the relevant entities mentioned in the text. Depending on the context, the same word or phrase may refer to different things. This service understands the context of the supplied text and will identify each entity in your text.
- The [Recommendations API](#) provides "frequently bought together" recommendations to a product, as well as personalized recommendations based on a user's history. Use this service to build and train a model

based on data that you provide, and then use this model to add recommendations to your application.

- The [Knowledge Exploration Service](#) provides natural language interpretation of user queries and returns annotated interpretations to enable rich search and auto-completion experiences that anticipate what the user is typing. Instant query completion suggestions and predictive query refinements are based on your own data and application-specific grammars to enable your users to perform fast queries.
- The [Academic Knowledge API](#) returns academic research papers, authors, journals, conferences, topics, and universities from the [Microsoft Academic Graph](#). Built as a domain-specific example of the Knowledge Exploration Service, the Academic Knowledge API provides a knowledge base using a graph-like dialog with search capabilities over hundreds of millions of research-related entities. Search for a topic, a professor, a university, or a conference, and the API will provide relevant publications and related entities. The grammar also supports natural queries like "Papers by Michael Jordan about machine learning after 2010".
- The [QnA Maker](#) is a free, easy-to-use, REST API and web-based service that trains AI to respond to users' questions in a natural, conversational way. With optimized machine learning logic and the ability to integrate industry-leading language processing, QnA Maker distills semi-structured data like question and answer pairs into distinct, helpful answers.

Learn more about [knowledge extraction](#) with Microsoft Cognitive Services.

Speech recognition and conversion

Use the Speech APIs to add advanced speech skills to your bot that leverage industry-leading algorithms for speech-to-text and text-to-speech conversion, as well as speaker recognition. The Speech APIs use built-in language and acoustic models that cover a wide range of scenarios with high accuracy.

For applications that require further customization, you can use the Custom Recognition Intelligent Service (CRIS). This allows you to calibrate the language and acoustic models of the speech recognizer by tailoring it to the vocabulary of the application, or even to the speaking style of your users.

There are three Speech APIs available in Cognitive Services to process or synthesize speech:

- The [Bing Speech API](#) provides speech-to-text and text-to-speech conversion capabilities.
- The [Custom Recognition Intelligent Service \(CRIS\)](#) allows you to create custom speech recognition models to tailor the speech-to-text conversion to an application's vocabulary or user's speaking style.
- The [Speaker Recognition API](#) enables speaker identification and verification through voice.

The following resources provide additional information about adding speech recognition to your bot.

- [Bot Conversations for Apps video overview](#)
- [Microsoft.Bot.Client library for UWP or Xamarin apps](#)
- [Bot Client Library Sample](#)
- [Speech-enabled WebChat Client](#)

Learn more about [speech recognition and conversion](#) with Microsoft Cognitive Services.

Web search

The Bing Search APIs enable you to add intelligent web search capabilities to your bots. With a few lines of code, you can access billions of webpages, images, videos, news, and other result types. You can configure the APIs to return results by geographical location, market, or language for better relevance. You can further customize your search using the supported search parameters, such as Safesearch to filter out adult content, and Freshness to return results according to a specific date.

There are five Bing Search APIs available in Cognitive Services.

- The [Web Search API](#) provides web, image, video, news and related search results with a single API call.
- The [Image Search API](#) returns image results with enhanced metadata (dominant color, image kind, etc.) and supports several image filters to customize the results.
- The [Video Search API](#) retrieves video results with rich metadata (video size, quality, price, etc.), video previews, and supports several video filters to customize the results.
- The [News Search API](#) finds news articles around the world that match your search query or are currently trending on the Internet.
- The [Autosuggest API](#) offers instant query completion suggestions to complete your search query faster and with less typing.

Learn more about [web search](#) with Microsoft Cognitive Services.

Image and video understanding

The Vision APIs bring advanced image and video understanding skills to your bots. State-of-the-art algorithms allow you to process images or videos and get back information you can transform into actions. For example, you can use them to recognize objects, people's faces, age, gender or even feelings.

The Vision APIs support a variety of image understanding features. They can identify mature or explicit content, estimate and accent colors, categorize the content of images, perform optical character recognition, and describe an image with complete English sentences. The Vision APIs also support several image and video processing capabilities, such as intelligently generating image or video thumbnails, or stabilizing the output of a video.

Cognitive Services provide four APIs you can use to process images or videos:

- The [Computer Vision API](#) extracts rich information about images (such as objects or people), determines if the image contains mature or explicit content, and processes text (using OCR) in images.
- The [Emotion API](#) analyzes human faces and recognizes their emotion across eight possible categories of human emotions.
- The [Face API](#) detects human faces, compares them to similar faces, and can even organize people into groups according to visual similarity.
- The [Video API](#) analyzes and processes video to stabilize video output, detects motion, tracks faces, and can generate a motion thumbnail summary of the video.

Learn more about [image and video understanding](#) with Microsoft Cognitive Services.

Additional resources

You can find comprehensive documentation of each product and their corresponding API references in the [Cognitive Services documentation](#).

Build bots with the Bot Framework

11/1/2017 • 3 min to read • [Edit Online](#)

The Bot Framework provides services, libraries, samples, and tools to help you build and debug bots. The Azure Bot Service is the best way to get started with building bots. With the Azure Bot Service, you can create and deploy a bot in just a few minutes. You can use the Bot Builder SDK to create a bot from scratch using C# or Node.js. And if you want to create a bot using any programming language you want, you can use the REST API.

Azure Bot Service

Azure Bot Service provides an integrated environment that is purpose-built for bot development, enabling you to build, connect, test, deploy, and manage intelligent bots, all from one place. You can write your bot in C# or Node.js directly in the browser using the Azure editor. Your bot is automatically deployed to Azure.

The [Azure Bot Service Quickstart](#) will guide you through creating a bot with the Azure Bot Service.

Bot Builder

The Bot Framework includes Bot Builder to help you develop bots. Bot Builder is an open-source SDK with support for .NET and Node.js. Using Bot Builder, you can get started and have a working bot in just a few minutes.

Bot Builder SDK for .NET

The Bot Builder SDK for .NET leverages C# to provide a familiar way for .NET developers to write bots. It is a powerful framework for constructing bots that can handle both free-form interactions and more guided conversations where the user selects from possible values.

The [.NET Quickstart](#) will guide you through creating a bot with the Bot Builder SDK for .NET.

The Bot Builder SDK for .NET is available as a [NuGet package](#).

IMPORTANT

The Bot Builder SDK for .NET requires .NET Framework 4.6 or newer. If you are adding the SDK to an existing project targeting a lower version of the .NET Framework, you will need to update your project to target .NET Framework 4.6 first.

To install the SDK in an existing Visual Studio project, complete the following steps:

1. In **Solution Explorer**, right-click the project name and select **Manage NuGet Packages...**
2. On the **Browse** tab, type "Microsoft.Bot.Builder" into the search box.
3. Select **Microsoft.Bot.Builder** in the list of results, click **Install**, and accept the changes.

You can also download the Bot Builder SDK for .NET [source code](#) from GitHub.

Visual Studio project templates

Download Visual Studio project templates to accelerate bot development.

- [Bot template for Visual Studio](#) for developing bots with C#
- [Cortana skill template for Visual Studio](#) for developing Cortana skills with C#

TIP

[Learn more](#) about how to install Visual Studio 2017 project templates.

Bot Builder SDK for Node.js

The Bot Builder SDK for Node.js provides a familiar way for Node.js developers to write bots. You can use it to build a wide variety of conversational user interfaces, from simple prompts to free-form conversations. The Bot Builder SDK for Node.js uses restify, a popular framework for building web services, to create the bot's web server. The SDK is also compatible with Express.

The [Node.js Quickstart](#) will guide you through creating a bot with the Bot Builder SDK for Node.js.

The Bot Builder SDK for Node.js is available as an npm package. To install the Bot Builder for Node.js SDK and its dependencies, first create a folder for your bot, navigate to it, and run the following **npm** command:

```
npm init
```

Next, install the Bot Builder SDK for Node.js and [Restify](#) modules by running the following **npm** commands:

```
npm install --save botbuilder  
npm install --save restify
```

You can also download the Bot Builder SDK for Node.js [source code](#) from GitHub.

REST API

You can create a bot with any programming language by using the Bot Framework REST API. The [REST Quickstart](#) will guide you through creating a bot with REST.

There are three REST APIs in the Bot Framework:

- The [Bot Connector REST API](#) enables your bot to send and receive messages to channels configured in the [Bot Framework Portal](#).
- The [Bot State REST API](#) enables your bot to store and retrieve state associated with the conversations that are conducted through the Bot Connector REST API.
- The [Direct Line REST API](#) enables you to connect your own application, such as a client application, web chat control, or mobile app, directly to a single bot.

Bot Framework Emulator

The Bot Framework Emulator is a desktop application that allows you to test and debug your bots, either locally or remotely. Using the emulator, you can chat with your bot and inspect the messages that your bot sends and receives. [Learn more about the Bot Framework emulator](#) and [download the emulator](#) for your platform.

Bot Framework Channel Inspector

Quickly see what bot features will look like on different channels with the Bot Framework [Channel Inspector](#).

Create a bot with the Azure Bot Service

10/12/2017 • 3 min to read • [Edit Online](#)

The Azure Bot Service accelerates the process of developing a bot by provisioning a web host with one of five bot templates you can modify in an integrated environment that is purpose-built for bot development. This tutorial walks you through the process of creating and testing a bot by using the Azure Bot Service.

Prerequisites

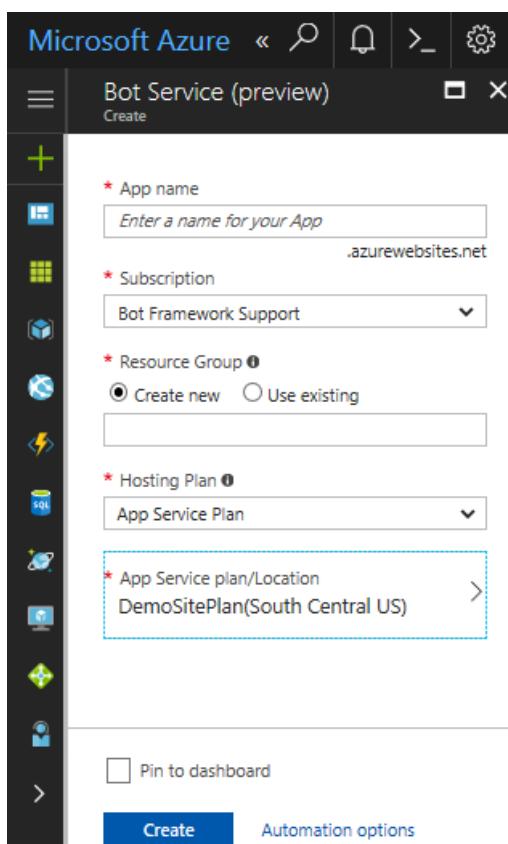
You must have a Microsoft Azure subscription before you can use the Azure Bot Service. If you do not already have a subscription, you can register for a [free trial](#).

Create your bot

To create a bot by using the Azure Bot Service, sign in to [Azure](#) and complete the following steps.

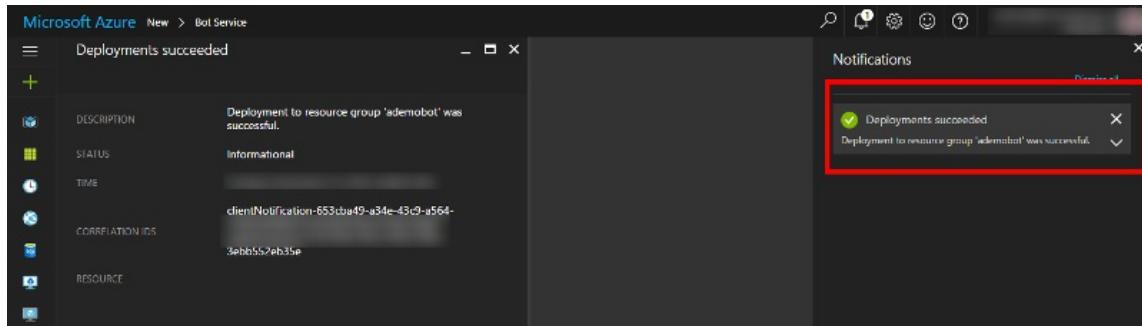
Create a new bot service

1. Select **New** in the menu blade.
2. In the **New** blade, navigate to the Data + Analytics category, and select **Bot Service**.
3. In the Bot Service blade, provide the requested information, and click **Create** to create the bot service and deploy it to the cloud.
 - Set **App name** to your bot's name. The name is used as the subdomain when your bot is deployed to the cloud (for example, *mybasicbot.azurewebsites.net*).
 - Select the subscription to use.
 - Select the [resource group](#), [hosting plan](#), and [location](#).



4. Confirm that the bot service has been deployed.

- Click **Notifications** (the bell icon that is located along the top edge of the Azure portal). The notification will change from **Deployment started** to **Deployment succeeded**.
- After the notification changes to **Deployment succeeded**, click **Go to resource** on that notification.



Select your programming language

Choose the programming language that you want to use to develop your bot.

A screenshot of the Microsoft Azure portal showing the 'myhikingtrailsbot' resource. A modal window titled 'Choose a template' is open, showing two options: 'C#' and 'NodeJS'. The 'C#' option is highlighted with a blue border.

Select a template and create the bot

Select the template to use as the starting point for developing your bot. For this tutorial, choose the **Basic** template.

A screenshot of the Microsoft Azure portal showing the 'Choose a template' step. It displays five template options: 'Basic' (selected), 'Form', 'Language understanding', 'Question and Answer', and 'Proactive'. Each template has a brief description. Below the templates is a 'Next' button.

Then, click **Next** to create the bot based on the programming language and template that you've chosen.

Create App ID and password

Next, create an app ID and password for your bot, so that it will be able to authenticate with the Bot Framework.

1. Click **Create Microsoft App ID and password**.

Microsoft Azure Deployments succeeded >

myhikingtrailsbot

Create a Microsoft App ID

In order to authenticate your bot with the Bot Framework, you'll need to register your application and generate an app ID and password.

1. Register your bot with Microsoft to generate a new ID and password

Create Microsoft App ID and password

2. Paste your app ID and password below to continue

Microsoft App ID from the Microsoft App registration portal

Paste password from the Microsoft App registration protal



2. On the page that opens in a new browser tab, click **Generate an app password to continue**.
3. Copy and securely store the password that is shown, and then click **Ok**.
4. Click **Finish and go back to Bot Framework**.
5. Back in the Azure Portal, assure the **app ID** field is auto-populated for you, and paste the password that you copied (in step 3 above) into the password field.

TIP

If the **app ID** field is not auto-populated, you can retrieve it by signing in to the [Microsoft Application Registration Portal](#) and copying the application ID from your application's registration settings.

Microsoft Azure Deployments succeeded >

myhikingtrailsbot

Create a Microsoft App ID

In order to authenticate your bot with the Bot Framework, you'll need to register your application and generate an app ID and password.

1. Register your bot with Microsoft to generate a new ID and password

Manage Microsoft App ID and password

2. Paste your app ID and password below to continue

e2bd1ca7-3157-4072-b77a-a14d45705e5c

Paste password from the Microsoft App registration portal



6. Agree to terms, and click **Create bot**.

IMPORTANT

When you click **Create bot**, there may be a slight delay before a splash screen renders to indicate that the bot service is generating your bot. *Do not click **Create bot** again.* Please wait for the splash screen to appear.

When the bot service finishes generating your bot, the Azure editor will contain the bot's source files. At this point, the bot has been created, registered with the Bot Framework, deployed to the cloud, and is fully functional. If you sign in to the [Bot Framework Portal](#), you'll see that your bot is now listed under **My bots**. Congratulations! You've successfully created a bot by using the Azure Bot Service!

The screenshot shows the Microsoft Bot Framework developer portal. At the top, there's a navigation bar with links for 'Bot Framework', 'My bots', 'Documentation', and 'Blog'. Below the navigation bar, the current bot 'my-hiking-trails-bot' is selected. A 'CHANNELS' tab is active, showing two channels: 'Skype' and 'Web Chat', both listed as 'Running'. There are 'Edit' buttons next to each entry. A blue 'Test' button is located at the bottom right of the channel list. The URL in the browser is dev.botframework.com/bots?id=the-hiking-trails-bot.

Connect to channels

Name	Health	Published	
Skype	Running	--	Edit
Web Chat	Running	--	Edit

[Get bot embed codes](#)

Add a channel



Test your bot

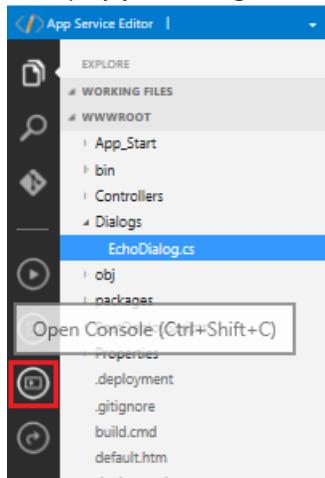
Now that your bot is running in the cloud, try it out by typing a few messages into the built-in chat control that's located to the right of the code editor in Azure. You should see that the bot responds to each message you send by echoing back your message prefixed with the text *You said.*

The screenshot shows the Microsoft Azure portal with the 'my-biking-bot' bot service selected. The left sidebar lists various resources under 'my-biking-bot'. The main area displays the 'CHANNELS' section, which is currently set to 'Chat'. It shows the same 'Skype' and 'Web Chat' channels as the previous screenshot. On the right side, there's a large chat window. A message from the user 'You' is shown: 'I want to go biking!'. The bot has responded with 'Welcome You!' and '1: You said I want to go biking!'. At the bottom of the chat window, there's a text input field with placeholder text 'Type your message...' and a send button.

Deploy changes to your web app bot

If you chose the App Service plan, follow these steps to modify your bot source and re-deploy your changes.

1. In Azure, click your bot's **BUILD** tab, and click **Open online code editor**.
2. Open the Dialogs folder, and click `EchoDialog.cs`.
3. Change text in line 22 from `You said` to `You just said`.
4. To deploy your changed source, click the Open Console icon.



5. In the Console window, type **build.cmd**, and press the enter key.

The console window shows the deployment's progress until it's complete.

NOTE

A bot on a Consumption plan automatically deploys each time you modify a source file in the online editor.

Next steps

In this tutorial, you created a simple bot by using the Azure Bot Service and verified the bot's functionality by using the built-in chat control within Azure. At this point, you may want to [add more functionality](#) to your bot or set up [continuous deployment](#). You can also configure your bot to run on one or more channels and publish your bot, without ever leaving the Azure portal.

Create a bot with the Bot Builder SDK for .NET

9/25/2017 • 4 min to read • [Edit Online](#)

The [Bot Builder SDK for .NET](#) is an easy-to-use framework for developing bots using Visual Studio and Windows. The SDK leverages C# to provide a familiar way for .NET developers to create powerful bots.

This tutorial walks you through building a bot by using the Bot Application template and the Bot Builder SDK for .NET, and then testing it with the Bot Framework Emulator.

IMPORTANT

The Bot Builder SDK for .NET currently supports C#. Visual Studio for Mac is not supported.

Prerequisites

Get started by completing the following prerequisite tasks:

1. Install [Visual Studio 2017](#) for Windows.
2. In Visual Studio, [update all extensions](#) to their latest versions.
3. Download the [Bot Application](#), [Bot Controller](#), and [Bot Dialog](#) .zip files. Install the templates by copying the .zip files to your Visual Studio 2017 project templates directory.

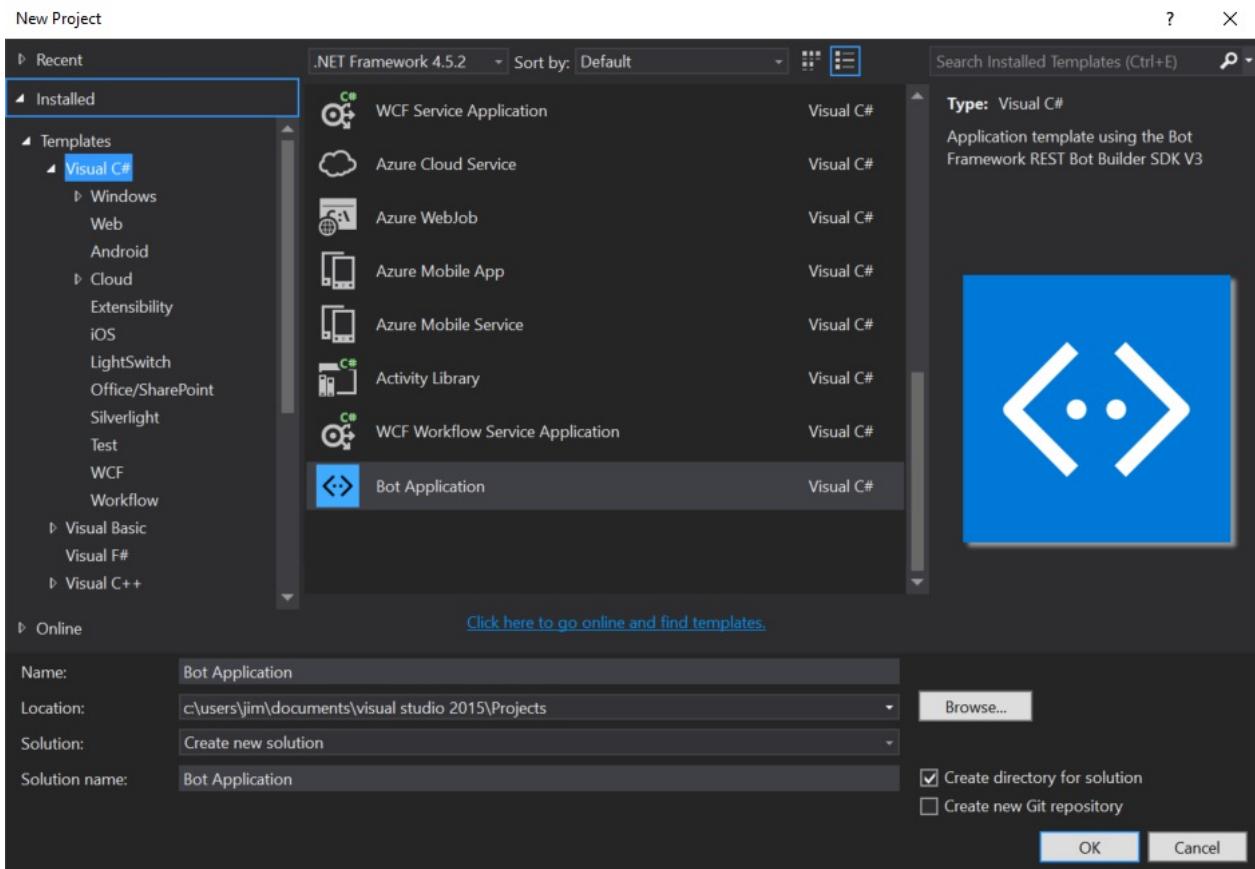
TIP

The Visual Studio 2017 project templates directory is typically located at

```
%USERPROFILE%\Documents\Visual Studio 2017\Templates\ProjectTemplates\Visual C#\ .
```

Create your bot

Next, open Visual Studio and create a new C# project. Choose the Bot Application template for your new project.



NOTE

Visual Studio might say you need to [download and install IIS Express](#).

By using the Bot Application template, you're creating a project that already contains all of the components that are required to build a simple bot, including a reference to the Bot Builder SDK for .NET, `Microsoft.Bot.Builder`. Verify that your project references the latest version of the SDK:

1. Right-click on the project and select **Manage NuGet Packages**.
2. In the **Browse** tab, type "Microsoft.Bot.Builder".
3. Locate the `Microsoft.Bot.Builder` package in the list of search results, and click the **Update** button for that package.
4. Follow the prompts to accept the changes and update the package.

Thanks to the Bot Application template, your project contains all of the code that's necessary to create the bot in this tutorial. You won't actually need to write any additional code. However, before we move on to testing your bot, take a quick look at some of the code that the Bot Application template provided.

Explore the code

First, the `Post` method within `Controllers\MessagesController.cs` receives the message from the user and invokes the root dialog.

```

[BotAuthentication]
public class MessagesController : ApiController
{
    /// <summary>
    /// POST: api/Messages
    /// Receive a message from a user and reply to it
    /// </summary>
    public async Task<HttpResponseMessage> Post([FromBody]Activity activity)
    {
        if (activity.Type == ActivityTypes.Message)
        {
            await Conversation.SendAsync(activity, () => new Dialogs.RootDialog());
        }
        else
        {
            HandleSystemMessage(activity);
        }
        var response = Request.CreateResponse(HttpStatusCode.OK);
        return response;
    }
    ...
}

```

The root dialog processes the message and generates a response. The `MessageReceivedAsync` method within **Dialogs\RootDialog.cs** sends a reply that echos back the user's message, prefixed with the text 'You sent' and ending in the text 'which was ## characters', where ## represents the number of characters in the user's message.

```

[Serializable]
public class RootDialog : IDialog<object>
{
    public Task StartAsync(IDialogContext context)
    {
        context.Wait(MessageReceivedAsync);
        return Task.CompletedTask;
    }

    private async Task MessageReceivedAsync(IDialogContext context, IAwaitable<object> result)
    {
        var activity = await result as Activity;

        // calculate something for us to return
        int length = (activity.Text ?? string.Empty).Length;

        // return our reply to the user
        await context.PostAsync($"You sent {activity.Text} which was {length} characters");

        context.Wait(MessageReceivedAsync);
    }
}

```

Test your bot

Next, test your bot by using the [Bot Framework Emulator](#) to see it in action. The emulator is a desktop application that lets you test and debug your bot on localhost or running remotely through a tunnel.

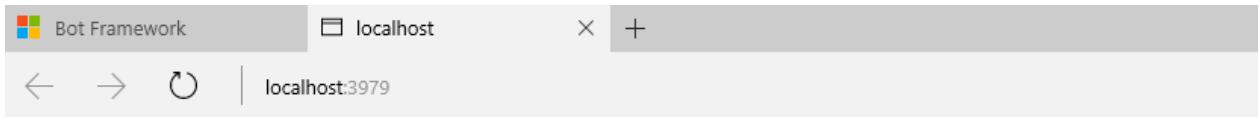
First, you'll need to download and install the emulator. Click [here](#) to download the emulator. After the download completes, launch the executable and complete the installation process.

Start your bot

After installing the emulator, start your bot in Visual Studio by using a browser as the application host. This Visual Studio screenshot shows that the bot will launch in Microsoft Edge when the run button is clicked.

```
using System;
using System.Linq;
using System.Net;
using System.Net.Http;
using System.Web.Http;
using System.Web.Http.Description;
using Microsoft.Bot.Connector;
using Microsoft.Bot.Connector.Utilities;
using Newtonsoft.Json;
```

When you click the run button, Visual Studio will build the application, deploy it to localhost, and launch the web browser to display the application's **default.htm** page. For example, here's the application's **default.htm** page shown in Microsoft Edge:



BotApplication1

Describe your bot here and your terms of use etc.

Visit [Bot Framework](#) to register your bot. When you register it, remember to set your bot's endpoint to
`https://your_bots_hostname/api/messages`

NOTE

You can modify the **default.htm** file within your project to specify the name and description of your bot application.

Start the emulator and connect your bot

At this point, your bot is running locally. Next, start the emulator and then connect to your bot in the emulator:

1. Type `http://localhost:port-number/api/messages` into the address bar, where **port-number** matches the port number shown in the browser where your application is running.
2. Click **Connect**. You won't need to specify **Microsoft App ID** and **Microsoft App Password**. You can leave these fields blank for now. You'll get this information later when you register your bot.

TIP

In the example shown above, the application is running on port number **3979**, so the emulator address would be set to:

`http://localhost:3979/api/messages`.

Test your bot

Now that your bot is running locally and is connected to the emulator, test your bot by typing a few messages in the emulator. You should see that the bot responds to each message you send by echoing back your message prefixed with the text 'You sent' and ending with the text 'which was ## characters', where ## is the total number of characters in the message that you sent.

TIP

In the emulator, click on any speech bubble in your conversation. Details about the message will appear in the Details pane, in JSON format.

You've successfully created a bot by using the Bot Application template Bot Builder SDK for .NET!

Next steps

In this quickstart, you created a simple bot by using the Bot Application template and the Bot Builder SDK for .NET and verified the bot's functionality by using the Bot Framework Emulator.

Next, learn about the key concepts of the Bot Builder SDK for .NET.

[Key concepts in the Bot Builder SDK for .NET](#)

Create a bot with the Bot Builder SDK for Node.js

9/25/2017 • 4 min to read • [Edit Online](#)

The Bot Builder SDK for Node.js is a framework for developing bots. It is easy to use and models frameworks like Express & restify to provide a familiar way for JavaScript developers to write bots.

This tutorial walks you through building a bot by using the Bot Builder SDK for Node.js. You can test the bot in a console window and with the Bot Framework Emulator.

Prerequisites

Get started by completing the following prerequisite tasks:

1. Install [Node.js](#).
2. Create a folder for your bot.
3. From a command prompt or terminal, navigate to the folder you just created.
4. Run the following **npm** command:

```
npm init
```

Follow the prompt on the screen to enter information about your bot and npm will create a **package.json** file that contains the information you provided.

Install the SDK

Next, install the Bot Builder SDK for Node.js by running the following **npm** command:

```
npm install --save botbuilder
```

Once you have the SDK installed, you are ready to write your first bot.

For your first bot, you will create a bot that simply echoes back any user input. To create your bot, follow these steps:

1. In the folder that you created earlier for your bot, create a new file named **app.js**.
2. Open **app.js** in a text editor or an IDE of your choice. Add the following code to the file:

```
var builder = require('botbuilder');

var connector = new builder.ConsoleConnector().listen();
var bot = new builder.UniversalBot(connector, function (session) {
    session.send("You said: %s", session.message.text);
});
```

3. Save the file. Now you are ready to run and test out your bot.

Start your bot

Navigate to your bot's directory in a console window and start your bot:

```
node app.js
```

Your bot is now running locally. Try out your bot by typing a few messages in the console window. You should see that the bot responds to each message you send by echoing back your message prefixed with the text "You said:".

Install Restify

Console bots are good text-based clients, but in order to use any of the Bot Framework channels (or run your bot in the emulator), your bot will need to run on an API endpoint. Install [restify](#) by running the following [npm](#) command:

```
npm install --save restify
```

Once you have Restify in place, you're ready to make some changes to your bot.

Edit your bot

You will need to make some changes to your [app.js](#) file.

1. Add a line to require the `restify` module.
2. Change the `ConsoleConnector` to a `ChatConnector`.
3. Include your Microsoft App ID and App Password.
4. Have the connector listen on an API endpoint.

```
var restify = require('restify');
var builder = require('botbuilder');

// Setup Restify Server
var server = restify.createServer();
server.listen(process.env.port || process.env.PORT || 3978, function () {
    console.log('%s listening to %s', server.name, server.url);
});

// Create chat connector for communicating with the Bot Framework Service
var connector = new builder.ChatConnector({
    appId: process.env.MICROSOFT_APP_ID,
    appPassword: process.env.MICROSOFT_APP_PASSWORD
});

// Listen for messages from users
server.post('/api/messages', connector.listen());

// Receive messages from the user and respond by echoing each message back (prefixed with 'You said:')
var bot = new builder.UniversalBot(connector, function (session) {
    session.send("You said: %s", session.message.text);
});
```

5. Save the file. Now you are ready to run and test out your bot in the emulator.

NOTE

You do not need a **Microsoft App ID** or **Microsoft App Password** to run your bot in the Bot Framework Emulator.

Please note that you do not need a Microsoft App ID or App Password to run your bot in the Bot Framework

Emulator.

Test your bot

Next, test your bot by using the [Bot Framework Emulator](#) to see it in action. The emulator is a desktop application that lets you test and debug your bot on localhost or running remotely through a tunnel.

First, you'll need to [download](#) and install the emulator. After the download completes, launch the executable and complete the installation process.

Start your bot

After installing the emulator, navigate to your bot's directory in a console window and start your bot:

```
node app.js
```

Your bot is now running locally.

Start the emulator and connect your bot

After you start your bot, connect to your bot in the emulator:

1. Type `http://localhost:3978/api/messages` into the address bar. (This is the default endpoint that your bot listens to when hosted locally.)
2. Click **Connect**. You won't need to specify **Microsoft App ID** and **Microsoft App Password**. You can leave these fields blank for now. You'll get this information later when you [register your bot](#).

Try out your bot

Now that your bot is running locally and is connected to the emulator, try out your bot by typing a few messages in the emulator. You should see that the bot responds to each message you send by echoing back your message prefixed with the text "You said:".

You've successfully created your first bot using the Bot Builder SDK for Node.js!

Next steps

[Bot Builder SDK for Node.js](#)

Create a bot with the Bot Connector service

9/25/2017 • 4 min to read • [Edit Online](#)

The Bot Connector service enables your bot to exchange messages with channels that are configured in the [Bot Framework Portal](#), by using industry-standard REST and JSON over HTTPS. This tutorial walks you through the process of obtaining an access token from the Bot Framework and using the Bot Connector service to exchange messages with the user.

Get an access token

IMPORTANT

If you have not already done so, you must [register](#) your bot with the Bot Framework to obtain its App ID and password. You will need the bot's App ID and password to get an access token.

To communicate with the Bot Connector service, you must specify an access token in the `Authorization` header of each API request, using this format:

```
Authorization: Bearer ACCESS_TOKEN
```

You can obtain the access token for your bot by issuing an API request.

Request

To request an access token that can be used to authenticate requests to the Bot Connector service, issue the following request, replacing **MICROSOFT-APP-ID** and **MICROSOFT-APP-PASSWORD** with the App ID and password that you obtained when you [registered](#) your bot with the Bot Framework.

```
POST https://login.microsoftonline.com/botframework.com/oauth2/v2.0/token
Host: login.microsoftonline.com
Content-Type: application/x-www-form-urlencoded

grant_type=client_credentials&client_id=MICROSOFT-APP-ID&client_secret=MICROSOFT-APP-
PASSWORD&scope=https%3A%2F%2Fapi.botframework.com%2F.default
```

Response

If the request succeeds, you will receive an HTTP 200 response that specifies the access token and information about its expiration.

```
{
  "token_type": "Bearer",
  "expires_in": 3600,
  "ext_expires_in": 3600,
  "access_token": "eyJhbGciOiJIUzI1Ni..."
}
```

TIP

For more details about authentication in the Bot Connector service, see [Authentication](#).

Exchange messages with the user

A conversation is a series of messages exchanged between a user and your bot.

Receive a message from the user

When the user sends a message, the Bot Framework Connector POSTs a request to the endpoint that you specified when you [registered](#) your bot. The body of the request is an [Activity](#) object. The following example shows the request body that a bot receives when the user sends a simple message to the bot.

```
{  
    "type": "message",  
    "id": "bf3cc9a2f5de...",  
    "timestamp": "2016-10-19T20:17:52.2891902Z",  
    "serviceUrl": "https://smba.trafficmanager.net/apis",  
    "channelId": "channel's name/id",  
    "from": {  
        "id": "1234abcd",  
        "name": "user's name"  
    },  
    "conversation": {  
        "id": "abcd1234",  
        "name": "conversation's name"  
    },  
    "recipient": {  
        "id": "12345678",  
        "name": "bot's name"  
    },  
    "text": "Haircut on Saturday"  
}
```

Reply to the user's message

When your bot's endpoint receives a `POST` request that represents a message from the user (i.e., `type` = **message**), use the information in that request to create the [Activity](#) object for your response.

1. Set the **conversation** property to the contents of the **conversation** property in the user's message.
2. Set the **from** property to the contents of the **recipient** property in the user's message.
3. Set the **recipient** property to the contents of the **from** property in the user's message.
4. Set the **text** and **attachments** properties as appropriate.

Use the `serviceUrl` property in the incoming request to [identify the base URI](#) that your bot should use to issue its response.

To send the response, `POST` your [Activity](#) object to `/v3/conversations/{conversationId}/activities/{activityId}`, as shown in the following example. The body of this request is an [Activity](#) object that prompts the user to select an available appointment time.

```
POST https://smba.trafficmanager.net/apis/v3/conversations/abcd1234/activities/bf3cc9a2f5de...  
Authorization: Bearer eyJhbGciOiJIUzI1Ni...  
Content-Type: application/json
```

```
{  
  "type": "message",  
  "from": {  
    "id": "12345678",  
    "name": "bot's name"  
  },  
  "conversation": {  
    "id": "abcd1234",  
    "name": "conversation's name"  
  },  
  "recipient": {  
    "id": "1234abcd",  
    "name": "user's name"  
  },  
  "text": "I have these times available:",  
  "replyToId": "bf3cc9a2f5de..."  
}
```

In this example request, `https://smba.trafficmanager.net/apis` represents the base URI; the base URI for requests that your bot issues may be different. For details about setting the base URI, see [API Reference](#).

IMPORTANT

As shown in this example, the `Authorization` header of each API request that you send must contain the word **Bearer** followed by the access token that you [obtained from the Bot Framework](#).

To send another message that enables a user to select an available appointment time by clicking a button, `POST` another request to the same endpoint:

```
POST https://smba.trafficmanager.net/apis/v3/conversations/abcd1234/activities/bf3cc9a2f5de...  
Authorization: Bearer eyJhbGciOiJIUzI1Ni...  
Content-Type: application/json
```

```
{
  "type": "message",
  "from": {
    "id": "12345678",
    "name": "bot's name"
  },
  "conversation": {
    "id": "abcd1234",
    "name": "conversation's name"
  },
  "recipient": {
    "id": "1234abcd",
    "name": "user's name"
  },
  "attachmentLayout": "list",
  "attachments": [
    {
      "contentType": "application/vnd.microsoft.card.thumbnail",
      "content": {
        "buttons": [
          {
            "type": "imBack",
            "title": "10:30",
            "value": "10:30"
          },
          {
            "type": "imBack",
            "title": "11:30",
            "value": "11:30"
          },
          {
            "type": "openUrl",
            "title": "See more",
            "value": "http://www.contososalon.com/scheduling"
          }
        ]
      }
    ],
    "replyToId": "bf3cc9a2f5de..."
  }
}
```

Next steps

In this tutorial, you obtained an access token from the Bot Framework and used the Bot Connector service to exchange messages with the user. You can use the [Bot Framework Emulator](#) to test and debug your bot. If you'd like to share your bot with others, you'll need to [configure](#) it to run on one or more channels and [deploy](#) it to the cloud.

To learn more about building great bots with the Bot Framework, see the following articles:

- [How the Bot Framework works](#)
- [Principles of bot design](#)
- [Bot Framework REST APIs](#)
- [Bot Builder SDK for .NET](#)
- [Bot Builder SDK for Node.js](#)
- [Deploy a bot to the cloud](#)
- [Bot Framework FAQ](#)

Bot Builder SDK for .NET samples

8/9/2017 • 2 min to read • [Edit Online](#)

These samples demonstrate task-focused bots that show how to take advantage of features in the Bot Builder SDK for .NET. You can use the samples to help you quickly get started with building great bots with rich capabilities.

Get the samples

To get the samples, clone the [BotBuilder-Samples](#) GitHub repository using Git.

```
git clone https://github.com/Microsoft/BotBuilder-Samples.git  
cd BotBuilder-Samples
```

The sample bots built with the Bot Builder SDK for .NET are organized in the **CSharp** directory.

You can also view the samples on GitHub and deploy them to Azure directly.

Core

These samples show the basic techniques for building rich and capable bots.

SAMPLE	DESCRIPTION
Send Attachment	A sample bot that sends simple media attachments (images) to the user.
Receive Attachment	A sample bot that receives attachments sent by the user and downloads them.
Create New Conversation	A sample bot that starts a new conversation using a previously stored user address.
Get Members of a Conversation	A sample bot that retrieves the conversation's members list and detects when it changes.
Direct Line	A sample bot and a custom client that communicate with each other using the Direct Line API.
Direct Line (WebSockets)	A sample bot and a custom client that communicate with each other using the Direct Line API + WebSockets.
Multi Dialogs	A sample bot that shows various kinds of dialogs.
State API	A stateless sample bot that tracks the context of a conversation.
Custom State API	A stateless sample bot that tracks the context of a conversation using a custom storage provider.
ChannelData	A sample bot that sends native metadata to Facebook using ChannelData.

SAMPLE	DESCRIPTION
AppInsights	A sample bot that logs telemetry to an Application Insights instance.

Search

This sample shows how to leverage Azure Search in data-driven bots.

SAMPLE	DESCRIPTION
Azure Search	Two sample bots that help the user navigate large amounts of content.

Cards

These samples show how to send rich cards in the Bot Framework.

SAMPLE	DESCRIPTION
Rich Cards	A sample bot that sends several different types of rich cards.
Carousel of Cards	A sample bot that sends multiple rich cards within a single message using the Carousel layout.

Intelligence

These samples show how to add artificial intelligence capabilities to a bot using Bing and Microsoft Cognitive Services APIs.

SAMPLE	DESCRIPTION
LUIS	A sample bot that uses LuisDialog to integrate with a LUIS.ai application.
Image Caption	A sample bot that gets an image caption using the Microsoft Cognitive Services Vision API.
Speech To Text	A sample bot that gets text from audio using the Bing Speech API.
Similar Products	A sample bot that finds visually similar products using the Bing Image Search API.
Zummer	A sample bot that finds wikipedia articles using the Bing Search API.

Reference implementation

This sample is designed to showcase an end-to-end scenario. It's a great source of code fragments if you're looking to implement more complex features in your bot.

SAMPLE	DESCRIPTION
Contoso Flowers	A sample bot that uses many features of the Bot Framework.

Bot Builder SDK for Node.js samples

8/9/2017 • 2 min to read • [Edit Online](#)

These samples demonstrate task-focused bots that show how to take advantage of features in the Bot Builder SDK for Node.js. You can use the samples to help you quickly get started with building great bots with rich capabilities.

Get the samples

To get the samples, clone the [BotBuilder-Samples](#) GitHub repository using Git.

```
git clone https://github.com/Microsoft/BotBuilder-Samples.git  
cd BotBuilder-Samples
```

The sample bots built with the Bot Builder SDK for Node.js are organized in the **Node** directory.

You can also view the samples on GitHub and deploy them to Azure directly.

Core

These samples show the basic techniques for building rich and capable bots.

SAMPLE	DESCRIPTION
Send Attachment	A sample bot that sends simple media attachments (images) to the user.
Receive Attachment	A sample bot that receives attachments sent by the user and downloads them.
Create New Conversation	A sample bot that starts a new conversation using a previously stored user address.
Get Members of a Conversation	A sample bot that retrieves the conversation's members list and detects when it changes.
Direct Line	A sample bot and a custom client that communicate with each other using the Direct Line API.
Direct Line (WebSockets)	A sample bot and a custom client that communicate with each other using the Direct Line API + WebSockets.
Multi Dialogs	A sample bot that shows various kinds of dialogs.
State API	A stateless sample bot that tracks the context of a conversation.
Custom State API	A stateless sample bot that tracks the context of a conversation using a custom storage provider.
ChannelData	A sample bot that sends native metadata to Facebook using ChannelData.

SAMPLE	DESCRIPTION
AppInsights	A sample bot that logs telemetry to an Application Insights instance.

Search

This sample shows how to leverage Azure Search in data-driven bots.

SAMPLE	DESCRIPTION
Azure Search	Two sample bots that help the user navigate large amounts of content.

Cards

These samples show how to send rich cards in the Bot Framework.

SAMPLE	DESCRIPTION
Rich Cards	A sample bot that sends several different types of rich cards.
Carousel of Cards	A sample bot that sends multiple rich cards within a single message using the Carousel layout.

Intelligence

These samples show how to add artificial intelligence capabilities to a bot using Bing and Microsoft Cognitive Services APIs.

SAMPLE	DESCRIPTION
LUIS	A sample bot that uses LuisDialog to integrate with a LUIS.ai application.
Image Caption	A sample bot that gets an image caption using the Microsoft Cognitive Services Vision API.
Speech To Text	A sample bot that gets text from audio using the Bing Speech API.
Similar Products	A sample bot that finds visually similar products using the Bing Image Search API.
Zummer	A sample bot that finds wikipedia articles using the Bing Search API.

Reference implementation

This sample is designed to showcase an end-to-end scenario. It's a great source of code fragments if you're looking to implement more complex features in your bot.

SAMPLE	DESCRIPTION
Contoso Flowers	A sample bot that uses many features of the Bot Framework.

Principles of bot design

8/7/2017 • 3 min to read • [Edit Online](#)

The Bot Framework enables developers to create compelling bot experiences that solve a variety of business problems. By learning the concepts described in this section, you'll become equipped to design a bot that aligns with best practices and capitalizes on lessons learned thus far in this relatively new arena.

Designing a bot

If you are building a bot, it is safe to assume that you are expecting users to use it. It is also safe to assume that you are hoping that users will prefer the bot experience over alternative experiences like apps, websites, phone calls, and other means of addressing their particular needs. In other words, your bot is competing for users' time against things like apps and websites. So, how can you maximize the odds that your bot will achieve its ultimate goal of attracting and keeping users? It's simply a matter of prioritizing the right factors when designing your bot.

Factors that do not guarantee a bot's success

When designing your bot, be aware that none of the following factors necessarily guarantee a bot's success:

- **How “smart” the bot is:** In most cases, it is unlikely that making your bot smarter will guarantee happy users and adoption of your platform. In reality, many bots have little advanced machine learning or natural language capabilities. Of course, a bot may include those capabilities if they're necessary to solve the problems that it's designed to address. However, you should not assume any correlation between a bot's intelligence and user adoption of the bot.
- **How much natural language the bot supports:** Your bot can be great at conversations. It can have a vast vocabulary and can even make great jokes. But unless it addresses the problems that your users need to solve, these capabilities may contribute very little to making your bot successful. In fact, some bots have no conversational capability at all. And in many cases, that's perfectly fine.
- **Voice:** It isn't always the case that enabling bots for speech will lead to great user experiences. Often, forcing users to use voice can result in a frustrating user experience. As you design your bot, always consider whether voice is the appropriate channel for the given problem. Is there going to be a noisy environment? Will voice convey the information that needs to be shared with the user?

Factors that do influence a bot's success

Most successful apps or websites have at least one thing in common: a great user experience. Bots are no different in that regard. Therefore, ensuring a great 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 the minimum number of steps?
- 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? Do the users naturally know what to do when using it?

Note that none of these questions directly relates to factors such as how smart the bot is, how much natural language capability it has, whether it uses machine learning, or which programming language was used to create it. Users are unlikely to care about any of these things if the bot solves the problem that they need to address and

delivers a great user experience. A great bot user experience does not require users to type too much, talk too much, repeat themselves several times, or explain things that the bot should automatically know.

TIP

Regardless of the type of application you're creating (bot, website, or app), make user experience a top priority.

The process of designing a bot is like the process of designing an app or website, so the lessons learned from decades of building UI and delivering UX for apps and websites still apply when it comes to designing bots. Whenever you are unsure about the right design approach for your bot, step back and ask yourself the following question: how would you solve that problem in an app or a website? Chances are, the same answer can be applied to bot design.

Next steps

Now that you're familiar with some basic principles of bot design, learn more about designing the user experience, and common patterns in the remainder of this section.

Design a bot's first user interaction

8/7/2017 • 1 min to read • [Edit Online](#)

First impressions matter

The very first interaction between the user and bot is critical to the user experience. When designing your bot, keep in mind that there is more to that first message than just saying "hi." When you build an app, you design the first screen to provide important navigation cues. Users should intuitively understand things such as where the menu is located and how it works, where to go for help, what the privacy policy is, and so on. When you design a bot, the user's first interaction with the bot should provide that same type of information. In other words, just saying "hi" won't be enough.

Language versus menus

Consider the following two designs:

Design 1



Hello user, how can I help you?

Design 2



Hello! How can I help you?

Orders

Products

Help

Starting the bot with an open-ended question such as "How can I help you?" is generally not recommended. If your bot has a hundred different things it can do, chances are users won't be able to guess most of them. Your bot didn't tell them what it can do, so how can they possibly know?

Menus provide a simple solution to that problem. First, by listing the available options, your bot is conveying its capabilities to the user. Second, menus spare the user from having to type too much. They can simply click. Finally, the use of menus can significantly simplify your natural language models by narrowing the scope of input that the bot could receive from the user.

TIP

Menus are a valuable tool when designing bots for a great user experience. Don't dismiss them as not being "smart enough." You may design your bot to use menus while still supporting free form input. If a user responds to the initial menu by typing rather than by selecting a menu option, your bot could attempt to parse the user's text input.

Other considerations

In addition to providing an intuitive and easily navigated first interaction, a well-designed bot provides the user with access to information about its privacy policy and terms of use.

TIP

If your bot collects personal data from the user, it's important to convey that and to describe what will be done with the data.

Next steps

Now that you're familiar with some basic principles for designing the first interaction between user and bot, learn more about [designing the flow of conversation](#).

Design and control conversation flow

9/12/2017 • 3 min to read • [Edit Online](#)

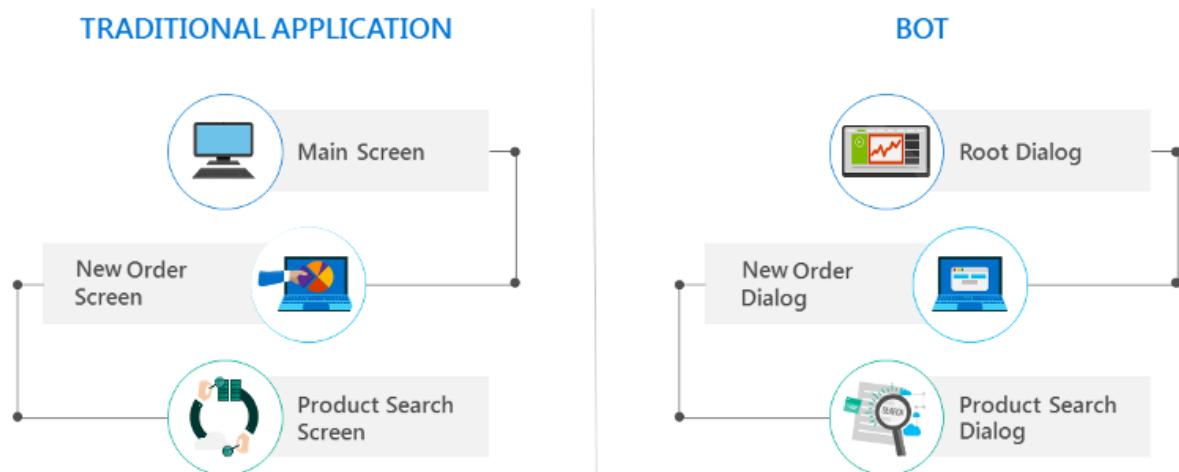
In a traditional application, the user interface (UI) is a series of screens. A single app or website can use one or more screens as needed to exchange information with the user. Most applications start with a main screen where users initially land and provide navigation that leads to other screens for various functions like starting a new order, browsing products, or looking for help.

Like apps and websites, bots have a UI, but it is made up of **dialogs**, rather than screens. Dialogs enable the bot developer to logically separate various areas of bot functionality and guide conversational flow. For example, you may design one dialog to contain the logic that helps the user browse for products and a separate dialog to contain the logic that helps the user create a new order.

Dialogs may or may not have graphical interfaces. They may contain buttons, text, and other elements, or be entirely speech-based. Dialogs also contain actions to perform tasks such as invoking other dialogs or processing user input.

Using dialogs to manage conversation flow

This diagram shows the screen flow of a traditional application compared to the dialog flow of a bot.



In a traditional application, everything begins with the **main screen**. The **main screen** invokes the **new order screen**. The **new order screen** remains in control until it either closes or invokes other screens. If the **new order screen** closes, the user is returned to the **main screen**.

In a bot, everything begins with the **root dialog**. The **root dialog** invokes the **new order dialog**. At that point, the **new order dialog** takes control of the conversation and remains in control until it either closes or invokes other dialogs. If the **new order dialog** closes, control of the conversation is returned back to the **root dialog**.

For a detailed walkthrough of managing conversation flow using dialogs and the Bot Builder SDK, see:

- [Manage conversation flow with dialogs \(.NET\)](#)
- [Manage conversation flow with dialogs \(Node.js\)](#)

Dialog stack

When one dialog invokes another, the Bot Builder adds the new dialog to the top of the dialog stack. The dialog

that is on top of the stack is in control of the conversation. Every new message sent by the user will be subject to processing by that dialog until it either closes or redirects to another dialog. When a dialog closes, it's removed from the stack, and the previous dialog in the stack assumes control of the conversation.

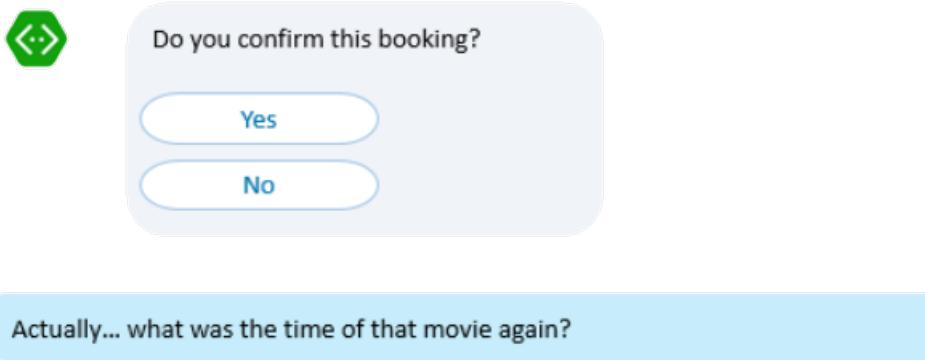
IMPORTANT

Understanding the concept of how the dialog stack is constructed and deconstructed by the Bot Builder as dialogs invoke one another, close, and so on is critical to being able to effectively design the conversation flow of a bot.

Dialogs, stacks and humans

It may be tempting to assume that users will navigate across dialogs, creating a dialog stack, and at some point will navigate back in the direction they came from, unstacking the dialogs one by one in a neat and orderly way. For example, the user will start at root dialog, invoke the new order dialog from there, and then invoke the product search dialog. Then the user will select a product and confirm, exiting the product search dialog, complete the order, exiting the new order dialog, and arrive back at the root dialog.

Although it would be great if users always traveled such a linear, logical path, it seldom occurs. Humans do not communicate in "stacks." They tend to frequently change their minds. Consider the following example:



While your bot may have logically constructed a stack of dialogs, the user may decide to do something entirely different or ask a question that may be unrelated to the current topic. In the example, the user asks a question rather than providing the yes/no response that the dialog expects. How should your dialog respond?

- Insist that the user answer the question first.
- Disregard everything that the user had done previously, reset the whole dialog stack, and start from the beginning by attempting to answer the user's question.
- Attempt to answer the user's question and then return to that yes/no question and try to resume from there.

There is no *right* answer to this question, as the best solution will depend upon the specifics of your scenario and how the user would reasonably expect the bot to respond.

Next steps

Managing the user's navigation across dialogs and designing conversation flow in a manner that enables users to achieve their goals (even in a non-linear fashion) is a fundamental challenge of bot design. The [next article](#) reviews some common pitfalls of poorly designed navigation and discusses strategies for avoiding those traps.

Design bot navigation

8/7/2017 • 4 min to read • [Edit Online](#)

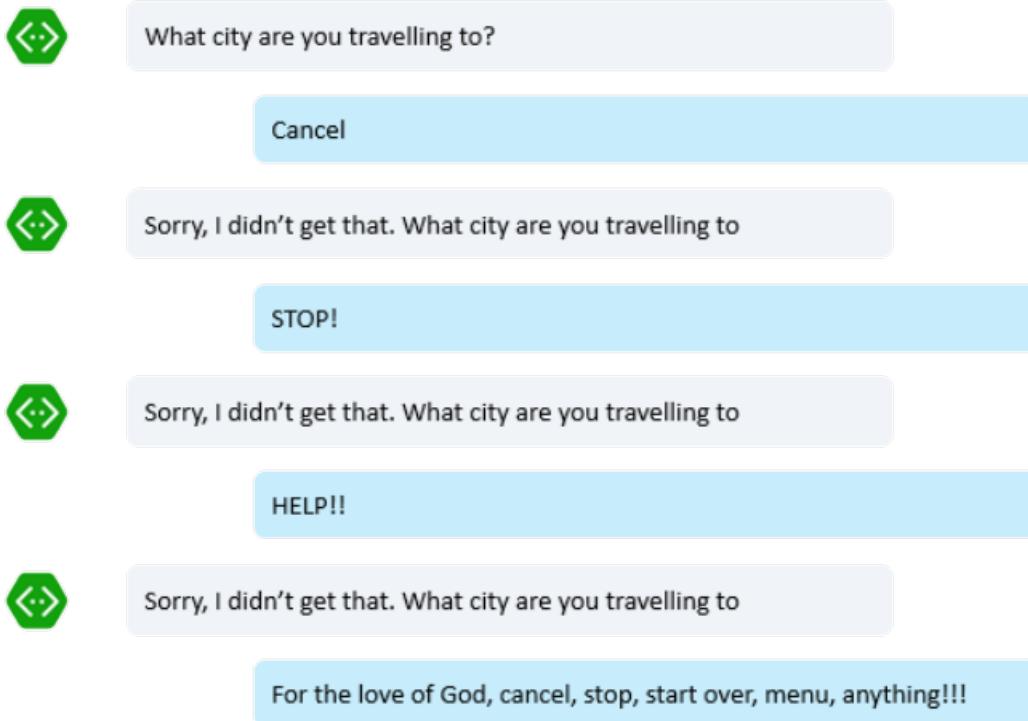
Users can navigate websites using breadcrumbs, apps using menus, and web browsers using buttons like **forward** and **back**. However, none of these well-established navigation techniques entirely address navigation requirements within a bot. As discussed [previously](#), users often interact with bots in a non-linear fashion, making it challenging to design bot navigation that consistently delivers a great user experience. Consider the following dilemmas:

- How do you ensure that a user doesn't get lost in a conversation with a bot?
- Can a user navigate "back" in a conversation with a bot?
- How does a user navigate to the "main menu" during a conversation with a bot?
- How does a user "cancel" an operation during a conversation with a bot?

The specifics of your bot's navigation design will depend largely upon the features and functionality that your bot supports. However, regardless of the type of bot you're developing, you'll want to avoid the common pitfalls of poorly designed conversational interfaces. This article describes these pitfalls in terms of five personalities: the "stubborn bot", the "clueless bot", the "mysterious bot", the "captain obvious bot", and the "bot that can't forget."

The "stubborn bot"

The stubborn bot insists upon maintaining the current course of conversation, even when the user attempts to steer things in a different direction. Consider the following scenario:



Users often change their minds. They can decide to cancel. Sometimes they want to start over altogether.

TIP

Do: Design your bot to consider that a user might attempt to change the course of the conversation at any time.

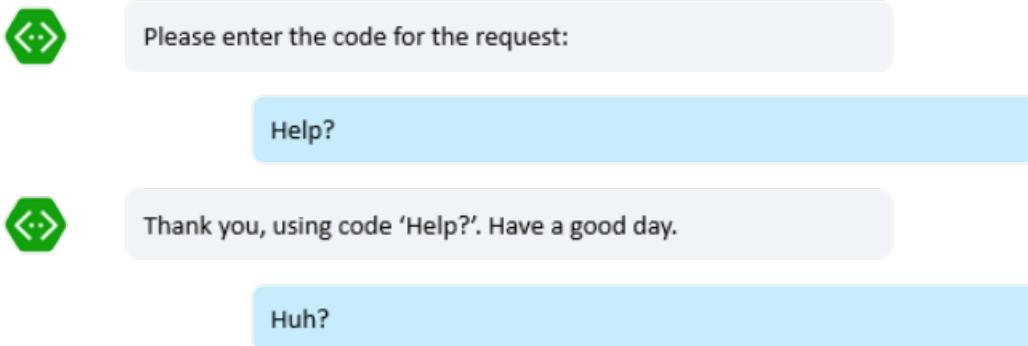
Don't Design your bot to ignore user input and keep repeating the same question in an endless loop.

There are many methods of avoiding this pitfall, but perhaps the easiest way to prevent a bot from asking the same question endlessly is to simply specify a maximum number of retry attempts for each question. If designed in this manner, the bot is not doing anything "smart" to understand the user's input and respond appropriately but will at least avoid asking the same question in an endless loop.

The "clueless bot"

The clueless bot responds in a nonsensical manner when it doesn't understand a user's attempt to access certain functionality. A user may try common keyword commands like "help" or "cancel" with reasonable expectations that the bot will respond appropriately.

Consider the following scenario:



Although you may be tempted to design every dialog within your bot to listen for, and respond appropriately to, certain keywords, this approach is not recommended.

TIP

Do: Implement global message handlers (using [.NET](#) or [Node.js](#)) that will examine user input for the keywords that you specify (ex: "help", "cancel", "start over", etc.) and respond appropriately.

Don't: Design every dialog to examine user input for a list of keywords.

By defining the logic in **global message handlers**, you're making it accessible to all dialogs. Using this approach, individual dialogs and prompts can be made to safely ignore the keywords, if necessary.

The "mysterious bot"

The mysterious bot fails to immediately acknowledge the user's input in any way. Consider the following scenario:



How can I help you today?

Which movies are showing this week?

Hello?

Which movies are showing this week?

Hey bot, are you there?

In some cases, this situation might be an indication that the bot is having an outage. However, it could just be that the bot is busy processing the user's input and hasn't yet finished compiling its response.

TIP

Do: Design your bot to immediately acknowledge user input, even in cases where the bot may take some time to compile its response.

Don't: Design your bot to postpone acknowledgement of user input until the bot finishes compiling its response.

By immediately acknowledging the user's input, you eliminate any potential for confusion as to the state of the bot.

The "captain obvious" bot

The captain obvious bot provides unsolicited information that is completely obvious and therefore useless to the user. Consider the following scenario:



You just spent \$10 on your credit card!

Yup, I know...



It looks like you are driving to work!

Really...



It looks like you arrived at work! Have a nice day!

Jeez, get a life, bot...

TIP

Do: Design your bot to provide information that will be useful to the user.

Don't: Design your bot to provide unsolicited information that is unlikely to be useful to the user.

By designing your bot to provide useful information, you're increasing the odds that the user will engage with your

bot.

The "bot that can't forget"

The bot that can't forget inappropriately integrates information from past conversations into the current conversation.

Consider the following scenario:



I want to travel to Italy

Please confirm: Are you ok with me charging \$200 for your trip to Las Vegas?

Which Trip to Las Vegas???



Your trip to Las Vegas you wanted me to book for June the 5th

OMG that was 3 months ago, bot...

TIP

Do: Design your bot to maintain the current topic of conversation, unless/until the user expresses a desire to revisit a prior topic.

Don't: Design your bot to interject information from past conversations when it is not relevant to the current conversation.

By maintaining the current topic of conversation, you reduce the potential for confusion and frustration and increase the odds that the user will continue to engage with your bot.

Next steps

By designing your bot to avoid these common pitfalls of poorly designed conversational interfaces, you're taking an important step toward ensuring a great user experience. Next, learn more about the [UX elements](#) that bots most typically rely upon to exchange information with users.

Design the user experience

8/7/2017 • 5 min to read • [Edit Online](#)

Bots typically use some combination of **rich user controls**, **text and natural language**, and **speech** to exchange information with users.

Rich user controls

Rich user controls are common UI controls such as buttons, images, carousels, and menus that the bot presents to the user and the user engages with to communicate choice and intent. A bot can use a collection of UI controls to mimic an app or can even run embedded within an app. When a bot is embedded within an app or website, it can represent virtually any UI control by leveraging the capabilities of the app that is hosting it.

For decades, application and website developers have relied on UI controls to enable users to interact with their applications. These same UI controls can also be very effective in bots. For example, buttons are a great way to present the user with a simple choice. Allowing the user to communicate "Hotels" by clicking a button labeled **Hotels** is easier and quicker than forcing the user to type "Hotels." This especially holds true on mobile devices, where clicking is greatly preferred over typing.

When designing your bot, do not automatically dismiss common UI elements as not being "smart enough." As discussed [previously](#), your bot should be designed to solve the user's problem in the best/quickest/easiest manner possible. Avoid the temptation to start by incorporating natural language understanding, as it is often unnecessary and just introduces unjustified complexity.

TIP

Start by using the minimum UI controls that enable the bot to solve the user's problem, and add other elements later if those controls are no longer sufficient.

Text and natural language understanding

A bot can accept **text** input from users and attempt to parse that input using regular expression matching or **natural language understanding** APIs such as [LUIS](#). There are many different types of text input that a bot might expect from a user. Depending on the type of input that the user provides, natural language understanding may or may not be a good solution.

In some cases, a user may be **answering a very specific question**. For example, if the bot asks, "What is your name?", the user may answer with text that specifies only the name, "John", or with a sentence, "My name is John". Asking specific questions reduces the scope of potential responses that the bot might reasonably receive, which decreases the complexity of the logic necessary to parse and understand the response. For example, consider the following broad, open-ended question: "How are you feeling?". Understanding the many possible permutations of potential answers to such a question is a very complex task. In contrast, specific questions such as "Are you feeling pain? yes/no" and "Where are you feeling pain? chest/head/arm/leg" would likely prompt more specific answers that a bot can parse and understand without needing to implement natural language understanding.

TIP

Whenever possible, ask specific questions that will not require natural language understanding capabilities to parse the response.

In other cases, a user may be **typing a specific command**. For example, a DevOps bot that enables developers to manage virtual machines could be designed to accept specific commands such as "/STOP VM XYZ" or "/START VM XYZ." Designing a bot to accept specific commands like this makes for a good user experience, as the syntax is easy to learn and the expected outcome of each command is clear. Additionally, the bot will not require natural language understanding capabilities, since the user's input can be easily parsed using regular expressions.

TIP

Designing a bot to require specific commands from the user can often provide a good user experience while also eliminating the need for natural language understanding capability.

In the case of a *knowledge base* bot or *questions and answers* bot, a user may be **asking general questions**. For example, imagine a bot that can answer questions based on the contents of thousands of documents. [QnA Maker](#) and [Azure Search](#) are both technologies which are designed specifically for this type of scenario. For more information, see [Design knowledge bots](#).

TIP

If you are designing a bot that will answer questions based on structured or unstructured data from databases, web pages, or documents, consider using technologies that are designed specifically to address this scenario rather than attempting to solve the problem with natural language understanding.

In other scenarios, a user may be **typing simple requests based on natural language**. For example, a user may type "I want a pepperoni pizza" or "Are there any vegetarian restaurants within 3 miles from my house open now?". Natural language understanding APIs such as [LUIS.ai](#) are a great fit for scenarios like this. Using the APIs, your bot can extract the key components of the user's text to identify the user's intent. When implementing natural language understanding capabilities in your bot, set realistic expectations for the level of detail that users are likely to provide in their input.

"I want to find a house for sale that has 3 or 4 bedrooms, priced between \$300 and \$350 with a large garden, about 2000 square feet, preferably green, within 10 miles from my work which is in the city center, with a large garage and a backyard with a pool"

-Said nobody, ever

"I want to find a house"

-Said everybody else

TIP

When building natural language models, do not assume that users will provide all the required information in their initial query. Design your bot to specifically request the information it requires, guiding the user to provide that information by asking a series of questions, if necessary.

Speech

A bot can use **speech** input and/or output to communicate with users. In cases where a bot is designed to support devices that have no keyboard or monitor, speech is the only means of communicating with the user.

Choosing between rich user controls, text and natural language, and speech

Just like people communicate with each other using a combination of gestures, voice, and symbols, bots can communicate with users using a combination of rich user controls, text (sometimes including natural language), and speech. You do not need to choose one over another. For example, imagine a "cooking bot" that helps users with recipes. The bot may provide instructions by playing a video or displaying a series of pictures to explain what needs to be done. Some users may prefer to flip pages of the recipe or ask the bot questions using speech while they are assembling a recipe. Others may prefer to touch the screen of a device instead of interacting with the bot via speech. When designing your bot, incorporate the UX elements that support the ways in which users will likely prefer to interact with your bot, given the specific use cases that it is intended support.

Create task automation bots

8/7/2017 • 3 min to read • [Edit Online](#)

A task automation bot enables the user to complete a specific task or set of tasks without any assistance from a human. This type of bot often closely resembles a typical app or website, communicating with the user primarily via rich user controls and text. It may have natural language understanding capabilities to enrich conversations with users.

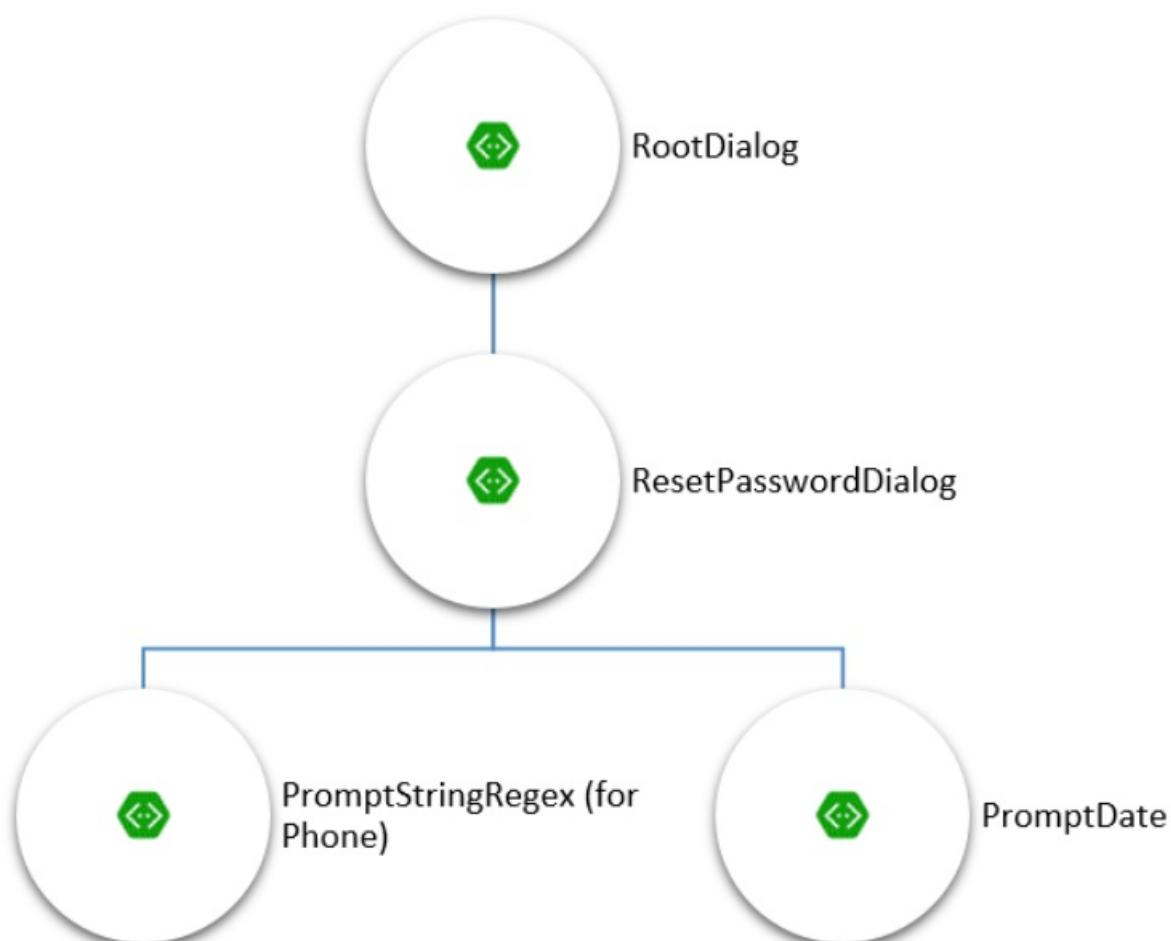
Example use case: password-reset

To better understand the nature of a task bot, consider an example use case: password-reset. The Contoso company receives several help desk calls each day from employees who need to reset their passwords. Contoso wants to automate the simple, repeatable task of resetting an employee's password so that help desk agents can devote their time to addressing more complex issues.

John, an experienced developer from Contoso, decides to create a bot to automate the password-reset task. He begins by writing a design specification for the bot, just as he would do if he were creating a new app or website.

Navigation model

The specification defines the navigation model:



The user begins at the `RootDialog`. When they request a password reset, they

will be directed to the `ResetPasswordDialog`. With the `ResetPasswordDialog`, the bot will prompt the user for two pieces of information: phone number and birth date.

IMPORTANT

The bot design described in this article is intended for example purposes only. In real-world scenarios, a password-reset bot would likely implement a more robust identity verification process.

Dialogs

Next, the specification describes the appearance and functionality of each dialog.

Root dialog

The root dialog provides the user with two options:

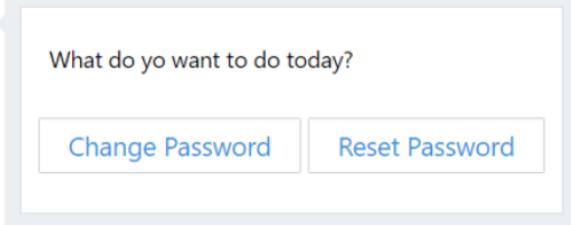
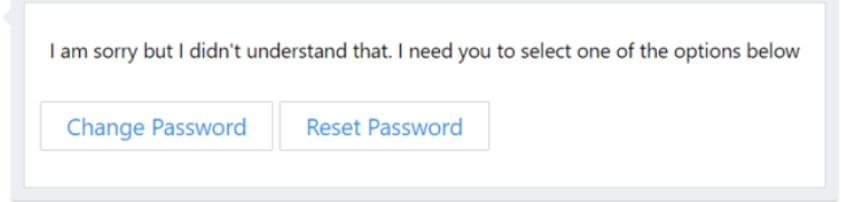
1. **Change Password** is for scenarios where the user knows their current password and simply wants to change it.
2. **Reset Password** is for scenarios where the user has forgotten or misplaced their password and needs to generate a new one.

NOTE

For simplicity, this article describes only the **reset password** flow.

The specification describes the root dialog as shown in the following screenshot.

4.1. RootDialog

Event	Description	Notes						
On Start:	Dialog initiates with the following message: 	On Start is what initially happens when the dialog is invoked						
User Input:	User will be allowed to click at one of the buttons above (or type one of those options on channels that don't support buttons): <table border="1" data-bbox="309 707 880 842"> <thead> <tr> <th>Action</th><th>Effect</th></tr> </thead> <tbody> <tr> <td>"Change Password"</td><td>Flow not implemented. Send a message to the user</td></tr> <tr> <td>"Reset Password"</td><td>Call to ResetPassword dialog</td></tr> </tbody> </table>	Action	Effect	"Change Password"	Flow not implemented. Send a message to the user	"Reset Password"	Call to ResetPassword dialog	
Action	Effect							
"Change Password"	Flow not implemented. Send a message to the user							
"Reset Password"	Call to ResetPassword dialog							
Invalid user input:	In case of an invalid user input: 	Assuming the user types something else and no scorable picks that up, we will reject the input and retry the question.						
Exit Criteria:	This is the root dialog and therefore it never exits.	Root dialogs shouldn't exit.						
		They are the very entry point where the whole experience starts.						

ResetPassword dialog

When the user chooses **Reset Password** from the root dialog, the `ResetPassword` dialog is invoked. The `ResetPassword` dialog then invokes two other dialogs. First, it invokes the `PromptStringRegex` dialog to collect the user's phone number. Then it invokes the `PromptDate` dialog to collect the user's date of birth.

NOTE

In this example, John chose to implement the logic for collecting the user's phone number and date of birth by using two separate dialogs. The approach not only simplifies the code required for each dialog, but also increases the odds of these dialogs being usable by other scenarios in the future.

The specification describes the `ResetPassword` dialog.

4.2. ResetPassword dialog

Event	Description	Notes
On Start:	<p>Dialog initiates calling the <code>PromptStringRegex</code> dialog to ask the user for a phone number.</p> <p>Once that dialog completes, if the phone number was provided, it will be shown:</p> <p>The phone you provided is: (425) 123-4235</p> <p>And then the <code>PromptDate</code> dialog will be called to ask the user for the date of birth.</p>	
User Input:	No user input is provided in this dialog	
Invalid user input:	No user input is provided in this dialog	
Exit Criteria:	<p>If the phone number and the date of birth are correctly provided, the dialog will provide a new password and ends with a “true” value to indicate success.</p> <p>Thanks! Your new password is <i>b3667b6b34ec4101bfe02e269fa8485c</i></p> <p>If the phone number or date of birth are not correctly provided or any of those prompts is cancelled, the dialog will end with a “false” value to indicate that it failed. RootDialog will detect that and will show:</p> <p>You identity was not verified and your password cannot be reset</p>	

`PromptStringRegex` dialog

The `PromptStringRegex` dialog prompts the user to enter their phone number, and verifies that the phone number that the user provides matches the expected format. It also accounts for the scenario where the user repeatedly provides invalid input. The spec describes the `PromptStringRegex` dialog.

4.3. PromptStringRegex dialog

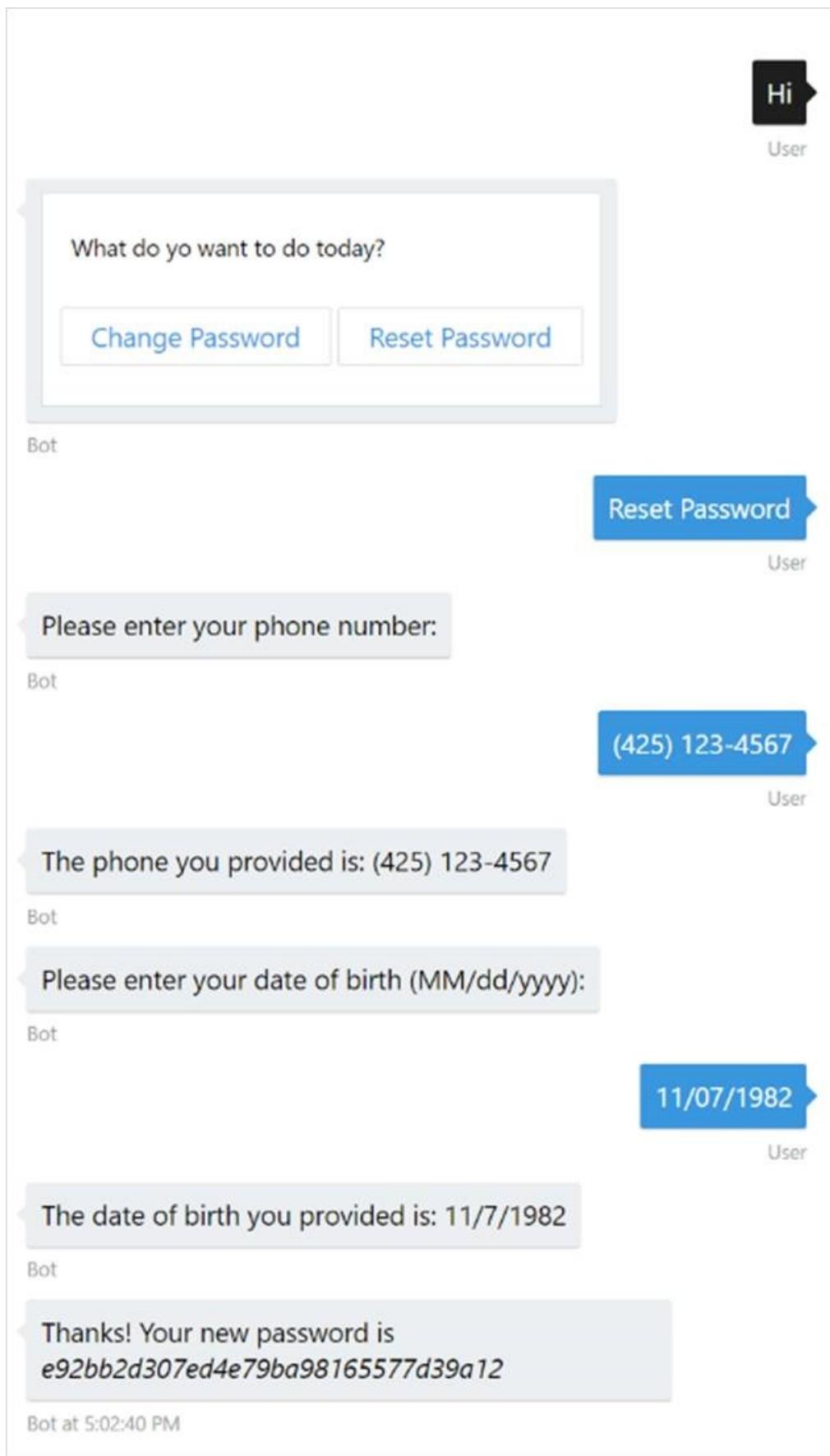
Event	Description	Notes		
On Start:	Dialog initiates with the following question: Please enter your phone number:	Since the options are too many, we will use free text input here. A custom PromptString dialog with Regex validation is being used		
User Input:	User will be allowed to type a phone number	A regular expression is being used to validate the input		
	<table border="1"> <thead> <tr> <th>Action</th><th>Effect</th></tr> </thead> <tbody> <tr> <td>Any input</td><td>Run a phone number regex validation</td></tr> </tbody> </table>		Action	Effect
Action	Effect			
Any input	Run a phone number regex validation			
Invalid user input:	In case on an invalid phone number: The value entered is not phone number. Please try again using the following format (xyz) xyz-wxyz:			
Exit Criteria:	To be defined in the alternative flows below			

Alternative Flow 1

Condition	An invalid phone was provided many times	
Effect	PromptStringRegex dialog ends with the following message: You have tried to enter your phone number many times. Please try again later.	
Exit Criteria:	PromptStringRegex throws a TooManyAttemptsException	No phone number is provided

Prototype

Finally, the spec provides an example of a user communicating with the bot to successfully complete the password-reset task.



Bot, app, or website?

You may be wondering, if a task automation bot closely resembles an app or website, why not just build an app or website instead? Depending on your particular scenario, building an app or website instead of a bot may be an entirely reasonable choice. You may even choose to embed your bot into an app, by using the [Bot Framework Direct Line API](#) or [Web Chat control](#). Implementing your bot within the context of an app provides the best of both worlds: a rich app experience and a conversational experience, all in one place.

In many cases, however, building an app or website can be significantly more complex and more expensive than building a bot. An app or website often needs to support multiple clients and platforms, packaging and deploying can be tedious and time-consuming processes, and the user experience of having to download and install an app is not necessarily ideal. For these reasons, a bot may often provide a much simpler way of solving the problem at hand.

Additionally, bots provide the freedom to easily expand and extend. For example, a developer may choose to add natural language and speech capabilities to the password-reset bot so that it can be accessed via audio call, or she may add support for text messages. The company may setup kiosks throughout the building and embed the password-reset bot into that experience.

Sample code

For a complete sample that shows how to implement simple task automation using the Bot Builder SDK for .NET, see the [Simple Task Automation sample](#) in GitHub.

For a complete sample that shows how to implement simple task automation using the Bot Builder SDK for Node.js, see the [Simple Task Automation sample](#) in GitHub.

Additional resources

- [Dialogs](#)
- [Manage conversation flow with dialogs \(.NET\)](#)
- [Manage conversation flow with dialogs \(Node.js\)](#)

Design knowledge bots

8/7/2017 • 8 min to read • [Edit Online](#)

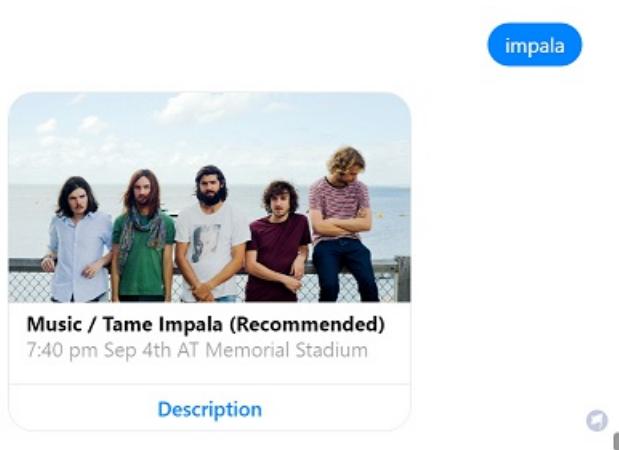
A knowledge bot can be designed to provide information about virtually any topic. For example, one knowledge bot might answer questions about events such as, "What bot events are there at this conference?", "When is the next Reggae show?", or "Who is Tame Impala?" Another might answer IT-related questions such as "How do I update my operating system?" or "Where do I go to reset my password?" Yet another might answer questions about contacts such as "Who is John Doe?" or "What is Jane Doe's email address?"

Regardless of the use case for which a knowledge bot is designed, its basic objective is always the same: find and return the information that the user has requested by leveraging a body of data, such as relational data in a SQL database, JSON data in a non-relational store, or PDFs in a document store.

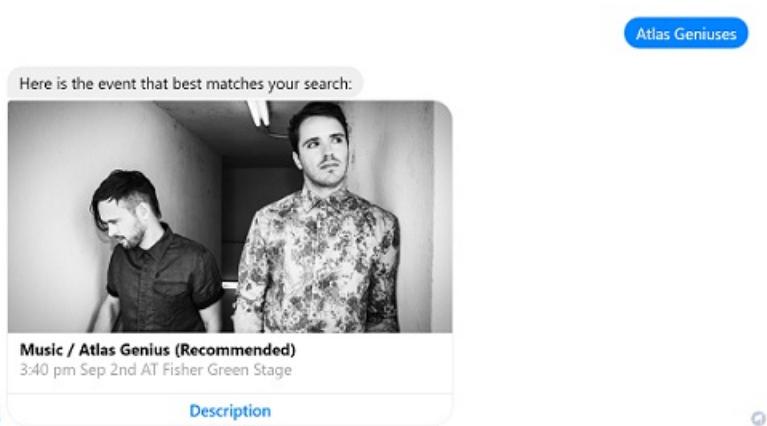
Search

Search functionality can be a valuable tool within a bot.

First, "fuzzy search" enables a bot to return information that's likely to be relevant to the user's question, without requiring that the user provide precise input. For example, if the user asks a music knowledge bot for information about "impala" (instead of "Tame Impala"), the bot can respond with information that's most likely to be relevant to that input.



Search scores indicate the level of confidence for the results of a specific search, enabling a bot to order its results accordingly, or even tailor its communication based upon confidence level. For example, if confidence level is high, the bot may respond with "Here is the event that best matches your search:".



If confidence level is low, the bot may respond with "Hmm... were you looking for any of these events?"

super smart atlas

Hmm... were you looking for any of these events?

Using Search to Guide a Conversation

If your motivation for building a bot is to enable basic search engine functionality, then you may not need a bot at all. What does a conversational interface offer that users can't get from a typical search engine in a web browser?

Knowledge bots are generally most effective when they are designed to guide the conversation. A conversation is composed of a back-and-forth exchange between user and bot, which presents the bot with opportunities to ask clarifying questions, present options, and validate outcomes in a way that a basic search is incapable of doing. For example, the following bot guides a user through a conversation that facets and filters a dataset until it locates the information that the user is seeking.

Events

What are you interested in?

Music Comedy Film Laser Dome >

What Music are you interested in?

Any Rock/Pop Hiphop/Rap Soul/R&B >

Rock/Pop

What day would you like to see Rock/Pop?

Friday Saturday Sunday Any

Here were the Rock/Pop shows Saturday:



Music / Pony Time (Recommended)

3:00 pm Sep 3rd AT KEXP



Music / Desi Valentine

3:30 pm Sep 3rd AT Starbucks Stage

By processing the user's input in each step and presenting the relevant options, the bot guides the user to the information that they're seeking. Once the bot delivers that information, it can even provide guidance about more efficient ways to find similar information in the future.

(By the way - you can also just type "Rock friday" or search an event by name)

Type a message...



Azure Search

By using [Azure Search](#), you can create an efficient search index that a bot can easily search, facet, and filter. Consider a search index that is created using the Azure portal.

We provided a default index for you. Right-click to delete the fields you don't need. Everything is editable, but once the index is built, deleting or changing existing fields will require re-indexing your documents.

FIELD NAME	TYPE	RETRIEVABLE	FILTERABLE	SORTABLE	FACETABLE	SEARCHABLE
imageURL	Edm.String	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Name	Edm.String	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>
Era	Edm.String	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>
Description	Edm.String	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
id	Edm.String	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
rid	Edm.String	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
	Edm.String	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>

You want to be able to access all properties of the data store, so you set each property as "retrievable." You want to be able to find musicians by name, so you set the **Name** property as "searchable." Finally, you want to be able to facet filter over musicians' eras, so you mark the **Eras** property as both "facetable" and "filterable."

Faceting determines the values that exist in the data store for a given property, along with the magnitude of each value. For example, this screenshot shows that there are 5 distinct eras in the data store:

Which era of music are you interested in?



Romantic (11)

Classical (3)

Baroque (2)

Impressionist (2)

Modernist (1)

Filtering, in turn, selects only the specified instances of a certain property. For example, you could filter the result set above to contain only items where **Era** is equal to "Romantic."

NOTE

See [a sample bot](#) for a complete example of a knowledge bot that is created using Azure Document DB, Azure Search, and the Microsoft Bot Framework.

For the sake of simplicity, the example above shows a search index that is created using the Azure portal. Indices can also be created programmatically.

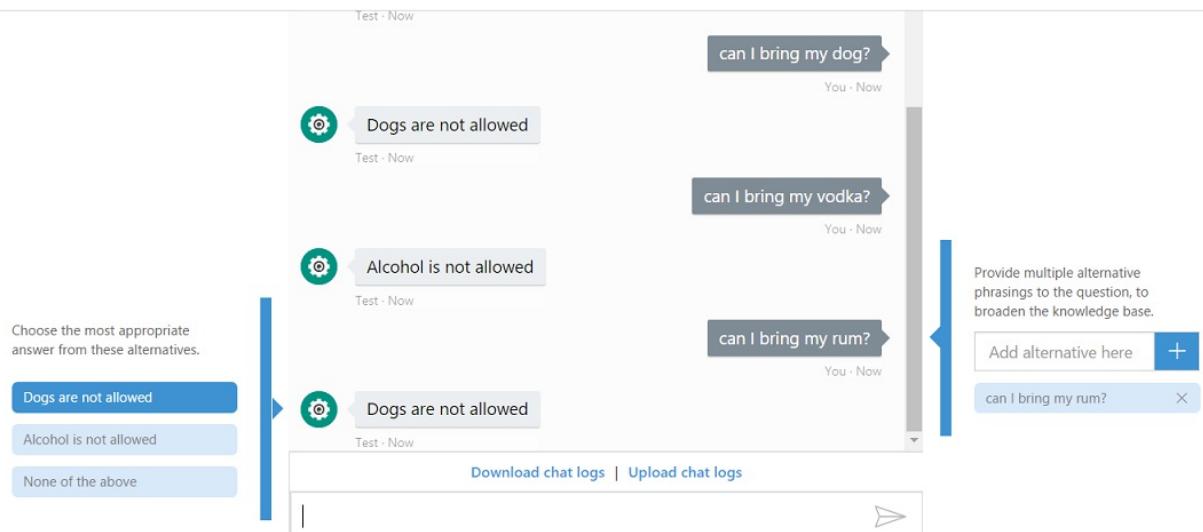
QnA Maker

Some knowledge bots may simply aim to answer frequently asked questions (FAQs). [QnA Maker](#) is a powerful tool that's designed specifically for this use case. QnA Maker has the built-in ability to scrape questions and answers from an existing FAQ site. It also allows you to manually configure your own custom list of questions and answers. QnA Maker has natural language processing (NLP) abilities, enabling it to even provide answers to questions that are worded slightly differently than expected. However, it does not have semantic language understanding abilities. It cannot determine that a puppy is a type of dog, for example.

Using the QnA Maker web interface, you can configure a knowledge base with three question and answer pairs:

KNOWLEDGE BASE 3 QnA pairs	
Question	Answer
^ Original source: Editorial	
1 hi	hello
2 can I bring my vodka?	Alcohol is not allowed
3 can I bring my dog?	Dogs are not allowed

Then, you can test it by asking a series of questions:



The bot correctly answers the questions that directly map to the ones that were configured in the knowledge base. However, it incorrectly responds to the question "can I bring my rum?". Because this question is most similar in structure to the question "can I bring my dog?" and because QnA Maker does not inherently understand the meaning of words, i.e., it does not know that "rum" is a type of liquor, it answers "Dogs are not allowed."

TIP

Create your QnA pairs and then test and re-train your bot by using the menu on the left side of the conversation to select an alternative answer for each incorrect answer that is given.

LUIS

Some knowledge bots require natural language processing (NLP) capabilities so that they can analyze a user's messages to determine the user's intent. Language Understanding Intelligent Service (LUIS) provides a fast and effective means of adding NLP capabilities to bots. LUIS enables you to use existing, pre-built models from Bing and Cortana whenever they meet your needs. It also allows you to create specialized models of your own.

When working with huge datasets, it's not necessarily feasible to train an NLP model with every variation of an entity. In a music playing bot, for example, a user might message "Play Reggae", "Play Bob Marley", or "Play One Love". Although a bot could map each of these messages to the intent "playMusic", without being trained with every artist, genre and song name, an NLP model would not be able to identify whether the entity is a genre, artist or song. By using an NLP model to identify the generic entity of type "music", the bot could search its data store for that entity, and proceed from there.

Combining Search, QnA Maker, and/or LUIS

Search, QnA Maker and LUIS are each powerful tools in their own right, but they can also be combined to build knowledge bots that possess more than one of those capabilities.

LUIS and Search

In the music festival bot example [covered earlier](#), the bot guides the conversation by showing buttons that represent the lineup. However, this bot could also incorporate natural language understanding by using LUIS to determine intent and entities within questions such as "what kind of music does Romit Girdhar play?". Then the bot could search against an Azure Search index using musician name.

It would not be feasible to train the model with every possible musician name since there are so many potential values, but you could provide enough representative examples for LUIS to properly identify the entity at hand. For example, consider that you train your model by providing examples of musicians:

what kind of music does **romit girdhar** play ?

answerGenre ▾

Submit

what genre of music is **foxygen** ?

answerGenre ▾

Submit

When you test this model with new utterances like, "what kind of music do the beatles play?", LUIS successfully determines the intent "answerGenre" and the identifies entity "the beatles." However, if you submit a longer question such as "what kind of music does the devil makes three play?", LUIS identifies "the devil" as the entity.

```
"entities": [  
  {  
    "entity": "the devil",  
    "type": "musician",  
    "startIndex": 25,  
    "endIndex": 33,  
    "score": 0.300864249  
  }  
]
```

By training the model with example entities that are representative of the underlying dataset, you can increase the accuracy of your bot's language understanding.

TIP

In general, it is better for the model to err by identifying excess words in its entity recognition, e.g., identify "John Smith please" from the utterance "Call John Smith please", rather than identify too few words, e.g., identify "John" from the utterance "Call John Smith please". The search index will ignore irrelevant words such as "please" in the phrase "John Smith please".

Luis and QnA Maker

Some knowledge bots might use QnA Maker to answer basic questions in combination with LUIS to determine intents, extract entities and invoke more elaborate dialogs. For example, consider a simple IT Help Desk bot. This bot may use QnA Maker to answer basic questions about Windows or Outlook, but it might also need to facilitate scenarios like password reset, which require intent recognition and back-and-forth communication between user and bot. There are a few ways that a bot may implement a hybrid of LUIS and QnA Maker:

1. Call both QnA Maker and LUIS at the same time, and respond to the user by using information from the first one that returns a score of a specific threshold.
2. Call LUIS first, and if no intent meets a specific threshold score, i.e., "None" intent is triggered, then call QnA Maker. Alternatively, create a LUIS intent for QnA Maker, feeding your LUIS model with example QnA questions that map to "QnAIntent."
3. Call QnA Maker first, and if no answer meets a specific threshold score, then call LUIS.

The Bot Builder SDK for Node.js and the Bot Builder SDK for C# provide built-in support for LUIS and QnA Maker. This enables you to trigger dialogs or automatically answer questions using LUIS and/or QnA Maker without having to implement custom calls to either tool. For example, if LUIS has enabled you to determine intent, you could trigger a [BasicQnAMakerDialog](#) to initiate the process of answering the user's question.

TIP

When implementing a combination of LUIS, QnA Maker, and/or Azure Search, test inputs with each of the tools to determine the threshold score for each of your models. LUIS, QnA Maker, and Azure Search each generate scores by using a different scoring criteria, so the scores generated across these tools are not directly comparable. Additionally, LUIS and QnA Maker normalize scores. A certain score may be considered 'good' in one LUIS model but not so in another model.

Sample code

- For a sample that shows how to create a basic knowledge bot using the Bot Builder SDK for .NET, see the [Knowledge Bot sample](#) in GitHub.
- For a sample that shows how to create more complex knowledge bots using the Bot Builder SDK for .NET, see the [Search-powered Bots sample](#) in GitHub.

Additional resources

- [Add intelligence to bots with Cognitive Services](#)

Integrate your bot with a web browser

6/13/2017 • 5 min to read • [Edit Online](#)

Some scenarios require more than just a bot to fulfill a requirement. A bot may need to send the user to a web browser to complete a task and then resume the conversation with the user after the task has been completed.

Authentication and authorization

If a bot wants the ability to read the user's calendar in Office 365, or perhaps even create appointments on behalf of that user, the user must first authenticate with Microsoft Azure Active Directory and authorize the bot to access the user's calendar data. The bot will redirect the user to a web browser to complete the authentication and authorization tasks, and then will subsequently resume the conversation with the user.

Security and compliance

Security and compliance requirements often restrict the type of information that a bot can exchange with a user. In some cases, it may be necessary for the user to send/receive data outside of the current conversation. For example, if a user wants to execute a payment using a third-party payment provider, a credit card number should not be specified within the context of the conversation. Instead, the bot will direct the user to a web browser to complete the payment process, and then will subsequently resume the conversation with the user.

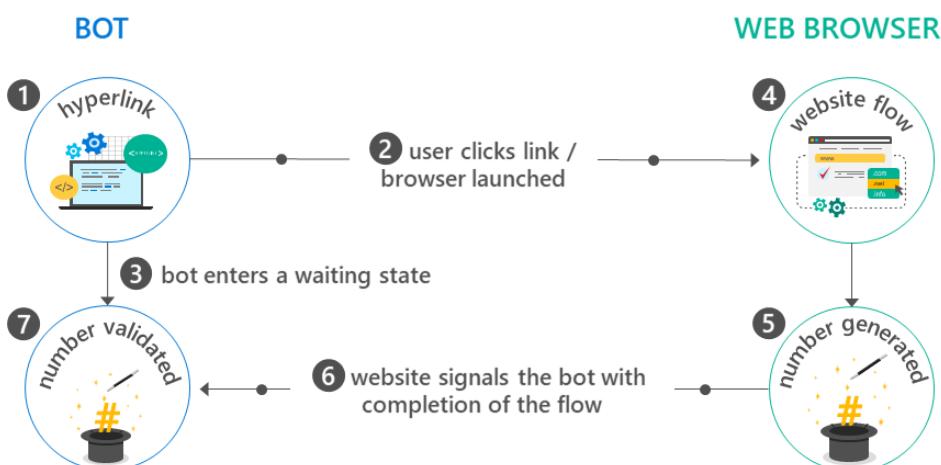
This article explores the process of facilitating a user's transition from bot to web browser, and back again.

NOTE

Transitioning from chat to web browser and back is not ideal, as switching between applications can easily confuse a user. To provide a better experience, many channels offer built-in HTML windows that a bot can use to present applications would otherwise appear in a web browser. This technique allows the user to remain within the conversation while still accessing external resources. This approach is conceptually similar to mobile applications managing authorization flows using OAuth within embedded web views.

Bot to web browser, and back again

This diagram shows the high-level flow for integration between bot and web browser.



Consider each step of the flow:

1. The bot generates and displays a hyperlink that will redirect the user to a website. The hyperlink typically includes data via querystring parameters on the target URL that specify information about the context of the current conversation, such as conversation ID, channel ID, and user ID in the channel.
2. The user clicks the hyperlink and is redirected to the target URL within a web browser.
3. The bot enters a state awaiting communication from the website to indicate that the website flow is complete.

TIP

Design this flow so that the bot will not permanently remain in the 'waiting' state if the user never completes the website flow. In other words, if the user abandons the web browser and starts communicating with the bot again, the bot should acknowledge, not [ignore](#) that input.

4. The user completes the necessary task(s) via the web browser. This could be an OAuth flow or any sequence of events required by the scenario at hand.
5. When the user completes the website flow, the website generates a '[magic number](#)' and instructs the user to copy the value and paste it back into the conversation with the bot.
6. The website [signals to the bot](#) that the user has completed the website flow. It communicates the 'magic number' to the bot and provides any other relevant data. For example, in the case of an OAuth flow, the website would provide an access token to the bot.
7. The user returns to the bot and pastes the 'magic number' into the chat. The bot validates that 'magic number' provided by the user matches the expected value, verifying that the current user is the same user who previously clicked the hyperlink to initiate the website flow.

Verifying user identity using the 'magic number'

The generation of a 'magic number' during the bot-to-website flow ([step 5](#) above) enables the bot to subsequently verify that the user who initiated the website flow is indeed the user for whom it was intended. For example, if a bot is conducting a group chat with multiple users, any one of them could have clicked the hyperlink to initiate the website flow. Without the 'magic number' validation process, the bot has no way of knowing which user completed the flow. One user could authenticate and inject access tokens in another user's session.

WARNING

This isn't just a risk within group chats. Without the 'magic number' validation process, anyone who obtains the hyperlink to launch the website flow can spoof a user's identity.

The magic number should be a random number generated using a strong cryptography library. For an example of the generation process in C#, see [this code](#) within the [AuthBot](#) library. AuthBot enables bots that are built in Microsoft Bot Framework to implement the bot-to-website flow to authenticate a user in a website and then to subsequently use the access token that was generated from the authentication process. Since AuthBot does not make any assumptions about the channel's capabilities, such flows should function well with most channels.

NOTE

The need for the 'magic number' validation process should be deprecated as channels build their own embedded web views.

How does the website 'signal' the bot?

When the bot [generates the hyperlink](#) that the user will click to initiate the website flow, it includes information via querystring parameters in the target URL about the context of the current conversation, such as conversation ID, channel ID, and user ID in the channel. The website can subsequently use this information to read and write state variables for that user or conversation using the Bot Builder SDK or REST APIs. See [step 6](#) above for an example of how the website 'signals' to the bot that the website flow is complete.

Sample code

As described in this article, the [AuthBot](#) library enables OAuth flows to be bound to bots that are built using .NET in the Microsoft Bot Framework. The [BotAuth](#) library enables OAuth flows to be bound to bots that are built using Node.js in the Microsoft Bot Framework.

Additional resources

- [Dialogs](#)
- [Manage conversation flow with dialogs \(.NET\)](#)
- [Manage conversation flow with dialogs \(Node.js\)](#)
- [Add intelligence to bots with Cognitive Services](#)

Transition conversations from bot to human

8/7/2017 • 5 min to read • [Edit Online](#)

Regardless of how much artificial intelligence a bot possesses, there may still be times when it needs to hand off the conversation to a human being. The bot should recognize when it needs to hand off and provide the user with a clear, smooth transition.

Scenarios that require human involvement

A wide variety of scenarios may require that a bot transition control of the conversation to a human. A few of those scenarios are *triage*, *escalation*, and *supervision*.

Triage

A typical help desk call starts with some very basic questions that can easily be answered by a bot. As the first responder to inbound requests from users, a bot could collect the user's name, address, and description of the problem and then transition control of the conversation to an agent. Using a bot to triage incoming requests allows agents to devote their time to solving the problem instead of collecting information.

Escalation

In the help desk scenario, a bot may be able to answer basic questions and resolve simple issues in addition to collecting information. For example, bots are commonly used to reset a user's password. However, if a conversation indicates that a user's issue is complex enough to require human involvement, the bot will need to escalate the issue to a human agent. To implement this type of scenario, a bot must be capable of differentiating between issues it can resolve independently and issues that must be escalated to a human. There are many ways that a bot may determine that it needs to transfer control of the conversation to a human, including:

User-driven menus

Perhaps the simplest way for a bot to handle this dilemma is to present the user with a menu of options. Tasks that the bot can handle independently appear in the menu above a link labeled "Chat with an agent." This type of implementation requires no advanced machine learning or natural language understanding. The bot simply transfers control of the conversation to a human agent when the user selects the "Chat with an agent" option.

Scenario-driven

The bot may decide whether or not to transfer control based upon whether or not it determines that it is capable of handling the scenario at hand. The bot collects some information about the user's request and then queries its internal list of capabilities to determine if it is capable of addressing that request. If the bot determines that it is capable of addressing the request, it does so. However, if the bot determines that the request is beyond the scope of issues it can resolve, it transfers control of the conversation to a human agent.

Natural language

Natural language understanding and sentiment analysis help the bot decide when to transfer control of the conversation to a human agent. This is particularly valuable when attempting to determine when the user is frustrated or wants to speak with a human agent.

The bot analyzes the content of the user's messages by using the [Text Analytics API](#) to infer sentiment or by using the [LUIS API](#).

TIP

Natural language understanding may not always be the best method for determining when a bot should transfer conversation control to a human being. Bots, like humans, don't always guess correctly, and invalid responses will frustrate the user. If the user selects from a menu of valid choices, however, the bot will always respond appropriately to that input.

Supervision

In some cases, the human agent will want to monitor the conversation instead of taking control.

For example, consider a help desk scenario where a bot is communicating with a user to diagnose computer problems. A machine learning model helps the bot determine the most likely cause of the issue. Before advising the user to take a specific course of action, the bot can privately confirm the diagnosis and remedy with the human agent and request authorization to proceed. The agent clicks a button, the bot presents the solution to the user, and the problem is solved. The bot is still performing the majority of the work, but the agent retains control the final decision.

Transitioning control of the conversation

When a bot decides to transfer control of a conversation to a human, it can inform the user that she is being transferred and put the conversation into a 'waiting' state until it confirms that an agent is available.

When the bot is waiting for a human, it may automatically answer all incoming user messages with a default response such as "waiting in queue". Furthermore, you could have the bot remove the conversation from the 'waiting' state if the user sent certain messages such as "never mind" or "cancel".

You specify how agents will be assigned to waiting users when you design your bot. For example, the bot may implement a simple queue system: first in, first out. More complex logic would assign users to agents based upon geography, language, or some other factor. The bot could also present some type of UI to the agent that they can use to select a user. When an agent becomes available, she connects to the bot and joins the conversation.

IMPORTANT

Even after an agent is engaged, the bot remains the behind-the-scenes facilitator of the conversation. The user and agent never communicate directly with each other; they just route messages through the bot.

Routing messages between user and agent

After the agent connects to the bot, the bot begins to route messages between user and agent. Although it may appear to the user and the agent that they are chatting directly with each other, they are exchanging messages via the bot. The bot receives messages from the user and sends those messages to the agent. Likewise, it receives messages from the agent and sends those messages to the user.

NOTE

In more advanced scenarios, the bot can assume responsibility beyond merely routing messages between user and agent. For example, the bot may decide which response is appropriate and simply ask the agent for confirmation to proceed.

Sample code

For a complete sample that shows how to hand off conversations from bot to human using the Bot Builder SDK for Node.js, see the [Bot-HandOff sample](#) in GitHub.

Additional resources

- [Dialogs](#)
- [Manage conversation flow with dialogs \(.NET\)](#)
- [Manage conversation flow with dialogs \(Node.js\)](#)
- [Add intelligence to bots with Cognitive Services](#)

Embed a bot in an app

8/7/2017 • 2 min to read • [Edit Online](#)

Although bots most commonly exist outside of apps, they can also be integrated with apps. For example, you could embed a [knowledge bot](#) within an app to help users find information that might otherwise be challenging to locate within complex app structures. You could embed a bot within a help desk app to act as the first responder to incoming user requests. The bot could independently resolve simple issues and [hand off](#) more complex issues to a human agent.

Integrating bot with app

The way to integrate a bot with an app varies depending on the type of app.

Native mobile app

An app that is created in native code can communicate with the Bot Framework by using the [Direct Line API](#), either via REST or websockets.

Web-based mobile app

A mobile app that is built by using web language and frameworks such as [Cordova](#) may communicate with the Bot Framework by using the same components that a [bot embedded within a website](#) would use, just encapsulated within a native app's shell.

IoT app

An IoT app can communicate with the Bot Framework by using the [Direct Line API](#). In some scenarios, it may also use [Microsoft Cognitive Services](#) to enable capabilities such as image recognition and speech.

Other types of apps and games

Other types of apps and games can communicate with the Bot Framework by using the [Direct Line API](#).

Creating a cross-platform mobile app that runs a bot

This example of creating a mobile app that runs a bot uses [Xamarin](#), a popular tool for building cross-platform mobile applications.

First, create a simple web view component and use it to host a [web chat control](#). Then, using the Bot Framework Portal, [connect the bot](#) to the Web Chat channel.

Channels

[Refresh](#)

	Test link	Status	Published	
	Web Chat	<input checked="" type="checkbox"/> Off	Edit	

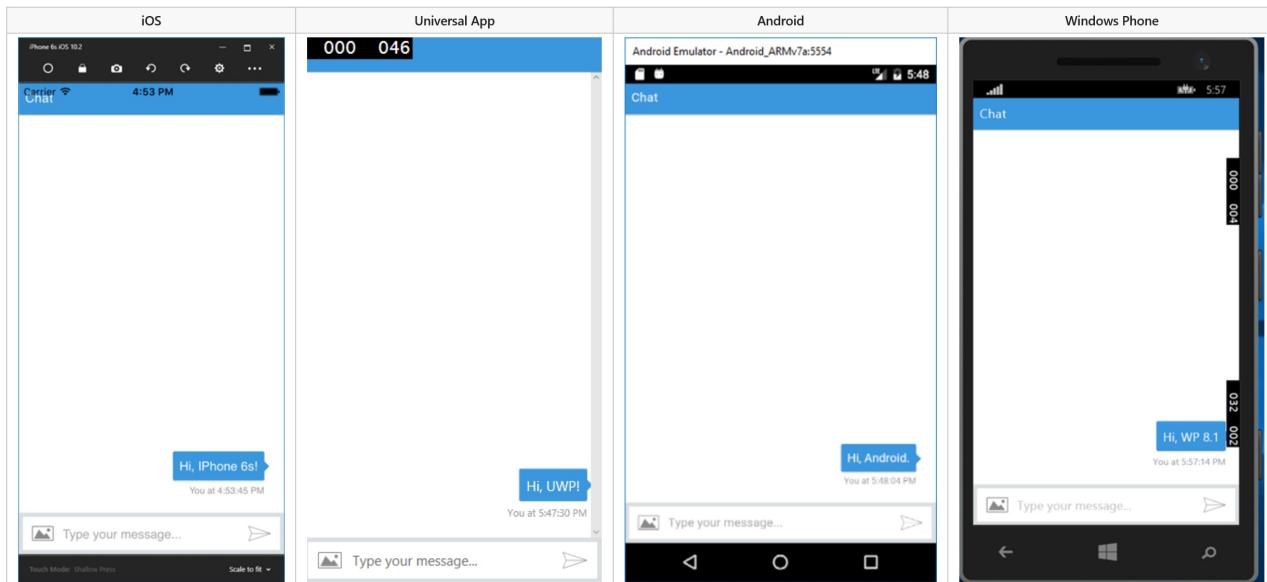
Add another channel

	Direct Line	Add
	Email	Add
	GroupMe	Add
	Skype	Add
	Slack	Add
	SMS	Add
	Telegram	Add

Next, specify the registered web chat URL as the source for the web view control in the Xamarin app:

```
public class WebPage : ContentPage
{
    public WebPage()
    {
        var browser = new WebView();
        browser.Source = "https://webchat.botframework.com/embed/<YOUR SECRET KEY HERE>";
        this.Content = browser;
    }
}
```

Using this process, you can create a cross-platform mobile application that renders the embedded web view with the web chat control.



Sample code

For a complete sample that shows how to create a cross-platform mobile app that runs a bot (as described in this article), see the [Bot in Apps sample](#) in GitHub.

Additional resources

- [Direct Line API](#)
- [Microsoft Cognitive Services](#)
- [Add intelligence to bots with Cognitive Services](#)

Embed a bot in a website

8/7/2017 • 2 min to read • [Edit Online](#)

Although bots commonly exist outside of websites, they can also be embedded within a website. For example, you may embed a [knowledge bot](#) within a website to enable users to quickly find information that might otherwise be challenging to locate within complex website structures. Or you might embed a bot within a help desk website to act as the first responder to incoming user requests. The bot could independently resolve simple issues and [handoff](#) more complex issues to a human agent.

This article explores integrating bots with websites and the process of using the *backchannel* mechanism to facilitate private communication between a web page and a bot.

Microsoft provides two different ways to integrate a bot in a website: the Skype web control and an open source web control.

Skype web control

The Skype web control is essentially a Skype client in a web-enabled control. Built-in Skype authentication enables the bot to authenticate and recognize users, without requiring the developer to write any custom code. Skype will automatically recognize Microsoft Accounts used in its web client.

Because the Skype web control simply acts as a front-end for Skype, the user's Skype client automatically has access to the full context of any conversation that the web control facilitates. Even after the web browser is closed, the user may continue to interact with the bot using the Skype client.

Open source web control

The [open source web chat control](#) is based upon ReactJS and uses the [Direct Line API](#) to communicate with the Bot Framework. The web chat control provides a blank canvas for implementing the web chat, giving you full control over its behaviors and the user experience that it delivers.

The *backchannel* mechanism enables the web page that is hosting the control to communicate directly with the bot in a manner that is entirely invisible to the user. This capability enables a number of useful scenarios:

- The web page can send relevant data to the bot (e.g., GPS location).
- The web page can advise the bot about user actions (e.g., "user just selected Option A from the dropdown").
- The web page can send the bot the auth token for a logged-in user.
- The bot can send relevant data to the web page (e.g., current value of user's portfolio).
- The bot can send "commands" to the web page (e.g., change background color).

Using the backchannel mechanism

The [open source web chat control](#) communicates with bots by using the [Direct Line API](#), which allows `activities` to be sent back and forth between client and bot. The most common type of activity is `message`, but there are other types as well. For example, the activity type `typing` indicates that a user is typing or that the bot is working to compile a response.

You can use the backchannel mechanism to exchange information between client and bot without presenting it to the user by setting the activity type to `event`. The web chat control will automatically ignore any activities where `type="event"`.

Sample code

The [open source web chat control](#) is available via GitHub. For details about how you can implement the backchannel mechanism using the open source web chat control and the Bot Builder SDK for Node.js, see [Use the backchannel mechanism](#).

Additional resources

- [Direct Line API](#)
- [Activity types](#)
- [Use the backchannel mechanism](#)

Azure Bot Service

9/13/2017 • 3 min to read • [Edit Online](#)

The Azure Bot Service accelerates the process of developing a bot. It provisions a web host with one of five bot templates you can modify in an integrated environment.

Azure Bot Service offers two different hosting plans for bots. Converting bot source code from one plan to the other is a manual process.

App Service plan

A bot that uses an App Service plan is a standard Azure web app you can set to allocate a predefined capacity with predictable costs and scaling. With a bot that uses this hosting plan, you can:

- Edit bot source code online using an advanced in-browser code editor.
- Download, debug, and re-publish your C# bot using Visual Studio.
- Set up continuous deployment easily for Visual Studio Online and Github.
- Use sample code prepared for the Bot Builder SDK.

Consumption plan

A bot that uses a Consumption plan is a serverless bot that runs on [Azure Functions](#), and uses the pay-per-run Azure Functions pricing. A bot that uses this hosting plan can scale to handle huge traffic spikes. You can edit bot source code online using a basic in-browser code editor. For more information about the runtime environment of a Consumption plan bot, see [Azure Functions Consumption and App Service plans](#).

Prerequisites

You must have a Microsoft Azure subscription before you can use the Azure Bot Service. If you do not already have a subscription, you can register for a [free trial](#).

Create a bot in seconds

The Azure Bot Service enables you to quickly and easily create a bot in either C# or Node.js by using one of five templates.

TEMPLATE	DESCRIPTION
Basic	Creates a bot that uses dialogs to respond to user input.
Form	Creates a bot that collects input from a user via a guided conversation that is created using FormFlow (in C#) or waterfalls (in Node.js).
Language understanding	Creates a bot that uses natural language models (LUIS) to understand user intent.
Proactive	Creates a bot that uses Azure Functions to alert users of events.

TEMPLATE	DESCRIPTION
Question and Answer	Creates a bot that uses a knowledge base to answer the user's questions.

For more information about templates, see [Templates in the Azure Bot Service](#). For a step-by-step tutorial that shows how to quickly build and test a simple bot using Azure Bot Service, see [Create a bot with Azure Bot Service](#).

Choose development tool(s)

By default, Azure Bot Service enables you to develop your bot directly in the browser using the Azure editor, without any need for a tool chain (i.e., local editor and source control). The integrated chat window sits side-by-side with the Azure editor, which lets you test your bot on-the-fly as you write code in the browser.

Although using Azure editor eliminates the need for an editor and source control on your local computer, Azure editor does not allow you to manage files (e.g., add files, rename files, or delete files). If you have Visual Studio Community or above, you can develop and debug your C# bot locally, and publish your bot to Azure. Also, you can [set up continuous deployment](#) by using the source control system of your choice, with easy setup for Visual Studio Online and Github. With continuous deployment configured, you can develop and debug in an IDE on your local computer, and any code changes that you commit to source control automatically deploy to Azure.

NOTE

After enabling continuous deployment, be sure to modify your code through continuous deployment only and not through other mechanisms to avoid conflict.

Manage your bot

During the process of creating a bot using Azure Bot Service, you specify a name for your bot and generate its App ID and password. After your bot has been created, you can change its settings, configure it to run on one or more channels, or publish it to one or more channels.

Next steps

[Create a bot with the Azure Bot Service](#)

Additional resources

If you encounter problems or have suggestions regarding the Azure Bot Service, see [Support](#) for a list of available resources.

Choose a hosting plan for a bot in Azure Bot Service

9/13/2017 • 1 min to read • [Edit Online](#)

Azure Bot Service offers two different hosting plans for bots. Converting bot source code from one plan to the other is a manual process.

App Service plan

A bot that uses an App Service plan is a standard Azure web app you can set to allocate a predefined capacity with predictable costs and scaling. With a bot that uses this hosting plan, you can:

- Edit bot source code online using an advanced in-browser code editor.
- Download, debug, and re-publish your C# bot using Visual Studio.
- Set up continuous deployment easily for Visual Studio Online and Github.
- Use sample code prepared for the Bot Builder SDK.

Consumption plan

A bot that uses a Consumption plan is a serverless bot that runs on [Azure Functions](#), and uses the pay-per-run Azure Functions pricing. A bot that uses this hosting plan can scale to handle huge traffic spikes. You can edit bot source code online using a basic in-browser code editor. For more information about the runtime environment of a Consumption plan bot, see [Azure Functions Consumption and App Service plans](#).

Templates in the Azure Bot Service

10/12/2017 • 3 min to read • [Edit Online](#)

The Azure Bot Service enables you to quickly and easily create a Consumption plan or App Service plan bot in either C# or Node.js by using one of five templates.

TEMPLATE	DESCRIPTION
Basic	Creates a bot that uses dialogs to respond to user input.
Form	Creates a bot that collects input from a user via a guided conversation that is created using FormFlow (in C#) or waterfalls (in Node.js).
Language understanding	Creates a bot that uses natural language models (LUIS) to understand user intent.
Proactive	Creates a bot that uses Azure Functions to alert users of events.
Question and Answer	Creates a bot that uses a knowledge base to answer the user's questions.

All bots that are created using the Azure Bot Service will initially contain a core set of files specific to the development language and the hosting plan that you choose, along with additional files that contain code that is specific to the selected template.

Files for a bot created in an App Service plan

Bots created in an App Service plan are based on standard Web apps. Here's a list of some important files you'll find in the zip file (in addition to files that are specific to the selected template).

Common files (both C# and Node.js)

FOLDER	FILE	DESCRIPTION
/	readme.md	This file contains information about building bots with the Azure Bot Service. Read this file before you use or modify the bot.
/PostDeployScripts	*.*	Files needed to run some post deployment tasks. Do not touch these files.

C# specific files

FOLDER	FILE	DESCRIPTION
/	*.sln	The Microsoft Visual Studio solutions file. It is used locally if you set up continuous deployment .

FOLDER	FILE	DESCRIPTION
/	build.cmd	This file is needed to deploy your code when you are editing it online via the Azure App Service editor. If you're working locally, you don't need this file.
/Dialogs	*.cs	The classes that define your bot dialogs.
/Controllers	MessagesController.cs	The main controller of your bot application.
/PostDeployScripts	*.PublishSettings	The publish profile for your bot. You can use this file to publish directly from Visual Studio.

Node.js specific files

FOLDER	FILE	DESCRIPTION
/	app.js	The main .js file of your bot.
/	package.json	This file contains the project's npm references. You can modify this file to add a new reference.

Files for a bot created in a Consumption plan

Bots in a Consumption plan are based on Azure Functions. Here's a list of some important files you'll find in the zip file (in addition to files that are specific to the selected template).

Common files (both C# and Node.js)

FOLDER	FILE	DESCRIPTION
/	readme.md	This file contains information about building bots with the Azure Bot Service. Read this file before you use or modify the bot.
/messages	function.json	This file contains the function's bindings. Do not modify this file.
/messages	host.json	This metadata file contains the global configuration options that affect the function.
/PostDeployScripts	*.*	Files needed to run some post deployment tasks. Do not touch these files.

C# specific files

FOLDER	FILE	DESCRIPTION
/	Bot.sln	The Microsoft Visual Studio solutions file. It is used locally if you set up continuous deployment .
/	commands.json	This file contains the commands that start debughost in Task Runner Explorer when you open the Bot.sln file. If you do not install Task Runner Explorer, you can delete this file.
/	debughost.cmd	This file contains the commands to load and run your bot. It is used locally if you set up continuous deployment and debug your bot locally. For more information, see Debug a C# bot using the Azure Bot Service on Windows . This file also contains your bot's App ID and password. To debug authentication, set the App ID and password in this file and also specify the App ID and password when testing your bot using the emulator .
/messages	project.json	This file contains the project's NuGet references. You can modify this file to add a new reference.
/messages	project.lock.json	This file is generated automatically. Do not modify this file.
/messages	run.csx	Defines the initial Run method that gets executed on an incoming request.

Node.js specific files

FOLDER	FILE	DESCRIPTION
/messages	index.js	The main .js file of your bot.
/messages	package.json	This file contains the project's npm references. You can modify this file to add a new reference.

Additional resources

- [Create a bot using the Basic template](#)
- [Create a bot using the Form template](#)
- [Create a bot using the Language understanding template](#)
- [Create a bot using the Proactive template](#)
- [Create a bot using the Question and Answer template](#)

Create a bot using the Basic template

11/2/2017 • 4 min to read • [Edit Online](#)

To create a bot that uses dialogs to respond to user input, choose the [Basic template](#) when creating the bot using Azure Bot Service. This article provides a walkthrough of the code that is automatically generated when you create a Consumption plan bot using the Basic template and describes some ways in which you might extend the bot's default functionality.

TIP

This article examines the source code provided by the Basic template for a Consumption plan C# bot. The code in the Basic template for a web app C# bot is similar. For a walkthrough of code in a basic web app bot, see [Dialogs in the Bot Builder SDK for .NET](#).

Code walkthrough

When a user posts a message, it is sent to the `Run` method in `Run.csx` as an [Activity](#). The bot reacts to an incoming activity by first attempting to authenticate the request. If request validation fails, the bot responds with an **Unauthorized** response.

```
// Authenticates the incoming request. If the request is authentic, it adds the activity.ServiceUrl to
// MicrosoftAppCredentials.TrustedHostNames. Otherwise, it returns Unauthorized.

if (!await botAuthenticator.Value.TryAuthenticateAsync(req, new [] {activity}, CancellationToken.None))
{
    return BotAuthentication.GenerateUnauthorizedResponse(req);
}
```

If request validation succeeds, the bot evaluates the [Activity type](#) to determine how it will process the activity. You may want to update this portion of the code to add processing logic for other types of activities. For example, if your bot saves user state, you would want to process a `DeleteUserData` activity by deleting any data that it had previously stored for the user.

```
switch (activity.GetActivityType())
{
    case ActivityTypes.Message:
        // Processes the user's message.
        break;
    case ActivityTypes.ConversationUpdate:
        // Welcomes the users to the conversation.
        break;
    case ActivityTypes.ContactRelationUpdate:
    case ActivityTypes.Typing:
    case ActivityTypes.DeleteUserData:
    case ActivityTypes.Ping:
    default:
        log.Error($"Unknown activity type ignored: {activity.GetActivityType()}");
        break;
}
```

The first message that a channel sends to the bot is typically a `ConversationUpdate` activity that specifies the users that are present in the conversation. The default code that is generated by the template shows how you can

respond to the `ConversationUpdate` activity by welcoming users to the conversation. The list of users that the `ConversationUpdate` activity specifies (as members of the conversation) will include the bot, so this code intentionally omits the bot from the list of users that it welcomes to the conversation.

```
case ActivityTypes.ConversationUpdate:  
    IConversationUpdateActivity update = activity;  
    using (var scope = DialogModule.BeginLifetimeScope(Conversation.Container, activity))  
    {  
        var client = scope.Resolve<IConnectorClient>();  
        if (update.MembersAdded.Any())  
        {  
            var reply = activity.CreateReply();  
            var newMembers = update.MembersAdded?.Where(t => t.Id != activity.Recipient.Id);  
            foreach (var newMember in newMembers)  
            {  
                reply.Text = "Welcome";  
                if (!string.IsNullOrEmpty(newMember.Name))  
                {  
                    reply.Text += $" {newMember.Name}";  
                }  
                reply.Text += "!";  
                await client.Conversations.ReplyToActivityAsync(reply);  
            }  
        }  
        break;  
    }
```

Most activities that the bot receives will be of type `Message` and will contain the text and attachments that the user sent to the bot. To process an incoming message, the bot posts the message to `EchoDialog` (in **EchoDialog.csx**).

```
switch (activity.GetActivityType())  
{  
    case ActivityTypes.Message:  
        await Conversation.SendAsync(activity, () => new EchoDialog());  
        break;  
    ...  
}
```

EchoDialog.csx

EchoDialog.csx contains the dialog that controls the conversation with the user. When the dialog is instantiated, the dialog's `StartAsync` method runs and calls `IDialogContext.Wait` with the continuation delegate that will be called when there is a new message. In the initial case, there is an immediate message available (the one that launched the dialog) and the message is immediately passed to the `MessageReceivedAsync` method.

```

[Serializable]
public class EchoDialog : IDialog<object>
{
    protected int count = 1;

    public Task StartAsync(IDialogContext context)
    {
        try
        {
            context.Wait(MessageReceivedAsync);
        }
        catch (OperationCanceledException error)
        {
            return Task.FromCanceled(error.CancellationToken);
        }
        catch (Exception error)
        {
            return Task.FromException(error);
        }

        return Task.CompletedTask;
    }

    ...
}

```

The `MessageReceivedAsync` method echoes the user's input and counts the number of interactions with the user. If the user's input is the word "reset", the method uses `PromptDialog` to confirm that the user wants to reset the counter. The method calls the `IDialogContext.Wait` method to suspend the bot until it receives the next message.

```

public virtual async Task MessageReceivedAsync(IDialogContext context, IAwaitable<IMessageActivity> argument)
{
    var message = await argument;
    if (message.Text == "reset")
    {
        PromptDialog.Confirm(
            context,
            AfterResetAsync,
            "Are you sure you want to reset the count?",
            "Didn't get that!",
            promptStyle: PromptStyle.Auto);
    }
    else
    {
        await context.PostAsync($"{this.count++}: You said {message.Text}");
        context.Wait(MessageReceivedAsync);
    }
}

```

The `AfterResetAsync` method processes the user's response to `PromptDialog`. If the user confirms the request, the counter is reset and a message is sent to the user confirming the action. The method then calls the `IDialogContext.Wait` method to suspend the bot until it receives the next message, which will be processed by `MessageReceivedAsync`.

```
public async Task AfterResetAsync(IDialogContext context, IAwaitable<bool> argument)
{
    var confirm = await argument;
    if (confirm)
    {
        this.count = 1;
        await context.PostAsync("Reset count.");
    }
    else
    {
        await context.PostAsync("Did not reset count.");
    }
    context.Wait(MessageReceivedAsync);
}
```

Next steps

The Basic template provides a good foundation that you can build upon to create a bot that is capable of handling more advanced interactions with users. For example, you might add more prompts to collect information from the user, add [media attachments](#) or [rich cards](#) to messages to provide a richer user experience, or model a more complex conversation flow by using [dialog chains](#).

Additional resources

- [Create a bot with the Azure Bot Service](#)
- [Templates in the Azure Bot Service](#)
- [Dialogs in the Bot Builder SDK for .NET](#)
- [Manage message flow with dialogs in the Bot Builder SDK for Node.js](#)
- [Bot Builder Samples GitHub repository](#)
- [Bot Builder SDK for .NET](#)
- [Bot Builder SDK for Node.js](#)

Create a bot using the Form template

9/13/2017 • 3 min to read • [Edit Online](#)

To create a bot that collects input from a user via a guided conversation, choose the [Form template](#) when creating the bot using Azure Bot Service. A form bot is designed to collect a specific set of information from the user. For example, a bot that is designed to obtain a user's sandwich order would need to collect information such as type of bread, choice of toppings, size of sandwich, etc. This article provides a walkthrough of the code that is automatically generated when you create a bot using the Form template and describes some ways in which you might extend the bot's default functionality.

NOTE

If you choose C# as your development language, the bot will use [FormFlow](#) to manage the guided conversation. If you choose Node.js as your development language, the bot will use [waterfalls](#) to manage the guided conversation. A bot that is created using the Form template routes messages in the same manner as described for the [Basic template](#).

TIP

This article examines the source code provided by the Form template for serverless bots. For examination of the Form template for a web app C# bot, see [Basic features of FormFlow](#).

Code walkthrough

Most activities that the bot receives will be of `type Message` and will contain the text and attachments that the user sent to the bot. To process an incoming message, the bot posts the message to `MainDialog` (in [MainDialog.csx](#)).

```
switch (activity.GetActivityType())
{
    case ActivityTypes.Message:
        await Conversation.SendAsync(activity, () => new MainDialog());
        break;
    ...
}
```

MainDialog.csx

`MainDialog` inherits from the `BasicForm` dialog (in [BasicForm.csx](#)), which defines the form. [MainDialog.csx](#) contains the root dialog that controls the conversation with the user. When the dialog is instantiated, its `StartAsync` method runs and calls `IDialogContext.Wait` with the continuation delegate that will be called when there is a new message. In the initial case, there is an immediate message available (the one that launched the dialog) and the message is immediately passed to the `MessageReceivedAsync` method.

```

public class MainDialog : IDialog<BasicForm>
{
    public MainDialog()
    {
    }

    public Task StartAsync(IDialogContext context)
    {
        context.Wait(MessageReceivedAsync);
        return Task.CompletedTask;
    }
    ...
}

```

The `MessageReceivedAsync` method creates the form and begins to ask the user the questions that are defined by the form. The `Call` method starts the form and specifies the delegate that handles the completed form. When the `FormComplete` method finishes processing the user's input, it calls the `IDialogContext.Wait` method to suspend the bot until it receives the next message.

```

public virtual async Task MessageReceivedAsync(IDialogContext context, IAwaitable<IMessageActivity> argument)
{
    var message = await argument;
    context.Call(BasicForm.BuildFormDialog(FormOptions.PromptInStart), FormComplete);
}

private async Task FormComplete(IDialogContext context, IAwaitable<BasicForm> result)
{
    try
    {
        var form = await result;
        if (form != null)
        {
            await context.PostAsync("Thanks for completing the form! Just type anything to restart it.");
        }
        else
        {
            await context.PostAsync("Form returned empty response! Type anything to restart it.");
        }
    }
    catch (OperationCanceledException)
    {
        await context.PostAsync("You canceled the form! Type anything to restart it.");
    }

    context.Wait(MessageReceivedAsync);
}

```

BasicForm.csx

The `BasicForm` class defines the form. Its public properties define the information that the bot needs to collect from the user. Each `Prompt` property uses [Pattern Language](#) to customize the text that prompts the user for the corresponding piece of information.

```

public enum CarOptions { Convertible=1, SUV, EV };
public enum ColorOptions { Red=1, White, Blue };

[Serializable]
public class BasicForm
{
    [Prompt("Hi! What is your {&}?")]
    public string Name { get; set; }

    [Prompt("Please select your favorite car type {||}")]
    public CarOptions Car { get; set; }

    [Prompt("Please select your favorite {&} {||}")]
    public ColorOptions Color { get; set; }

    public static IForm<BasicForm> BuildForm()
    {
        // Builds an IForm<T> based on BasicForm
        return new FormBuilder<BasicForm>().Build();
    }

    public static IFormDialog<BasicForm> BuildFormDialog(FormOptions options = FormOptions.PromptInStart)
    {
        // Generate a new FormDialog<T> based on IForm<BasicForm>
        return FormDialog.FromForm(BuildForm, options);
    }
}

```

Extend default functionality

The Form template provides a good foundation that you can build upon and customize to create a bot that collects input from a user via a guided conversation. For example, you could edit the `BasicForm` class within **BasicForm.csx** to define the set of information that your bot needs to collect from the user. To learn more about FormFlow, see [FormFlow in .NET](#).

Additional resources

- [Create a bot with the Azure Bot Service](#)
- [Templates in the Azure Bot Service](#)
- [Basic features of FormFlow \(.NET\)](#)
- [Prompts and waterfalls \(Node.js\)](#)
- [Bot Builder Samples GitHub repository](#)
- [Bot Builder SDK for .NET](#)
- [Bot Builder SDK for Node.js](#)

Create a bot using the Language understanding template

11/2/2017 • 3 min to read • [Edit Online](#)

If a user sends a message such as "get news about virtual reality companies," your bot can use LUIS to interpret the meaning of the message. Using [LUIS](#), you can quickly deploy an HTTP endpoint that will interpret user input in terms of the intention that it conveys (find news) and the key entities that are present (virtual reality companies). LUIS enables you to specify the set of intents and entities that are relevant to your application, and then guides you through the process of building a language understanding application.

To create a bot that uses natural language models (LUIS) to understand user intent, choose the [Language understanding template](#) when creating the bot using Azure Bot Service. This article provides a walkthrough of the code that is automatically generated when you create a bot using the Language understanding template.

Customize the LUIS application model

When you create a bot using the Language understanding template, Azure Bot Service creates a corresponding LUIS application that is empty (i.e., that always returns `None`). To update your LUIS application model so that it is capable of interpreting user input, you must sign-in to [LUIS](#), click **My applications**, select the application that the service created for you, and then create intents, specify entities, and train the application.

NOTE

A bot that is created using the Language understanding template routes messages in the same manner as described for the [Basic template](#).

Code walkthrough

Most activities that the bot receives will be of `type Message` and will contain the text and attachments that the user sent to the bot. To process an incoming message, the bot posts the message to `BasicLuisDialog` (in [BasicLuisDialog.cs](#) or [BasicLuisDialog.csx](#)).

```
switch (activity.GetActivityType())
{
    case ActivityTypes.Message:
        await Conversation.SendAsync(activity, () => new BasicLuisDialog());
        break;
    ...
}
```

BasicLuisDialog

The [BasicLuisDialog.cs](#) or [BasicLuisDialog.csx](#) source file contains the root dialog that controls the conversation with the user. The `BasicLuisDialog` object defines an intent method handler for each intent that you define in your LUIS application model. The naming convention for intent handlers is `<intent name>+Intent` (for example, `NoneIntent`). The `LuisIntent` method attribute defines a method as an intent handler and the name specified in the `LuisIntent` attribute must match the name of the corresponding intent that is specified in the LUIS application model. In this example, the dialog will handle the `None` intent (which LUIS returns if it cannot determine the intent) and the `MyIntent` intent (which you will need to define in your LUIS application model).

The `BasicLuisDialog` object inherits from the `LuisDialog` object, which contains the `StartAsync` and `MessageReceived` methods. When the dialog is instantiated, its `StartAsync` method runs and calls `IDialogContext.Wait` with the continuation delegate that will be called when there is a new message. In the initial case, there is an immediate message available (the one that launched the dialog) and the message is immediately passed to the `MessageReceived` method (in the `LuisDialog` object).

The `MessageReceived` method calls your LUIS application model to determine intent and then calls the appropriate intent handler in the `BasicLuisDialog` object. The handler processes the intent and then waits for the next message from the user.

```
[Serializable]
public class BasicLuisDialog : LuisDialog<object>
{
    public BasicLuisDialog() : base(new LuisService(new LuisModelAttribute(Utils.GetAppSetting("LuisAppId"),
    Utils.GetAppSetting("LuisAPIKey"))))
    {
    }

    [LuisIntent("None")]
    public async Task NoneIntent(IDialogContext context, LuisResult result)
    {
        await context.PostAsync($"You have reached the none intent. You said: {result.Query}"); // 
        context.Wait(MessageReceived);
    }

    [LuisIntent("MyIntent")]
    public async Task MyIntent(IDialogContext context, LuisResult result)
    {
        await context.PostAsync($"You have reached the MyIntent intent. You said: {result.Query}"); // 
        context.Wait(MessageReceived);
    }
}
```

Extend default functionality

The Language understanding template provides a good foundation that you can build upon to create a bot that is capable of using natural language models to understand user intent. To learn more about developing bots with LUIS in .NET, see [Enable language understanding with LUIS](#). To learn more about developing bots with LUIS in Node.js, see [Recognize user intent](#).

Additional resources

- [Create a bot with the Azure Bot Service](#)
- [Templates in the Azure Bot Service](#)
- [LUIS](#)
- [Enable language understanding with LUIS \(.NET\)](#)
- [Recognize user intent \(Node.js\)](#)
- [Bot Builder Samples GitHub repository](#)
- [Bot Builder SDK for .NET](#)
- [Bot Builder SDK for Node.js](#)

Create a bot using the Proactive template

11/2/2017 • 5 min to read • [Edit Online](#)

Typically, each message that a bot sends to the user directly relates to the user's prior input. In some cases though, a bot may need to send the user information that is not directly related to the user's most recent message. These types of messages are called **proactive messages**. Proactive messages can be useful in a variety of scenarios. For example, if a bot sets a timer or reminder, it may need to notify the user when the time arrives. Or, if a bot receives a notification about an external event, it may need to communicate that information to the user.

To create a bot that can send proactive messages to the user, choose the [Proactive template](#) when creating the bot using Azure Bot Service. This article describes the Azure resources that are created when you create a bot using the Proactive template and provides a walkthrough of the code that is automatically generated by the template.

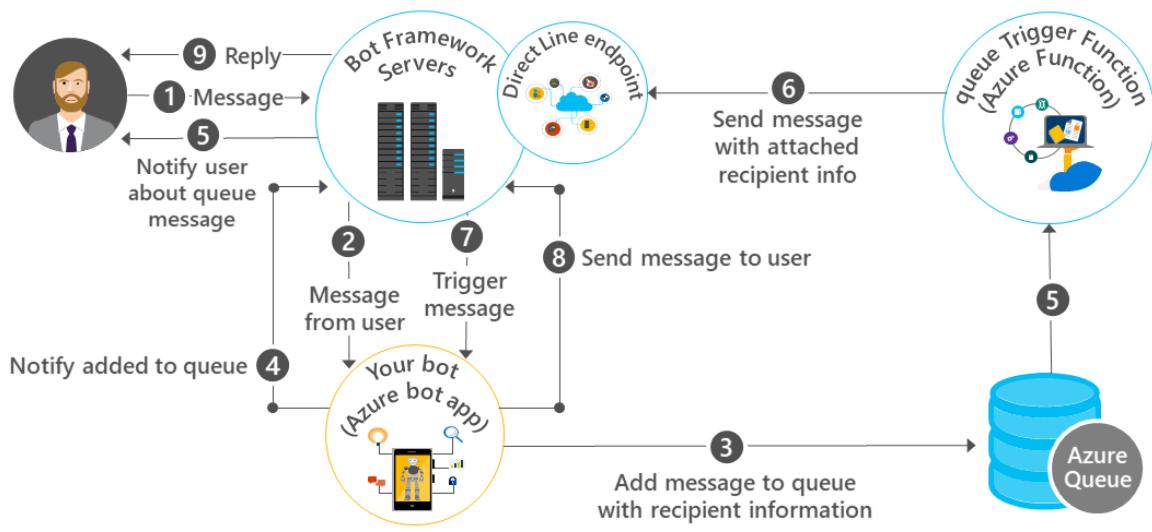
Azure resources

When you create a bot by using the Proactive template, several Azure resources are automatically created and added to your resource group. By default, these Azure resources are already configured to enable a very simple proactive messaging scenario.

RESOURCE	DESCRIPTION
Azure Storage	Used to create the queue.
Azure Function App	A <code>queueTrigger</code> Azure Function that is triggered whenever there is a message in the queue. It communicates to the Azure Bot Service by using Direct Line . This function uses bot binding to send the message as part of the trigger's payload. Our example function forwards the user's message as-is from the queue.
Azure Bot Service	Your bot. Contains the logic that receives the message from user, adds the message to the Azure queue, receives triggers from Azure Function, and sends back the message it received via trigger's payload.

Process overview

This diagram shows how triggered events work when you create a bot using the Proactive template.



The process begins when the user sends a message to your bot via Bot Framework servers (step 1). The [Code walkthrough](#) section of this article describes the remaining steps of the process.

Code walkthrough

Receive a message from the user and add it to an Azure Storage queue

When the bot receives a message from the user (step 2), it adds the message to the Azure Storage queue (step 3), and notifies the user (via the Bot Framework servers) that the message has been queued (step 4). The message is encapsulated within an object that contains information that will eventually be needed to send the proactive message to the user on the correct channel.

This code snippet uses C# to receive a message from the user, add the message to the Azure Storage queue, and notify the user that the message has been queued. The [RelatesTo](#) property of the queue message is set to a [ConversationReference](#) object, which contains the information that is needed to respond to the user on the correct channel.

```
public virtual async Task MessageReceivedAsync(IDialogContext context, IAwaitable<IMessageActivity> argument)
{
    var message = await argument;

    // Create a queue Message
    var queueMessage = new Message
    {
        RelatesTo = context.Activity.ToConversationReference(),
        Text = message.Text
    };

    // Write the queue Message to the queue
    // AddMessageToQueue() is a utility method you can find in the template
    await AddMessageToQueue(JsonConvert.SerializeObject(queueMessage));
    await context.PostAsync($"{this.count++}: You said {queueMessage.Text}. Message added to the queue.");
    context.Wait(MessageReceivedAsync);
}
```

This code snippet uses Node.js to receive a message from the user, add the message to the Azure Storage queue, and notify the user that the message has been queued. The [session.message.address](#) object contains the information that is needed to respond to the user on the correct channel.

```

bot.dialog('/', function (session) {
  var queuedMessage = { address: session.message.address, text: session.message.text };
  session.sendTyping();
  var queueSvc = azure.createQueueService(process.env.AzureWebJobsStorage);
  queueSvc.createQueueIfNotExists('bot-queue', function(err, result, response){
    if(!err){
      var queueMessageBuffer = new Buffer(JSON.stringify(queuedMessage)).toString('base64');
      queueSvc.createMessage('bot-queue', queueMessageBuffer, function(err, result, response){
        if(!err){
          session.send('Your message (\'' + session.message.text + '\') has been added to a queue,
and it will be sent back to you via a Function');
        } else {
          session.send('There was an error inserting your message into queue');
        }
      });
    } else {
      session.send('There was an error creating your queue');
    }
  });
});
});

```

Trigger an Azure Function and send the message to the bot

After the message is added to the queue, the function is automatically triggered (step 5). The message is then removed from the queue and sent back to the bot (step 6). The function's configuration file (**functions.json**) contains an input binding of type `queueTrigger` and an output binding of type `bot`.

```
{
  "bindings": [
    {
      "name": "myQueueItem",
      "type": "queueTrigger",
      "direction": "in",
      "queueName": "bot-queue",
      "connection": ""
    },
    {
      "type": "bot",
      "name": "$return",
      "direction": "out",
      "botId": "yourbot"
    },
    {
      "type": "http",
      "name": "res",
      "direction": "out"
    }
  ]
}
```

This code snippet shows the Function code in C#.

```

using System;
using System.Net;
using System.Net.Http;
using Microsoft.Azure.WebJobs.Host;

public class BotMessage
{
    public string Source { get; set; }
    public string Message { get; set; }
}

public static HttpResponseMessage Run(string myQueueItem, out BotMessage message, TraceWriter log)
{
    message = new BotMessage
    {
        Source = "Azure Functions (C#)!",
        Message = myQueueItem
    };

    return new HttpResponseMessage(HttpStatusCode.OK);
}

```

This code snippet shows the Function code in Node.js.

```

module.exports = function (context, myQueueItem) {
    context.log('Sending Bot message', myQueueItem);

    var message = {
        'text': myQueueItem.text,
        'address': myQueueItem.address
    };

    context.done(null, message);
}

```

Receive the message from the Azure Function and send proactive message

Finally, the bot receives the message from the trigger function (step 7) and sends a proactive message to the user (step 8).

This code snippet uses C# to receive the message from the Azure Function and send a message to the user.

```

switch (activity.GetActivityType())
{
    case ActivityTypes.Event:
        // handle proactive Message from function
        IEventActivity triggerEvent = activity;
        var message = JsonConvert.DeserializeObject<Message>(((JObject)
triggerEvent.Value).GetValue("Message").ToString());
        var messageactivity = (Activity)message.RelatesTo.GetPostToBotMessage();

        client = new ConnectorClient(new Uri(messageactivity.ServiceUrl));
        var triggerReply = messageactivity.CreateReply();
        triggerReply.Text = $"This is coming back from the trigger! {message.Text}";
        await client.Conversations.ReplyToActivityAsync(triggerReply);
        break;
    default:
        break;
}

```

This code snippet uses Node.js to receive the message from the Azure Function and send a message to the user.

```
bot.on('trigger', function (message) {
  // handle message from trigger function
  var queuedMessage = message.value;
  var reply = new builder.Message()
    .address(queuedMessage.address)
    .text('This is coming from the trigger: ' + queuedMessage.text);
  bot.send(reply);
});
```

Extend default functionality

The Proactive template provides a good foundation that you can build upon and customize to enable proactive messaging scenarios for your bot. To learn more about sending proactive messages, see [Send proactive messages with .NET](#) or [Send proactive messages with Node.js](#).

Additional resources

- [Create a bot with the Azure Bot Service](#)
- [Templates in the Azure Bot Service](#)
- [Azure Functions](#)
- [Azure Functions Storage queue bindings](#)
- [Send proactive messages \(.NET\)](#)
- [Send proactive messages \(Node.js\)](#)
- [Bot Builder Samples GitHub repository](#)
- [Bot Builder SDK for .NET](#)
- [Bot Builder SDK for Node.js](#)

Create a bot using the Question and Answer template

9/13/2017 • 3 min to read • [Edit Online](#)

To create a bot that can answer frequently asked questions (FAQs), choose the [Question and Answer template](#) when creating the bot using Azure Bot Service. This article describes how to specify a knowledge base for your bot and provides a walkthrough of the code that is automatically generated when you create a bot using the Question and Answer template.

NOTE

A bot that is created using the Question and Answer template routes messages in the same manner as described for the [Basic template](#).

Specify a knowledge base for your bot

To enable your bot to answer FAQs, you must connect it to a knowledge base of questions and answers. Using [QnA Maker](#), you can either manually configure your own custom list of questions and answers or scrape questions and answers from an existing FAQ site or structured document. For more information about creating a knowledge base for your bot using QnA Maker, see the QnA Maker [documentation](#).

When you use the Question and Answer template to create a bot with the Azure Bot Service, you can connect to an existing knowledge base that you have already created using [QnA Maker](#) or create a new (empty) knowledge base.

Code walkthrough

Most activities that the bot receives will be of type `Message` and will contain the text and attachments that the user sent to the bot. To process an incoming message, the bot posts the message to `BasicQnAMakerDialog` (in [BasicQnAMakerDialog.cs](#) or [BasicQnAMakerDialog.csx](#)).

This code snippet uses C# to post an incoming message to `BasicQnAMakerDialog`.

```
switch (activity.GetActivityType())
{
    case ActivityTypes.Message:
        await Conversation.SendAsync(activity, () => new BasicQnAMakerDialog());
        break;
    ...
}
```

This code snippet uses Node.js to post an incoming message to `BasicQnAMakerDialog`.

```
bot.dialog('/', BasicQnAMakerDialog);
```

BasicQnAMakerDialog

The `BasicQnAMakerDialog` object inherits from the `QnAMakerDialog` object, which contains the `StartAsync` and `MessageReceived` methods. When the dialog is instantiated, its `StartAsync` method runs and calls `IDialogContext.Wait` with the continuation delegate that will be called when there is a new message. In the initial case, there is an immediate message available (the one that launched the dialog) and the message is immediately

passed to the `MessageReceived` method. The `MessageReceived` method calls the QnA Maker service and returns the response to the user.

TIP

For C#, the QnAMaker Dialog is distributed via the `Microsoft.Bot.Builder.CognitiveServices` NuGet package. For Nodejs, the QnAMaker Dialog is distributed via the `botbuilder-cognitiveservices` npm module.

The call to invoke the QnA Maker service can include up to four parameters:

PARAMETER	REQUIRED OR OPTIONAL	DESCRIPTION
Subscription Key	required	The subscription key that you acquire when you register with QnA Maker. (QnA Maker assigns each registered user a unique subscription key for metering purposes.)
Knowledge Base ID	required	The unique identifier for the QnA Maker knowledge base to which you want to connect.
Default Message	optional	The message to show if no match is found in the knowledge base.
Score Threshold	optional	A value between 0-1 that represents the threshold value of the match confidence score returned by the service. You can use this parameter to control the relevance of the responses.

This code snippet uses C# to invoke the QnA Maker service.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using System.Threading;
using Microsoft.Bot.Connector;
using Microsoft.Bot.Builder.Dialogs;
using Microsoft.Bot.Builder.CognitiveServices.QnAMaker;

[Serializable]
public class BasicQnAMakerDialog : QnAMakerDialog
{
    //Parameters to QnAMakerService are:
    //Required: subscriptionKey, knowledgebaseId,
    //Optional: defaultMessage, scoreThreshold[Range 0.0 - 1.0]
    public BasicQnAMakerDialog() : base(new QnAMakerService(new
        QnAMakerAttribute(Utils.GetAppSetting("QnASubscriptionKey"), Utils.GetAppSetting("QnAKnowledgebaseId"), "No
        good match in FAQ.", 0.5)))
    {}
}
```

This code snippet uses Node.js to invoke the QnA Maker service.

```
var recognizer = new cognitiveservices.QnAMakerRecognizer({
  knowledgeBaseId: 'set your kbid here',
  subscriptionKey: 'set your subscription key here'});

var BasicQnAMakerDialog = new cognitiveservices.QnAMakerDialog({
  recognizers: [recognizer],
  defaultMessage: 'No good match in FAQ.',
  qnaThreshold: 0.5});
```

Extend default functionality

The Question and Answer template provides a good foundation that you can build upon and customize to create a bot that can answer frequently asked questions (FAQs). To learn more about using QnA Maker with your bot, see [Design knowledge bots](#) and [QnA Maker](#).

Additional resources

- [Create a bot with the Azure Bot Service](#)
- [Templates in the Azure Bot Service](#)
- [Design knowledge bots](#)
- [QnA Maker](#)
- [Bot Builder Samples GitHub repository](#)
- [Bot Builder SDK for .NET](#)
- [Bot Builder SDK for Node.js](#)

Build tab features of Azure Bot Service

10/12/2017 • 2 min to read • [Edit Online](#)

An Azure Bot Service bot that uses an App Service plan includes these features on its **Build** tab.

NOTE

The **Build** tab of an Azure Bot Service bot that uses a Consumption plan only provides a basic browser-based code editor. You can download source code and configure continuous deployment for a Consumption plan bot using features located on its **Settings** tab, in the **Continuous deployment** section.

Online code editor

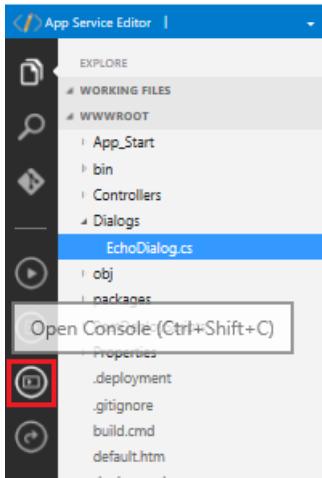
The online code editor lets you change the source code of your bot via the [Azure App Service Editor](#). The editor provides IntelliSense in C# source files.

The online code editor is a great way to start, but if you need advanced debugging and file management, get the bot code locally by downloading it and optionally by setting up continuous deployment.

Deploy from the online code editor

When you change a source file in the online code editor, you must run the deployment script before your changes take effect. Follow these steps to run the deployment script.

1. In the App Service Editor, click the Open Console icon.



2. In the Console window, type **build.cmd**, and press the enter key.

Download source code

You can download a zip file that contains all source files, a Visual Studio solution file, and a configuration file that lets you publish from Visual Studio. Using these files, you can modify and debug your bot in Visual Studio, or your favorite IDE, and test it with the Bot Framework Emulator. Once you are ready to publish, you can setup publishing directly from Visual Studio by importing the publish profile saved in the `.PublishSettings` file within the `PostDeployScripts` folder. For more information, see [Publish C# bot on App Service plan from Visual Studio](#).

Continuous deployment from source control

Continuous deployment lets you re-publish to Azure whenever you check a code change in to your source control

service. If you work in a team and need to share code in a source control system, then you should setup continuous deployment and use the integrated development environment (IDE) and source control system of your choice.

NOTE

Some templates provided by the Azure Bot Service, and especially templates on a Consumption plan, require additional setup steps to [debug on your local computer](#).

The Azure Bot Service provides a quick way to set up continuous deployment for Visual Studio Online and GitHub, by providing an access token issued to you on those web sites. For other source control systems, select **other** and follow the steps that appear. For more help setting up continuous deployment on a source control other than Visual Studio Online or GitHub, see [Set up continuous deployment](#).

WARNING

When you use continuous deployment, be sure to only modify code by checking it into your source control service. Do not use the online code editor to change source code when continuous deployment is enabled. For Consumption plan bots, the online code editor is read-only when continuous deployment is enabled.

Publish a bot to Azure Bot Service

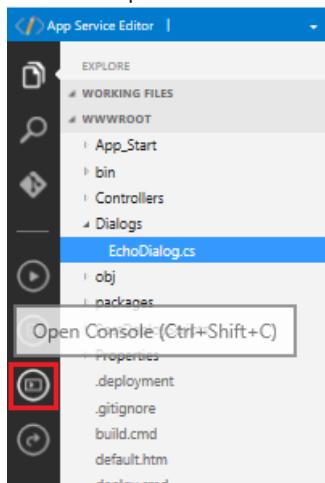
10/12/2017 • 4 min to read • [Edit Online](#)

After you update your C# bot source code, you can publish it into a running bot in Azure Bot Service using Visual Studio. You can also publish either C# or Node.js bot source code written in any integrated development environment (IDE) automatically each time you check a source file into a source control service.

Publish a bot on App Service plan from the online code editor

If you have not configured continuous deployment, you can modify your source files in the online code editor. To deploy your modified source, follow these steps.

1. Click the Open Console icon.



2. In the Console window, type **build.cmd**, and press the enter key.

Publish C# bot on App Service plan from Visual Studio

To set up publishing from Visual Studio using the `.PublishSettings` file, perform the following steps:

1. In the Azure Portal, click your Bot Service, click the **BUILD** tab, and click **Download zip file**.
2. Extract the contents of the downloaded zip file to a local folder.
3. In Explorer, find the Visual Studio Solution (.sln) file for your bot, and double-click it.
4. In Visual Studio, click **View**, and click **Solution Explorer**.
5. In the Solution Explorer pane, right-click your project, and click **Publish...** The Publish window opens.
6. In the Publish window, click **Create new profile**, click **Import profile**, and click **OK**.
7. Navigate to your project folder, navigate to the **PostDeployScripts** folder, select the file that ends in `.PublishSettings`, and click **Open**.

You have now configured publishing for this project. To publish your local source code to Azure Bot Service, right-click your project, click **Publish...**, and click the **Publish** button.

Set up continuous deployment

By default, Azure Bot Service enables you to develop your bot directly in the browser using the Azure editor, without any need for a local editor or source control. However, Azure editor does not allow you to manage files within your application (e.g., add files, rename files, or delete files). If you want the ability to manage files within your application, you can set up continuous deployment and use the integrated development environment (IDE)

and source control system of your choice (e.g., Visual Studio Team, GitHub, Bitbucket). With continuous deployment configured, any code changes that you commit to source control will automatically be deployed to Azure. After you configure continuous deployment, you can [debug your bot locally](#).

NOTE

If you enable continuous deployment for your bot, you must check in code changes to your source control service. If you want to edit your code in Azure editor once again, you must [disable continuous deployment](#).

You can enable continuous deployment for your bot app by completing the following steps.

Set up continuous deployment for a bot on an App Service plan

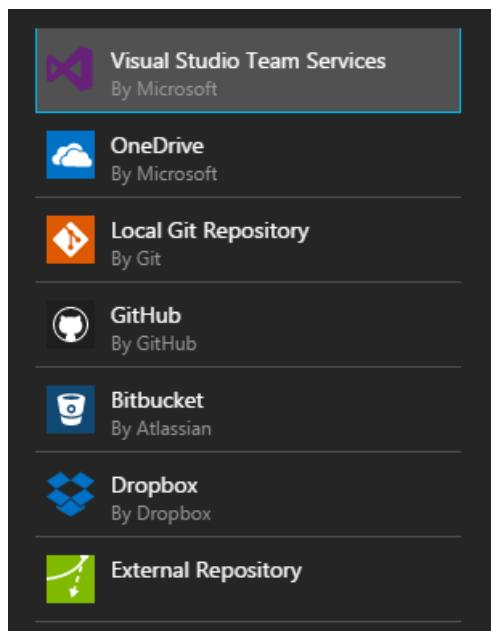
This section describes how to enable continuous deployment for a bot that you created using Azure Bot Service that has an App Service hosting plan.

1. Within the Azure Portal, find your Azure bot, click the **BUILD** tab and find the **Continuous deployment from source control** section.
2. For Visual Studio Online or GitHub, provide an access token issued to you on those web sites. Your source will be pulled from Azure into your source repository.
3. For other source control systems, select **other** and follow the steps that appear.
4. Click **Enable**.

Create an empty repository and download bot source code

Follow these steps if you want to use a source control service *other than* Visual Studio Online or Github. Visual Studio Online and Github will pull the source code for your bot from Azure, so users of those two services can skip these steps.

1. For a bot on an App Service plan, find your bot page on Azure, click the **BUILD** tab, find the **Download source code** section, and click **Download zip file**.
2. Create an empty repository within one of the source control systems that Azure supports.



3. Extract the contents of the downloaded zip file to the local folder where you are planning to sync your deployment source.
4. Click **Configure** and follow the steps that appear.

Set up continuous deployment for a bot on a Consumption plan

Choose the deployment source for your bot and connect your repository.

1. Within the Azure Portal, in your Azure bot, click the **SETTINGS** tab and click **Configure** to expand the **Continuous deployment** section.
2. Follow the steps and click the checkbox to confirm you're ready.
3. Click **Configure**, select the deployment source that corresponds to the source control system where you previously created the empty repository, and complete the steps to connect it.

Disable continuous deployment

When you disable continuous deployment, your source control service continues functioning, but changes you check in are not automatically published to Azure. To disable continuous deployment, perform the following steps:

1. If your bot has an App Service hosting plan, within the Azure Portal, find your Azure bot, click the **BUILD** tab and find the **Continuous deployment from source control** section, or...
2. if your bot has a Consumption plan, click the **Settings** tab, expand the **Continuous deployment** section, and click **Configure**.
3. In the **Deployments** pane, select the source control service where continuous deployment is enabled, and click **Disconnect**.

Additional resources

To learn how to debug your bot locally after you have configured continuous deployment, see [Debug an Azure Bot Service bot](#).

This article has highlighted the specific continuous deployment features of Azure Bot Service. For information about continuous deployment as it relates to Azure App Services, see [Continuous Deployment to Azure App Service](#).

Migrate a bot from a Consumption plan to an App Service plan

10/12/2017 • 3 min to read • [Edit Online](#)

This article shows you how you can migrate a C# script bot with a Consumption plan into a C# bot with an App Service plan.

This migration assumes you have Visual Studio (Community or above).

Advantages of a bot on an App Service plan

Bots on an App Service plan run as Azure web apps. Web app bots can do things that Consumption plan bots cannot:

- A web app bot can add custom route definitions.
- A web app bot can enable a Websocket server.
- A bot that uses a Consumption hosting plan has the same limitations as all code running on Azure Functions.

For more information, see [Azure Functions Consumption and App Service plans](#).

Download your existing bot source

Follow these steps to download the source code of your existing bot:

1. Within your Azure bot, click the **Settings** tab and expand the **Continuous deployment** section.
2. Click the blue button to download the zip file that contains the source code for your bot.

Continuous deployment [Configure ^](#)
Deploy your bot code from GitHub, Visual Studio Team Services, and more.

Step 1: download your source code:

[Download zip file](#)

Step 2: create a folder/repo for the source files in your preferred service

I completed step 2 and can proceed to step 3

Step 3: Configure continuous deployment

[Configure](#)

Note: After enabling continuous deployment, you will no longer be able to modify the code in Azure.

3. Extract the contents of the downloaded zip file to a local folder.

Create a bot template

Azure Bot Service has the same templates for Consumption plan and App Service plan bots. To migrate from a Consumption plan bot, create a new App Service plan bot in Azure Bot Service, based on the same template. Underlying code can differ between the two hosting types for the same template, but the new web app has similar structure and configuration features that your existing bot uses.

Download the new bot source

Follow these steps to download the source code of the new bot:

1. Within your Azure bot, click the **BUILD** tab, find the **Download source code** section, and click **Download zip file**.
2. Extract the contents of the downloaded zip file to the local folder.

Add source files to new solution

Some .csx files might compile and run as .cs files in the new solution. Create a .cs file for each .csx file in your solution, except `run.csx`. You will migrate `run.csx` logic by hand. In your .cs files, you may need to add a class declaration and optional namespace declaration.

Migrate run.csx logic into your project

C# script projects have a `Run` method where different `ActivityTypes` values are handled. Import your activity handling logic into the `MessageController.Post` method, in `MessageController.cs`.

Remove #r and #load compiler keywords

C# script files can include a reference module using the `#r` keyword. Remove these lines, and also add these as references to your Visual Studio project. Also remove `#load` keywords, which insert other source code files in a file compilation. Instead, add all `.csx` files as to your project as `.cs` source code.

Add references from project.json

If your Consumption plan bot adds NuGet references in its `project.json` file, add these references to your new Visual Studio solution by right-clicking the project in the Solution Explorer pane, and clicking **Add Reference**.

Add references that were implicit

An Azure Bot Service bot on a Consumption plan implicitly includes these references in all .csx source files. Source migrated into .cs source files may need to add explicit references for these classes:

- `TraceWriter` in the NuGet package `Microsoft.Azure.WebJobs` that provides the `Microsoft.Azure.WebJobs.Host` namespace types.
- timer triggers in the NuGet package `Microsoft.Azure.WebJobs.Extensions`
- `Newtonsoft.Json`, `Microsoft.ServiceBus`, and other automatically referenced assemblies
- `System.Threading.Tasks` and other automatically imported namespaces

For additional guidance, see *Converting to class files* in [Publishing a .NET class library as a Function App](#).

Debug your new bot

A bot on an App Service plan is much easier than a bot on a Consumption plan to debug locally. You can use the [emulator](#) to debug your migrated code locally.

Publish from Visual Studio, or set up continuous deployment

Finally, publish your migrated source code to Azure Bot Service either by importing its `.PublishSettings` file and clicking **Publish**, or by [setting up continuous deployment](#).

Bot Builder SDK for .NET

8/7/2017 • 2 min to read • [Edit Online](#)

The Bot Builder SDK for .NET is a powerful framework for constructing bots that can handle both free-form interactions and more guided conversations where the user selects from possible values. It is easy to use and leverages C# to provide a familiar way for .NET developers to write bots.

Using the SDK, you can build bots that take advantage of the following SDK features:

- Powerful dialog system with dialogs that are isolated and composable
- Built-in prompts for simple things such as Yes/No, strings, numbers, and enumerations
- Built-in dialogs that utilize powerful AI frameworks such as [LUIS](#)
- FormFlow for automatically generating a bot (from a C# class) that guides the user through the conversation, providing help, navigation, clarification, and confirmation as needed

IMPORTANT

On July 31, 2017 breaking changes will be implemented in the Bot Framework security protocol. To prevent these changes from adversely impacting your bot, you must ensure that your application is using Bot Builder SDK v3.5 or greater. If you've built a bot using an SDK that you obtained prior to January 5, 2017 (the release date for Bot Builder SDK v3.5), be sure to upgrade to Bot Builder SDK v3.5 or later before July 31, 2017.

Get the SDK

The SDK is available as a NuGet package and as open source on [GitHub](#).

IMPORTANT

The Bot Builder SDK for .NET requires .NET Framework 4.6 or newer. If you are adding the SDK to an existing project targeting a lower version of the .NET Framework, you will need to update your project to target .NET Framework 4.6 first.

To install the SDK within a Visual Studio project, complete the following steps:

1. In **Solution Explorer**, right-click the project name and select **Manage NuGet Packages....**
2. On the **Browse** tab, type "Microsoft.Bot.Builder" into the search box.
3. Select **Microsoft.Bot.Builder** in the list of results, click **Install**, and accept the changes.

Get code samples

This SDK includes [sample source code](#) that uses the features of the Bot Builder SDK for .NET.

Next steps

Learn more about building bots using the Bot Builder SDK for .NET by reviewing articles throughout this section, beginning with:

- [Quickstart](#): Quickly build and test a simple bot by following instructions in this step-by-step tutorial.
- [Key concepts](#): Learn about key concepts in the Bot Builder SDK for .NET.

If you encounter problems or have suggestions regarding the Bot Builder SDK for .NET, see [Support](#) for a list of

available resources.

Key concepts in the Bot Builder SDK for .NET

11/2/2017 • 2 min to read • [Edit Online](#)

This article introduces key concepts in the Bot Builder SDK for .NET.

Connector

The [Bot Framework Connector](#) provides a single REST API that enables a bot to communicate across multiple channels such as Skype, Email, Slack, and more. It facilitates communication between bot and user by relaying messages from bot to channel and from channel to bot.

In the Bot Builder SDK for .NET, the [Connector](#) library enables access to the Connector.

Activity

The [Connector](#) uses an [Activity](#) object to pass information back and forth between bot and channel (user). The most common type of activity is **message**, but there are other activity types that can be used to communicate various types of information to a bot or channel.

For details about Activities in the Bot Builder SDK for .NET, see [Activities overview](#).

Dialog

When you create a bot using the Bot Builder SDK for .NET, you can use [dialogs](#) to model a conversation and manage [conversation flow](#). A dialog can be composed of other dialogs to maximize reuse, and a dialog context maintains the [stack of dialogs](#) that are active in the conversation at any point in time. A conversation that comprises dialogs is portable across computers, which makes it possible for your bot implementation to scale.

In the Bot Builder SDK for .NET, the [Builder](#) library enables you to manage dialogs.

FormFlow

You can use [FormFlow](#) within the Bot Builder SDK for .NET to streamline of building a bot that collects information from the user. For example, a bot that takes sandwich orders must collect several pieces of information from the user such as type of bread, choice of toppings, size, and so on. Given basic guidelines, FormFlow can automatically generate the dialogs necessary to manage a guided conversation like this.

State

The Bot Framework State service enables your bot to store and retrieve state data that is associated with a user, a conversation, or a specific user within the context of a specific conversation. State data can be used for many purposes, such as determining where the prior conversation left off or simply greeting a returning user by name. If you store a user's preferences, you can use that information to customize the conversation the next time you chat. For example, you might alert the user to a news article about a topic that interests her, or alert a user when an appointment becomes available.

For details about managing state using the Bot Builder SDK for .NET, see [Manage state data](#).

Naming conventions

The Bot Builder SDK for .NET library uses strongly-typed, Pascal-cased naming conventions. However, the JSON

messages that are transported back and forth over the wire use camel-case naming conventions. For example, the C# property **ReplyToId** is serialized as **replyToId** in the JSON message that's transported over the wire.

Security

You should ensure that your bot's endpoint can only be called by the Bot Framework Connector service. For more information on this topic, see [Secure your bot](#).

Next steps

Now you know the concepts behind every bot. You can quickly [build a bot using Visual Studio](#) using a template.

Next, study each key concept in more detail, starting with dialogs.

[Dialogs in the Bot Builder SDK for .NET](#)

Activities overview

11/2/2017 • 4 min to read • [Edit Online](#)

The [Connector](#) uses an [Activity](#) object to pass information back and forth between bot and channel (user). The most common type of activity is **message**, but there are other activity types that can be used to communicate various types of information to a bot or channel.

Activity types in the Bot Builder SDK for .NET

The following activity types are supported by the Bot Builder SDK for .NET.

ACTIVITY.TYPE	INTERFACE	DESCRIPTION
message	IMessageActivity	Represents a communication between bot and user.
conversationUpdate	IConversationUpdateActivity	Indicates that the bot was added to a conversation, other members were added to or removed from the conversation, or conversation metadata has changed.
contactRelationUpdate	IContactRelationUpdateActivity	Indicates that the bot was added or removed from a user's contact list.
typing	ITypingActivity	Indicates that the user or bot on the other end of the conversation is compiling a response.
ping	n/a	Represents an attempt to determine whether a bot's endpoint is accessible.
deleteUserData	n/a	Indicates to a bot that a user has requested that the bot delete any user data it may have stored.
endOfConversation	IEndOfConversationActivity	Indicates the end of a conversation.
event	IEventActivity	Represents a communication sent to a bot that is not visible to the user.
invoke	IInvokeActivity	Represents a communication sent to a bot to request that it perform a specific operation. This activity type is reserved for internal use by the Microsoft Bot Framework.
messageReaction	IMessageReactionActivity	Indicates that a user has reacted to an existing activity. For example, a user clicks the "Like" button on a message.

message

Your bot will send **message** activities to communicate information to and receive **message** activities from users. Some messages may simply consist of plain text, while others may contain richer content such as [text to be spoken](#), [suggested actions](#), [media attachments](#), [rich cards](#), and [channel-specific data](#). For information about commonly-used message properties, see [Create messages](#).

conversationUpdate

A bot receives a **conversationUpdate** activity whenever it has been added to a conversation, other members have been added to or removed from a conversation, or conversation metadata has changed.

If members have been added to the conversation, the activity's `MembersAdded` property will contain an array of `ChannelAccount` objects to identify the new members.

To determine whether your bot has been added to the conversation (i.e., is one of the new members), evaluate whether the `Recipient.Id` value for the activity (i.e., your bot's id) matches the `Id` property for any of the accounts in the `MembersAdded` array.

If members have been removed from the conversation, the `MembersRemoved` property will contain an array of `ChannelAccount` objects to identify the removed members.

TIP

If your bot receives a **conversationUpdate** activity indicating that a user has joined the conversation, you may choose to have it respond by sending a welcome message to that user.

contactRelationUpdate

A bot receives a **contactRelationUpdate** activity whenever it is added to or removed from a user's contact list. The value of the activity's `Action` property (add | remove) indicates whether the bot has been added or removed from the user's contact list.

typing

A bot receives a **typing** activity to indicate that the user is typing a response. A bot may send a **typing** activity to indicate to the user that it is working to fulfill a request or compile a response.

ping

A bot receives a **ping** activity to determine whether its endpoint is accessible. The bot should respond with HTTP status code 200 (OK), 403 (Forbidden), or 401 (Unauthorized).

deleteUserData

A bot receives a **deleteUserData** activity when a user requests deletion of any data that the bot has previously persisted for him or her. If your bot receives this type of activity, it should delete any personally identifiable information (PII) that it has previously stored for the user that made the request.

endOfConversation

A bot receives an **endOfConversation** activity to indicate that the user has ended the conversation. A bot may send an **endOfConversation** activity to indicate to the user that the conversation is ending.

event

Your bot may receive an **event** activity from an external process or service that wants to communicate information to your bot without that information being visible to users. The sender of an **event** activity typically does not expect the bot to acknowledge receipt in any way.

invoke

Your bot may receive an **invoke** activity that represents a request for it to perform a specific operation. The sender of an **invoke** activity typically expects the bot to acknowledge receipt via HTTP response. This activity type is reserved for internal use by the Microsoft Bot Framework.

messageReaction

Some channels will send **messageReaction** activities to your bot when a user reacted to an existing activity. For example, a user clicks the "Like" button on a message. The **ReplyToId** property will indicate which activity the user reacted to.

The **messageReaction** activity may correspond to any number of **messageReactionTypes** that the channel defined. For example, "Like" or "PlusOne" as reaction types that a channel may send.

Additional resources

- [Send and receive activities](#)
- [Create messages](#)
- [Activity class](#)

Create messages

11/2/2017 • 4 min to read • [Edit Online](#)

Your bot will send **message activities** to communicate information to users, and likewise, will also receive **message** activities from users. Some messages may simply consist of plain text, while others may contain richer content such as [text to be spoken](#), [suggested actions](#), [media attachments](#), [rich cards](#), and [channel-specific data](#).

This article describes some of the commonly-used message properties.

Customizing a message

To have more control over the text formatting of your messages, you can create a custom message using the [Activity](#) object and set the properties necessary before sending it to the user.

This sample shows how to create a custom `message` object and set the `Text`, `TextFormat`, and `Locale` properties.

```
IMessageActivity message = Activity.CreateMessageActivity();
message.Text = "Hello!";
message.TextFormat = "plain";
message.Locale = "en-US";
```

The `TextFormat` property of a message can be used to specify the format of the text. The `TextFormat` property can be set to **plain**, **markdown**, or **xml**. The default value for `TextFormat` is **markdown**.

For a list of commonly supported text formatting, see [Text formatting](#). To ensure that the feature(s) you want to use is supported by the target channel, preview the feature(s) using the [Channel Inspector](#).

Attachments

The `Attachments` property of a message activity can be used to send and receive simple media attachments (image, audio, video, file) and rich cards. For details, see [Add media attachments to messages](#) and [Add rich cards to messages](#).

Entities

The `Entities` property of a message is an array of open-ended [schema.org](#) objects which allows the exchange of common contextual metadata between the channel and bot.

Mention entities

Many channels support the ability for a bot or user to "mention" someone within the context of a conversation. To mention a user in a message, populate the message's `Entities` property with a `Mention` object. The `Mention` object contains these properties:

PROPERTY	DESCRIPTION
Type	type of the entity ("mention")
Mentioned	<code>ChannelAccount</code> object that indicates which user was mentioned

PROPERTY	DESCRIPTION
Text	text within the <code>Activity.Text</code> property that represents the mention itself (may be empty or null)

This code example shows how to add a `Mention` entity to the `Entities` collection.

```
var entity = new Entity();
entity.SetAs(new Mention())
{
    Text = "@johndoe",
    Mentioned = new ChannelAccount()
    {
        Name = "John Doe",
        Id = "UV341235"
    }
);
entities.Add(entity);
```

TIP

When attempting to determine user intent, the bot may want to ignore that portion of the message where it is mentioned. Call the `GetMentions` method and evaluate the `Mention` objects returned in the response.

Place objects

Location-related information can be conveyed within a message by populating the message's `Entities` property with either a `Place` object or a `GeoCoordinates` object.

The `Place` object contains these properties:

PROPERTY	DESCRIPTION
Type	type of the entity ("Place")
Address	description or <code>PostalAddress</code> object (future)
Geo	<code>GeoCoordinates</code>
HasMap	URL to a map or <code>Map</code> object (future)
Name	name of the place

The `GeoCoordinates` object contains these properties:

PROPERTY	DESCRIPTION
Type	type of the entity ("GeoCoordinates")
Name	name of the place
Longitude	longitude of the location (WGS 84)
Latitude	latitude of the location (WGS 84)

PROPERTY	DESCRIPTION
Elevation	elevation of the location (WGS 84)

This code example shows how to add a `Place` entity to the `Entities` collection:

```
var entity = new Entity();
entity.SetAs(new Place())
{
    Geo = new GeoCoordinates()
    {
        Latitude = 32.4141,
        Longitude = 43.1123123,
    }
});
entities.Add(entity);
```

Consume entities

To consume entities, use either the `dynamic` keyword or strongly-typed classes.

This code example shows how to use the `dynamic` keyword to process an entity within the `Entities` property of a message:

```
if (entity.Type == "Place")
{
    dynamic place = entity.Properties;
    if (place.geo.latitude > 34)
        // do something
}
```

This code example shows how to use a strongly-typed class to process an entity within the `Entities` property of a message:

```
if (entity.Type == "Place")
{
    Place place = entity.GetAs<Place>();
    GeoCoordinates geo = place.Geo.ToObject<GeoCoordinates>();
    if (geo.Latitude > 34)
        // do something
}
```

Channel data

The `channelData` property of a message activity can be used to implement channel-specific functionality. For details, see [Implement channel-specific functionality](#).

Text to speech

The `speak` property of a message activity can be used to specify the text to be spoken by your bot on a speech-enabled channel. The `InputHint` property of a message activity can be used to control the state of the client's microphone and input box (if any). For details, see [Add speech to messages](#).

Suggested actions

The `SuggestedActions` property of a message activity can be used to present buttons that the user can tap to

provide input. Unlike buttons that appear within rich cards (which remain visible and accessible to the user even after being tapped), buttons that appear within the suggested actions pane will disappear after the user makes a selection. For details, see [Add suggested actions to messages](#).

Next steps

A bot and a user can send messages to each other. When the message is more complex, your bot can send a rich card in a message to the user. Rich cards cover many presentation and interaction scenarios commonly needed in most bots.

[Send a rich card in a message](#)

Additional resources

- [Activities overview](#)
- [Send and receive activities](#)
- [Add media attachments to messages](#)
- [Add rich cards to messages](#)
- [Add speech to messages](#)
- [Add suggested actions to messages](#)
- [Implement channel-specific functionality](#)
- [Activity class](#)
- [IMessageActivity interface](#)

Add media attachments to messages

11/2/2017 • 1 min to read • [Edit Online](#)

A message exchange between user and bot can contain media attachments (e.g., image, video, audio, file). The `Attachments` property of the [Activity](#) object contains an array of [Attachment](#) objects that represent the media attachments and rich cards within to the message.

NOTE

For information about how to add rich cards to messages, see [Add rich cards to messages](#).

Add a media attachment

To add a media attachment to a message, create an `Attachment` object for the `message` activity and set the `ContentType`, `ContentUrl`, and `Name` properties.

```
replyMessage.Attachments.Add(new Attachment()
{
    ContentUrl = "https://upload.wikimedia.org/wikipedia/en/a/a6/Bender_Rodriguez.png",
    ContentType = "image/png",
    Name = "Bender_Rodriguez.png"
});
```

If an attachment is an image, audio, or video, the Connector service will communicate attachment data to the channel in a way that enables the [channel](#) to render that attachment within the conversation. If the attachment is a file, the file URL will be rendered as a hyperlink within the conversation.

Additional resources

- [Preview features with the Channel Inspector](#)
- [Activities overview](#)
- [Create messages](#)
- [Add rich cards to messages](#)
- [Activity class](#)
- [Attachment class](#)

Add rich card attachments to messages

11/2/2017 • 7 min to read • [Edit Online](#)

A message exchange between user and bot can contain one or more rich cards rendered as a list or carousel.

The `Attachments` property of the [Activity](#) object contains an array of [Attachment](#) objects that represent the rich cards and media attachments within the message.

NOTE

For information about how to add media attachments to messages, see [Add media attachments to messages](#).

Types of rich cards

The Bot Framework currently supports eight types of rich cards:

CARD TYPE	DESCRIPTION
Adaptive Card	A customizable card that can contain any combination of text, speech, images, buttons, and input fields. See per-channel support .
Animation Card	A card that can play animated GIFs or short videos.
Audio Card	A card that can play an audio file.
Hero Card	A card that typically contains a single large image, one or more buttons, and text.
Thumbnail Card	A card that typically contains a single thumbnail image, one or more buttons, and text.
Receipt Card	A card that enables a bot to provide a receipt to the user. It typically contains the list of items to include on the receipt, tax and total information, and other 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.
Video Card	A card that can play videos.

TIP

To display multiple rich cards in list format, set the activity's `AttachmentLayout` property to "list". To display multiple rich cards in carousel format, set the activity's `AttachmentLayout` property to "carousel". If the channel does not support carousel format, it will display the rich cards in list format, even if the `AttachmentLayout` property specifies "carousel".

Process events within rich cards

To process events within rich cards, define `CardAction` objects to specify what should happen when the user clicks a button or taps a section of the card. Each `CardAction` object contains these properties:

PROPERTY	TYPE	DESCRIPTION
Type	string	type of action (one of the values specified in the table below)
Title	string	title of the button
Image	string	image URL for the button
Value	string	value needed to perform the specified type of action

NOTE

Buttons within Adaptive Cards are not created using `CardAction` objects, but instead using the schema that is defined by [Adaptive Cards](#). See [Add an Adaptive Card to a message](#) for an example that shows how to add buttons to an Adaptive Card.

This table lists the valid values for `CardAction.Type` and describes the expected contents of `CardAction.Value` for each type:

CARDACTION.TYPE	CARDACTION.VALUE
openUrl	URL to be opened in the built-in browser
imBack	Text of the message to send to the bot (from the user who clicked the button or tapped the card). This message (from user to bot) will be visible to all conversation participants via the client application that is hosting the conversation.
postBack	Text of the message to send to the bot (from the user who clicked the button or tapped the card). Some client applications may display this text in the message feed, where it will be visible to all conversation participants.
call	Destination for a phone call in this format: tel:123123123123
playAudio	URL of audio to be played
playVideo	URL of video to be played
showImage	URL of image to be displayed
downloadFile	URL of file to be downloaded
signin	URL of OAuth flow to be initiated

Add a Hero card to a message

The Hero card typically contains a single large image, one or more buttons, and text.

This code example shows how to create a reply message that contains three Hero cards rendered in carousel format:

```
Activity replyToConversation = message.CreateReply("Should go to conversation, in carousel format");
replyToConversation.AttachmentLayout = AttachmentLayoutTypes.Carousel;
replyToConversation.Attachments = new List<Attachment>();

Dictionary<string, string> cardContentList = new Dictionary<string, string>();
cardContentList.Add("PigLatin", "https://<ImageUrl1>");
cardContentList.Add("Pork Shoulder", "https://<ImageUrl2>");
cardContentList.Add("Bacon", "https://<ImageUrl3>");

foreach(KeyValuePair<string, string> cardContent in cardContentList)
{
    List<CardImage> cardImages = new List<CardImage>();
    cardImages.Add(new CardImage(url:cardContent.Value ));

    List<CardAction> cardButtons = new List<CardAction>();

    CardAction plButton = new CardAction()
    {
        Value = $"https://en.wikipedia.org/wiki/{cardContent.Key}",
        Type = "openUrl",
        Title = "WikiPedia Page"
    };

    cardButtons.Add(plButton);

    HeroCard plCard = new HeroCard()
    {
        Title = $"I'm a hero card about {cardContent.Key}",
        Subtitle = $"{cardContent.Key} Wikipedia Page",
        Images = cardImages,
        Buttons = cardButtons
    };

    Attachment plAttachment = plCard.ToAttachment();
    replyToConversation.Attachments.Add(plAttachment);
}

var reply = await connector.Conversations.SendToConversationAsync(replyToConversation);
```

Add a Thumbnail card to a message

The Thumbnail card typically contains a single thumbnail image, one or more buttons, and text.

This code example shows how to create a reply message that contains two Thumbnail cards rendered in list format:

```

Activity replyToConversation = message.CreateReply("Should go to conversation, in list format");
replyToConversation.AttachmentLayout = AttachmentLayoutTypes.List;
replyToConversation.Attachments = new List<Attachment>();

Dictionary<string, string> cardContentList = new Dictionary<string, string>();
cardContentList.Add("PigLatin", "https://<ImageUrl1>");
cardContentList.Add("Pork Shoulder", "https://<ImageUrl2>");

foreach(KeyValuePair<string, string> cardContent in cardContentList)
{
    List<CardImage> cardImages = new List<CardImage>();
    cardImages.Add(new CardImage(url:cardContent.Value ));

    List<CardAction> cardButtons = new List<CardAction>();

    CardAction plButton = new CardAction()
    {
        Value = $"https://en.wikipedia.org/wiki/{cardContent.Key}",
        Type = "openUrl",
        Title = "WikiPedia Page"
    };

    cardButtons.Add(plButton);

    ThumbnailCard plCard = new ThumbnailCard()
    {
        Title = $"I'm a thumbnail card about {cardContent.Key}",
        Subtitle = $"{cardContent.Key} Wikipedia Page",
        Images = cardImages,
        Buttons = cardButtons
    };

    Attachment plAttachment = plCard.ToAttachment();
    replyToConversation.Attachments.Add(plAttachment);
}

var reply = await connector.Conversations.SendToConversationAsync(replyToConversation);

```

Add a Receipt card to a message

The Receipt card enables a bot to provide a receipt to the user. It typically contains the list of items to include on the receipt, tax and total information, and other text.

This code example shows how to create a reply message that contains a Receipt card:

```

Activity replyToConversation = message.CreateReply("Should go to conversation");
replyToConversation.Attachments = new List<Attachment>();

List<CardImage> cardImages = new List<CardImage>();
cardImages.Add(new CardImage(url: "https://<imageUrl1>" ));

List<CardAction> cardButtons = new List<CardAction>();

CardAction plButton = new CardAction()
{
    Value = $"https://en.wikipedia.org/wiki/PigLatin",
    Type = "openUrl",
    Title = "WikiPedia Page"
};

cardButtons.Add(plButton);

ReceiptItem lineItem1 = new ReceiptItem()
{
    Title = "Pork Shoulder",
    Subtitle = "8 lbs",
    Text = null,
    Image = new CardImage(url: "https://<ImageUrl1>"),
    Price = "16.25",
    Quantity = "1",
    Tap = null
};

ReceiptItem lineItem2 = new ReceiptItem()
{
    Title = "Bacon",
    Subtitle = "5 lbs",
    Text = null,
    Image = new CardImage(url: "https://<ImageUrl2>"),
    Price = "34.50",
    Quantity = "2",
    Tap = null
};

List<ReceiptItem> receiptList = new List<ReceiptItem>();
receiptList.Add(lineItem1);
receiptList.Add(lineItem2);

ReceiptCard plCard = new ReceiptCard()
{
    Title = "I'm a receipt card, isn't this bacon expensive?",
    Buttons = cardButtons,
    Items = receiptList,
    Total = "112.77",
    Tax = "27.52"
};

Attachment plAttachment = plCard.ToAttachment();
replyToConversation.Attachments.Add(plAttachment);

var reply = await connector.Conversations.SendToConversationAsync(replyToConversation);

```

Add a Sign-in card to a message

The Sign-in card 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.

This code example shows how to create a reply message that contains a Sign-in card:

```

Activity replyToConversation = message.CreateReply("Should go to conversation");
replyToConversation.Attachments = new List<Attachment>();

List<CardAction> cardButtons = new List<CardAction>();

CardAction plButton = new CardAction()
{
    Value = $"https://<OAuthSignInURL",
    Type = "signin",
    Title = "Connect"
};

cardButtons.Add(plButton);

SigninCard plCard = new SigninCard(title: "You need to authorize me", button: plButton);

Attachment plAttachment = plCard.ToAttachment();
replyToConversation.Attachments.Add(plAttachment);

var reply = await connector.Conversations.SendToConversationAsync(replyToConversation);

```

Add an Adaptive card to a message

The Adaptive Card can contain any combination of text, speech, images, buttons, and input fields. Adaptive Cards are created using the JSON format specified in [Adaptive Cards](#), which gives you full control over card content and format.

To create an Adaptive Card using .NET, install the [Microsoft.AdaptiveCards](#) NuGet package. Then, leverage the information within the [Adaptive Cards](#) site to understand Adaptive Card schema, explore Adaptive Card elements, and see JSON samples that can be used to create cards of varying composition and complexity. Additionally, you can use the Interactive Visualizer to design Adaptive Card payloads and preview card output.

This code example shows how to create a message that contains an Adaptive Card for a calendar reminder:

```

Activity replyToConversation = message.CreateReply("Should go to conversation");
replyToConversation.Attachments = new List<Attachment>();

AdaptiveCard card = new AdaptiveCard();

// Specify speech for the card.
card.Speak = "<s>Your meeting about \"Adaptive Card design session\"<break strength='weak'> is starting at 12:30pm</s><s>Do you want to snooze <break strength='weak'> or do you want to send a late notification to the attendees?</s>";

// Add text to the card.
card.Body.Add(new TextBlock()
{
    Text = "Adaptive Card design session",
    Size = TextSize.Large,
    Weight = TextWeight.Bolder
});

// Add text to the card.
card.Body.Add(new TextBlock()
{
    Text = "Conf Room 112/3377 (10)"
});

// Add text to the card.
card.Body.Add(new TextBlock()
{
    Text = "12:30 PM - 1:30 PM"
});

```

```

// Add list of choices to the card.
card.Body.Add(new ChoiceSet()
{
    Id = "snooze",
    Style = ChoiceInputStyle.Compact,
    Choices = new List<Choice>()
    {
        new Choice() { Title = "5 minutes", Value = "5", IsSelected = true },
        new Choice() { Title = "15 minutes", Value = "15" },
        new Choice() { Title = "30 minutes", Value = "30" }
    }
});

// Add buttons to the card.
card.Actions.Add(new HttpAction()
{
    Url = "http://foo.com",
    Title = "Snooze"
});

card.Actions.Add(new HttpAction()
{
    Url = "http://foo.com",
    Title = "I'll be late"
});

card.Actions.Add(new HttpAction()
{
    Url = "http://foo.com",
    Title = "Dismiss"
});

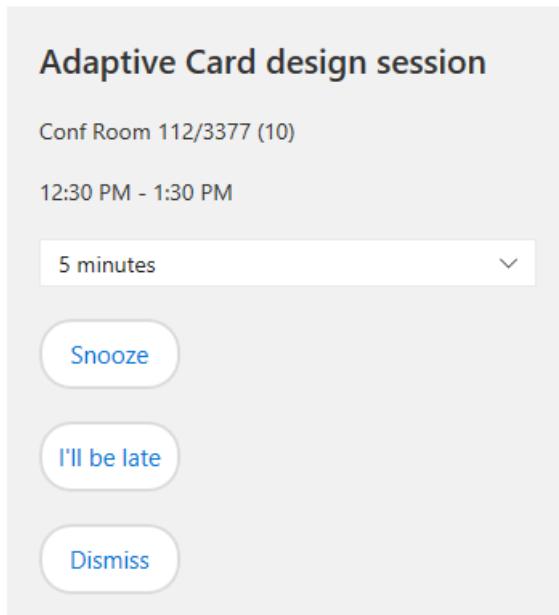
// Create the attachment.
Attachment attachment = new Attachment()
{
    ContentType = AdaptiveCard.ContentType,
    Content = card
};

replyToConversation.Attachments.Add(attachment);

var reply = await connector.Conversations.SendToConversationAsync(replyToConversation);

```

The resulting card contains three blocks of text, an input field (choice list), and three buttons:



Additional resources

- [Preview features with the Channel Inspector](#)
- [Adaptive Cards](#)
- [Activities overview](#)
- [Create messages](#)
- [Add media attachments to messages](#)
- [Activity class](#)
- [Attachment class](#)

Add speech to messages

11/2/2017 • 2 min to read • [Edit Online](#)

If you are building a bot for a speech-enabled channel such as Cortana, you can construct messages that specify the text to be spoken by your bot. You can also attempt to influence the state of the client's microphone by specifying an [input hint](#) to indicate whether your bot is accepting, expecting, or ignoring user input.

Specify text to be spoken by your bot

Using the Bot Builder SDK for .NET, there are multiple ways to specify the text to be spoken by your bot on a speech-enabled channel. You can set the `Speak` property of the [message](#), call the `IDialogContext.SayAsync()` method, or specify prompt options `speak` and `retrySpeak` when sending a message using a built-in prompt.

IMessageActivity.Speak

If you are creating a [message](#) and setting its individual properties, you can set the `Speak` property of the message to specify the text to be spoken by your bot. The following code example creates a message that specifies text to be displayed and text to be spoken and indicates that the bot is [accepting user input](#).

```
Activity reply = activity.CreateReply("This is the text that will be displayed.");
reply.Speak = "This is the text that will be spoken.";
reply.InputHint = InputHints.AcceptingInput;
await connector.Conversations.ReplyToActivityAsync(reply);
```

IDialogContext.SayAsync()

If you are using [dialogs](#), you can call the `SayAsync()` method to create and send a message that specifies the text to be spoken, in addition to the text to be displayed and other options. The following code example creates a message that specifies text to be displayed and text to be spoken.

```
await context.SayAsync(text: "Thank you for your order!", speak: "Thank you for your order!");
```

Prompt options

Using any of the built-in prompts, you can set the options `speak` and `retrySpeak` to specify the text to be spoken by your bot. The following code example creates a prompt that specifies text to be displayed, text to be spoken initially, and text to be spoken after waiting a while for user input. It uses [SSML](#) formatting to indicate that the word "sure" should be spoken with a moderate amount of emphasis.

```
PromptDialog.Confirm(context, AfterResetAsync,
    new PromptOptions<string>(prompt: "Are you sure that you want to cancel this transaction?",
        speak: "Are you <emphasis level=\"moderate\\\">sure</emphasis> that you want to cancel this
transaction?",  
        retrySpeak: "Are you <emphasis level=\"moderate\\\">sure</emphasis> that you want to cancel this
transaction?"));
```

Speech Synthesis Markup Language (SSML)

To specify text to be spoken by your bot, you can use either a plain text string or a string that is formatted as Speech Synthesis Markup Language (SSML), an XML-based markup language that enables you to control various characteristics of your bot's speech such as voice, rate, volume, pronunciation, pitch, and more. For details about

SSML, see [Speech Synthesis Markup Language Reference](#).

Input hints

When you send a message on a speech-enabled channel, you can attempt to influence the state of the client's microphone by also including an input hint to indicate whether your bot is accepting, expecting, or ignoring user input. For more information, see [Add input hints to messages](#).

Sample code

For a complete sample that shows how to create a speech-enabled bot using the Bot Builder SDK for .NET, see the [Roller Skill sample](#) in GitHub.

Additional resources

- [Create messages](#)
- [Add input hints to messages](#)
- [Speech Synthesis Markup Language \(SSML\)](#)
- [Roller Skill sample \(GitHub\)](#)
- [Activity class](#)
- [IMessageActivity interface](#)
- [DialogContext class](#)
- [Prompt class](#)

Add input hints to messages

11/2/2017 • 2 min to read • [Edit Online](#)

By specifying an input hint for a message, you can indicate whether your bot is accepting, expecting, or ignoring user input after the message is delivered to the client. For many channels, this enables clients to set the state of user input controls accordingly. For example, if a message's input hint indicates that the bot is ignoring user input, the client may close the microphone and disable the input box to prevent the user from providing input.

Accepting input

To indicate that your bot is passively ready for input but is not awaiting a response from the user, set the message's input hint to `InputHints.AcceptingInput`. On many channels, this will cause the client's input box to be enabled and microphone to be closed, but still accessible to the user. For example, Cortana will open the microphone to accept input from the user if the user holds down the microphone button. The following code example creates a message that indicates the bot is accepting user input.

```
Activity reply = activity.CreateReply("This is the text that will be displayed.");
reply.Speak = "This is the text that will be spoken.";
reply.InputHint = InputHints.AcceptingInput;
await connector.Conversations.ReplyToActivityAsync(reply);
```

Expecting input

To indicate that your bot is awaiting a response from the user, set the message's input hint to `InputHints.ExpectingInput`. On many channels, this will cause the client's input box to be enabled and microphone to be open. The following code example creates a message that indicates the bot is expecting user input.

```
Activity reply = activity.CreateReply("This is the text that will be displayed.");
reply.Speak = "This is the text that will be spoken.";
reply.InputHint = InputHints.ExpectingInput;
```

Ignoring input

To indicate that your bot is not ready to receive input from the user, set the message's input hint to `InputHints.IgnoringInput`. On many channels, this will cause the client's input box to be disabled and microphone to be closed. The following code example creates a message that indicates the bot is ignoring user input.

```
Activity reply = activity.CreateReply("This is the text that will be displayed.");
reply.Speak = "This is the text that will be spoken.";
reply.InputHint = InputHints.IgnoringInput;
```

Default values for input hint

If you do not set the input hint for a message, the Bot Builder SDK will automatically set it for you by using this logic:

- If your bot sends a prompt, the input hint for the message will specify that your bot is **expecting input**.

- If your bot sends single message, the input hint for the message will specify that your bot is **accepting input**.
- If your bot sends a series of consecutive messages, the input hint for all but the final message in the series will specify that your bot is **ignoring input**, and the input hint for the final message in the series will specify that your bot is **accepting input**.

Additional resources

- [Create messages](#)
- [Add speech to messages](#)
- [Activity class](#)
- [InputHints class](#)

Add suggested actions to messages

11/2/2017 • 1 min to read • [Edit Online](#)

Suggested actions enable your bot to present buttons that the user can tap to provide input. Suggested actions appear close to the composer and enhance user experience by enabling the user to answer a question or make a selection with a simple tap of a button, rather than having to type a response with a keyboard. Unlike buttons that appear within rich cards (which remain visible and accessible to the user even after being tapped), buttons that appear within the suggested actions pane will disappear after the user makes a selection. This prevents the user from tapping stale buttons within a conversation and simplifies bot development (since you will not need to account for that scenario).

TIP

Use the [Channel Inspector](#) to see how suggested actions look and work on various channels.

Send suggested actions

To add suggested actions to a message, set the `SuggestedActions` property of the activity to a list of [CardAction](#) objects that represent the buttons to be presented to the user.

This code example shows how to create a message that presents three suggested actions to the user:

```
var reply = activity.CreateReply("I have colors in mind, but need your help to choose the best one.");
reply.Type = ActivityTypes.Message;
reply.TextFormat = TextFormatTypes.Plain;

reply.SuggestedActions = new SuggestedActions()
{
    Actions = new List<CardAction>()
    {
        new CardAction(){ Title = "Blue", Type=ActionTypes.ImBack, Value="Blue" },
        new CardAction(){ Title = "Red", Type=ActionTypes.ImBack, Value="Red" },
        new CardAction(){ Title = "Green", Type=ActionTypes.ImBack, Value="Green" }
    }
};
```

When the user taps one of the suggested actions, the bot will receive a message from the user that contains the `Value` of the corresponding action.

Additional resources

- [Preview features with the Channel Inspector](#)
- [Activities overview](#)
- [Create messages](#)
- [Activity class](#)
- [IMessageActivity interface](#)
- [CardAction class](#)
- [SuggestedActions class](#)

Implement channel-specific functionality

11/2/2017 • 5 min to read • [Edit Online](#)

Some channels provide features that cannot be implemented by using only [message text and attachments](#). To implement channel-specific functionality, you can pass native metadata to a channel in the `Activity` object's `ChannelData` property. For example, your bot can use the `ChannelData` property to instruct Telegram to send a sticker or to instruct Office365 to send an email.

This article describes how to use a message activity's `ChannelData` property to implement this channel-specific functionality:

CHANNEL	FUNCTIONALITY
Email	Send and receive an email that contains body, subject, and importance metadata
Slack	Send full fidelity Slack messages
Facebook	Send Facebook notifications natively
Telegram	Perform Telegram-specific actions, such as sharing a voice memo or a sticker
Kik	Send and receive native Kik messages

NOTE

The value of an `Activity` object's `ChannelData` property is a JSON object. Therefore, the examples in this article show the expected format of the `channelData` JSON property in various scenarios. To create a JSON object using .NET, use the `JObject` (.NET) class.

Create a custom Email message

To create an email message, set the `Activity` object's `channelData` property to a JSON object that contains these properties:

PROPERTY	DESCRIPTION
<code>htmlBody</code>	An HTML document that specifies the body of the email message. See the channel's documentation for information about supported HTML elements and attributes.
<code>importance</code>	The email's importance level. Valid values are high , normal , and low . The default value is normal .
<code>subject</code>	The email's subject. See the channel's documentation for information about field requirements.

NOTE

Messages that your bot receives from users via the Email channel may contain a `channelData` property that is populated with a JSON object like the one described above.

This snippet shows an example of the `channelData` property for a custom email message.

```
"channelData": {  
    "htmlBody" : "<html><body style=\"font-family: Calibri; font-size: 11pt;\">This is the email body!</body>  
    </html>",  
    "subject":"This is the email subject",  
    "importance":"high"  
}
```

Create a full-fidelity Slack message

To create a full-fidelity Slack message, set the `Activity` object's `channelData` property to a JSON object that specifies [Slack messages](#), [Slack attachments](#), and/or [Slack buttons](#).

NOTE

To support buttons in Slack messages, you must enable **Interactive Messages** when you [connect your bot](#) to the Slack channel.

This snippet shows an example of the `channelData` property for a custom Slack message.

```

"channelData": {
    "text": "Now back in stock! :tada:",
    "attachments": [
        {
            "title": "The Further Adventures of Slackbot",
            "author_name": "Stanford S. Strickland",
            "author_icon": "https://api.slack.com/img/api/homepage_custom_integrations-2x.png",
            "image_url": "http://i.imgur.com/OJkaVOI.jpg?1"
        },
        {
            "fields": [
                {
                    "title": "Volume",
                    "value": "1",
                    "short": true
                },
                {
                    "title": "Issue",
                    "value": "3",
                    "short": true
                }
            ]
        },
        {
            "title": "Synopsis",
            "text": "After @episod pushed exciting changes to a devious new branch back in Issue 1, Slackbot notifies @don about an unexpected deploy..."
        },
        {
            "fallback": "Would you recommend it to customers?",
            "title": "Would you recommend it to customers?",
            "callback_id": "comic_1234_xyz",
            "color": "#3AA3E3",
            "attachment_type": "default",
            "actions": [
                {
                    "name": "recommend",
                    "text": "Recommend",
                    "type": "button",
                    "value": "recommend"
                },
                {
                    "name": "no",
                    "text": "No",
                    "type": "button",
                    "value": "bad"
                }
            ]
        }
    ]
}

```

When a user clicks a button within a Slack message, your bot will receive a response message in which the `ChannelData` property is populated with a `payload` JSON object. The `payload` object specifies contents of the original message, identifies the button that was clicked, and identifies the user who clicked the button.

This snippet shows an example of the `channelData` property in the message that a bot receives when a user clicks a button in the Slack message.

```

"channelData": {
    "payload": {
        "actions": [
            {
                "name": "recommend",
                "value": "yes"
            }
        ],
        ...
    },
    "original_message": "...",
    "response_url": "https://hooks.slack.com/actions/..."
}
}

```

Your bot can reply to this message in the [normal manner](#), or it can post its response directly to the endpoint that is specified by the `payload` object's `response_url` property. For information about when and how to post a response to the `response_url`, see [Slack Buttons](#).

Create a Facebook notification

To create a Facebook notification, set the `Activity` object's `ChannelData` property to a JSON object that specifies these properties:

PROPERTY	DESCRIPTION
<code>notification_type</code>	The type of notification (e.g., REGULAR , SILENT_PUSH , NO_PUSH).
<code>attachment</code>	An attachment that specifies an image, video, or other multimedia type, or a templated attachment such as a receipt.

NOTE

For details about format and contents of the `notification_type` property and `attachment` property, see the [Facebook API documentation](#).

This snippet shows an example of the `channelData` property for a Facebook receipt attachment.

```

"channelData": {
    "notification_type": "NO_PUSH",
    "attachment": {
        "type": "template",
        "payload": {
            "template_type": "receipt",
            ...
        }
    }
}

```

Create a Telegram message

To create a message that implements Telegram-specific actions, such as sharing a voice memo or a sticker, set the `Activity` object's `ChannelData` property to a JSON object that specifies these properties:

PROPERTY	DESCRIPTION
method	The Telegram Bot API method to call.
parameters	The parameters of the specified method.

These Telegram methods are supported:

- answerInlineQuery
- editMessageCaption
- editMessageReplyMarkup
- editMessageText
- forwardMessage
- kickChatMember
- sendAudio
- sendChatAction
- sendContact
- sendDocument
- sendLocation
- sendMessage
- sendPhoto
- sendSticker
- sendVenue
- sendVideo
- sendVoice
- unbanChateMember

For details about these Telegram methods and their parameters, see the [Telegram Bot API documentation](#).

NOTE

- The `chat_id` parameter is common to all Telegram methods. If you do not specify `chat_id` as a parameter, the framework will provide the ID for you.
- Instead of passing file contents inline, specify the file using a URL and media type as shown in the example below.
- Within each message that your bot receives from the Telegram channel, the `ChannelData` property will include the message that your bot sent previously.

This snippet shows an example of a `channelData` property that specifies a single Telegram method.

```
"channelData": {
    "method": "sendSticker",
    "parameters": {
        "sticker": {
            "url": "https://domain.com/path/gif",
            "mediaType": "image/gif",
        }
    }
}
```

This snippet shows an example of a `channelData` property that specifies an array of Telegram methods.

```

"channelData": [
    {
        "method": "sendSticker",
        "parameters": {
            "sticker": {
                "url": "https://domain.com/path/gif",
                "mediaType": "image/gif",
            }
        }
    },
    {
        "method": "sendMessage",
        "parameters": {
            "text": "<b>This message is HTML formatted.</b>",
            "parseMode": "HTML"
        }
    }
]

```

Create a native Kik message

To create a native Kik message, set the `Activity` object's `ChannelData` property to a JSON object that specifies this property:

PROPERTY	DESCRIPTION
messages	An array of Kik messages. For details about Kik message format, see Kik Message Formats .

This snippet shows an example of the `channelData` property for a native Kik message.

```

"channelData": {
    "messages": [
        {
            "chatId": "c6dd8165...",
            "type": "link",
            "to": "kikhandle",
            "title": "My Webpage",
            "text": "Some text to display",
            "url": "http://botframework.com",
            "picUrl": "http://lorempixel.com/400/200/",
            "attribution": {
                "name": "My App",
                "iconUrl": "http://lorempixel.com/50/50/"
            },
            "noForward": true,
            "kikJsData": {
                "key": "value"
            }
        }
    ]
}

```

Additional resources

- [Activities overview](#)
- [Create messages](#)
- [Activity class](#)

Send and receive activities

11/2/2017 • 4 min to read • [Edit Online](#)

The Bot Framework Connector provides a single REST API that enables a bot to communicate across multiple channels such as Skype, Email, Slack, and more. It facilitates communication between bot and user, by relaying messages from bot to channel and from channel to bot.

This article describes how to use the Connector via the Bot Builder SDK for .NET to exchange information between bot and user on a channel.

NOTE

While it is possible to construct a bot by exclusively using the techniques that are described in this article, the Bot Builder SDK provides additional features like [dialogs](#) and [FormFlow](#) that can streamline the process of managing conversation flow and state and make it simpler to incorporate cognitive services such as language understanding.

Create a connector client

The [ConnectorClient](#) class contains the methods that a bot uses to communicate with a user on a channel. When your bot receives an [Activity](#) object from the Connector, it should use the `ServiceUrl` specified for that activity to create the connector client that it'll subsequently use to generate a response.

```
public async Task<HttpResponseMessage> Post([FromBody]Activity activity)
{
    var connector = new ConnectorClient(new Uri(activity.ServiceUrl));
    . .
}
```

TIP

Because a channel's endpoint may not be stable, your bot should direct communications to the endpoint that the Connector specifies in the `Activity` object, whenever possible (rather than relying upon a cached endpoint).

If your bot needs to initiate the conversation, it can use a cached endpoint for the specified channel (since there will be no incoming `Activity` object in that scenario), but it should refresh cached endpoints often.

Create a reply

The Connector uses an [Activity](#) object to pass information back and forth between bot and channel (user). Every activity contains information used for routing the message to the appropriate destination along with information about who created the message (`From` property), the context of the message, and the recipient of the message (`Recipient` property).

When your bot receives an activity from the Connector, the incoming activity's `Recipient` property specifies the bot's identity in that conversation. Because some channels (e.g., Slack) assign the bot a new identity when it's added to a conversation, the bot should always use the value of the incoming activity's `Recipient` property as the value of the `From` property in its response.

Although you can create and initialize the outgoing `Activity` object yourself from scratch, the Bot Builder SDK provides an easier way of creating a reply. By using the incoming activity's `CreateReply` method, you simply

specify the message text for the response, and the outgoing activity is created with the `Recipient`, `From`, and `Conversation` properties automatically populated.

```
Activity reply = activity.CreateReply($"You sent {activity.Text} which was {length} characters");
```

Send a reply

Once you've created a reply, you can send it by calling the connector client's `ReplyToActivity` method. The Connector will deliver the reply using the appropriate channel semantics.

```
await connector.Conversations.ReplyToActivityAsync(reply);
```

TIP

If your bot is replying to a user's message, always use the `ReplyToActivity` method.

Send a (non-reply) message

If your bot is part of a conversation, it can send a message that is not a direct reply to any message from the user by calling the `SendToConversation` method.

```
await connector.Conversations.SendToConversationAsync((Activity)newMessage);
```

You may use the `CreateReply` method to initialize the new message (which would automatically set the `Recipient`, `From`, and `Conversation` properties for the message). Alternatively, you could use the `CreateMessageActivity` method to create the new message and set all property values yourself.

NOTE

The Bot Framework does not impose any restrictions on the number of messages that a bot may send. However, most channels enforce throttling limits to restrict bots from sending a large number of messages in a short period of time. Additionally, if the bot sends multiple messages in quick succession, the channel may not always render the messages in the proper sequence.

Start a conversation

There may be times when your bot needs to initiate a conversation with one or more users. You can start a conversation by calling either the `CreateDirectConversation` method (for a private conversation with a single user) or the `CreateConversation` method (for a group conversation with multiple users) to retrieve a `ConversationAccount` object. Then, create the message and send it by calling the `SendToConversation` method. To use either the `CreateDirectConversation` method or the `CreateConversation` method, you must first [create the connector client](#) by using the target channel's service URL (which you may retrieve from cache, if you've persisted it from previous messages).

NOTE

Not all channels support group conversations. To determine whether a channel supports group conversations, consult the channel's documentation.

This code example uses the `CreateDirectConversation` method to create a private conversation with a single user.

```
var userAccount = new ChannelAccount(name: "Larry", id: "@UV357341");
var connector = new ConnectorClient(new Uri(actvity.ServiceUrl));
var conversationId = await connector.Conversations.CreateDirectConversationAsync(botAccount, userAccount);

IMessageActivity message = Activity.CreateMessageActivity();
message.From = botAccount;
message.Recipient = userAccount;
message.Conversation = new ConversationAccount(id: conversationId.Id);
message.Text = "Hello, Larry!";
message.Locale = "en-US";
await connector.Conversations.SendToConversationAsync((Activity)message);
```

This code example uses the `CreateConversation` method to create a group conversation with multiple users.

```
var connector = new ConnectorClient(new Uri(incomingMessage.ServiceUrl));

List<ChannelAccount> participants = new List<ChannelAccount>();
participants.Add(new ChannelAccount("joe@contoso.com", "Joe the Engineer"));
participants.Add(new ChannelAccount("sara@contoso.com", "Sara in Finance"));

ConversationParameters cpMessage = new ConversationParameters(message.Recipient, true, participants, "Quarter
End Discussion");
var conversationId = await connector.Conversations.CreateConversationAsync(cpMessage);

IMessageActivity message = Activity.CreateMessageActivity();
message.From = botAccount;
message.Recipient = new ChannelAccount("lydia@contoso.com", "Lydia the CFO");
message.Conversation = new ConversationAccount(id: conversationId.Id);
message.ChannelId = incomingMessage.ChannelId;
message.Text = "Hello, everyone!";
message.Locale = "en-US";

await connector.Conversations.SendToConversationAsync((Activity)message);
```

Additional resources

- [Activities overview](#)
- [Create messages](#)
- [Bot Builder SDK for .NET Reference](#)
- [Activity class](#)
- [ConnectorClient class](#)

Implement global message handlers

11/2/2017 • 1 min to read • [Edit Online](#)

Users commonly attempt to access certain functionality within a bot by using keywords like "help", "cancel", or "start over". This often occurs in the middle of a conversation, when the bot is expecting a different response. By implementing **global message handlers**, you can design your bot to gracefully handle such requests. The handlers will examine user input for the keywords that you specify, such as "help", "cancel", or "start over", and respond appropriately.



Do you confirm this order?

Yes

No

Help?



I see you have questions, anything specific you want me to help you with?

NOTE

By defining the logic in global message handlers, you're making it accessible to all dialogs. Individual dialogs and prompts can be configured to safely ignore the keywords.

Listen for keywords in user input

The following walk through shows how to implement global message handlers by using the Bot Builder SDK for .NET.

First, `Global.asax.cs` registers `GlobalMessageHandlersBotModule`, which is implemented as shown here. In this example, the module registers two scorables: one for managing a request to change settings (`SettingsScorable`) and another for managing a request to cancel (`CancelScoreable`).

```

public class GlobalMessageHandlersBotModule : Module
{
    protected override void Load(ContainerBuilder builder)
    {
        base.Load(builder);

        builder
            .Register(c => new SettingsScorable(c.Resolve<IDialogTask>()))
            .As<IScorable<IActivity, double>>()
            .InstancePerLifetimeScope();

        builder
            .Register(c => new CancelScorable(c.Resolve<IDialogTask>()))
            .As<IScorable<IActivity, double>>()
            .InstancePerLifetimeScope();
    }
}

```

The `CancelScorable` contains a `PrepareAsync` method that defines the trigger: if the message text is "cancel", this scorables will be triggered.

```

protected override async Task<string> PrepareAsync(IActivity activity, CancellationToken token)
{
    var message = activity as IMessageActivity;
    if (message != null && !string.IsNullOrWhiteSpace(message.Text))
    {
        if (message.Text.Equals("cancel", StringComparison.InvariantCultureIgnoreCase))
        {
            return message.Text;
        }
    }
    return null;
}

```

When a "cancel" request is received, the `PostAsync` method within `CancelScoreable` resets the dialog stack.

```

protected override async Task PostAsync(IActivity item, string state, CancellationToken token)
{
    this.task.Reset();
}

```

When a "change settings" request is received, the `PostAsync` method within `SettingsScorable` invokes the `SettingsDialog` (passing the request to that dialog), thereby adding the `SettingsDialog` to the top of the dialog stack and putting it in control of the conversation.

```

protected override async Task PostAsync(IActivity item, string state, CancellationToken token)
{
    var message = item as IMessageActivity;
    if (message != null)
    {
        var settingsDialog = new SettingsDialog();
        var interruption = settingsDialog.Void<object, IMessageActivity>();
        this.task.Call(interruption, null);
        await this.task.PollAsync(token);
    }
}

```

Sample code

For a complete sample that shows how to implement global message handlers using the Bot Builder SDK for .NET, see the [Global Message Handlers sample](#) in GitHub.

Additional resources

- [Design and control conversation flow](#)
- [Bot Builder SDK for .NET Reference](#)
- [Global Message Handlers sample \(GitHub\)](#)

Intercept messages

11/2/2017 • 1 min to read • [Edit Online](#)

The **middleware** functionality in the Bot Builder SDK enables your bot to intercept all messages that are exchanged between user and bot. For each message that is intercepted, you may choose to do things such as save the message to a data store that you specify, which creates a conversation log, or inspect the message in some way and take whatever action your code specifies.

NOTE

The Bot Framework does not automatically save conversation details, as doing so could potentially capture private information that bots and users do not wish to share with outside parties. If your bot saves conversation details, it should communicate that to the user and describe what will be done with the data.

Intercept and log messages

The following code sample shows how to intercept messages that are exchanged between user and bot using the concept of **middleware** in the Bot Builder SDK for .NET.

First, create a `DebugActivityLogger` class and define a `LogAsync` method to specify what action is taken for each intercepted message. This example just prints some information about each message.

```
public class DebugActivityLogger : IActivityLogger
{
    public async Task LogAsync(IActivity activity)
    {
        Debug.WriteLine($"From:{activity.From.Id} - To:{activity.Recipient.Id} - Message:
{activity.AsMessageActivity()?.Text}");
    }
}
```

Then, add the following code to `Global.asax.cs`. Every message that is exchanged between user and bot (in either direction) will now trigger the `LogAsync` method in the `DebugActivityLogger` class.

```
public class WebApiApplication : System.Web.HttpApplication
{
    protected void Application_Start()
    {
        var builder = new ContainerBuilder();
        builder.RegisterType<DebugActivityLogger>().AsImplementedInterfaces().InstancePerDependency();
        builder.Update(Conversation.Container);

        GlobalConfiguration.Configure(WebApiConfig.Register);
    }
}
```

Although this example simply prints some information about each message, you can update the `LogAsync` method to specify the actions that you want to take for each message.

Sample code

For a complete sample that shows how to intercept and log messages using the Bot Builder SDK for .NET, see the

[Middleware sample](#) in GitHub.

Additional resources

- [Bot Builder SDK for .NET Reference](#)
- [Middleware sample \(GitHub\)](#)

Send proactive messages

11/2/2017 • 7 min to read • [Edit Online](#)

Typically, each message that a bot sends to the user directly relates to the user's prior input. In some cases, a bot may need to send the user a message that is not directly related to the current topic of conversation. These types of messages are called **proactive messages**.

Proactive messages can be useful in a variety of scenarios. If a bot sets a timer or reminder, it will need to notify the user when the time arrives. Or, if a bot receives a notification from an external system, it may need to communicate that information to the user immediately. For example, if the user has previously asked the bot to monitor the price of a product, the bot will alert the user if it receives notification that the price of the product has dropped by 20%. Or, if a bot requires some time to compile a response to the user's question, it may inform the user of the delay and allow the conversation to continue in the meantime. When the bot finishes compiling the response to the question, it will share that information with the user.

When implementing proactive messages in your bot:

Don't send several proactive messages within a short amount of time. Some channels enforce restrictions on how frequently a bot can send messages to the user, and will disable the bot if it violates those restrictions.

Don't send proactive messages to users who have not previously interacted with the bot or solicited contact with the bot through another means such as e-mail or SMS.

Consider the following scenario:



What city are you travelling to?

London



What is your planned date for the trip?



Good news! Your preferred hotel in Las Vegas is offering a discount! Want to book a trip?

Wait, what?



Sorry, this isn't a valid date for your London trip...

Bot you're drunk...

In this example, the user has previously asked the bot to monitor prices of a hotel in Las Vegas. The bot launched a background monitoring task, which has been running continuously for the past several days. In the current conversation, the user is booking a trip to London when the background task triggers a notification message about a discount for the Las Vegas hotel. The bot interjects

this information into the current conversation, making for a confusing user experience.

How should the bot have handled this situation?

- Wait for the current travel booking to finish, then deliver the notification. This approach would be minimally disruptive, but the delay in communicating the information might cause the user to miss out on the low-price opportunity for the Las Vegas hotel.
- Cancel the current travel booking flow and deliver the notification immediately. This approach delivers the information in a timely fashion but would likely frustrate the user by forcing them start over with their travel booking.
- Interrupt the current booking, clearly change the topic of conversation to the hotel in Las Vegas until the user responds, and then switch back to the in-progress travel booking and continue from where it was interrupted. This approach may seem like the best choice, but it introduces complexity both for the bot developer and the user.

Most commonly, your bot will use some combination of **ad hoc proactive messages** and **dialog-based proactive messages** to handle situations like this.

Types of proactive messages

An **ad hoc proactive message** is the simplest type of proactive message. The bot simply interjects the message into the conversation whenever it is triggered, without any regard for whether the user is currently engaged in a separate topic of conversation with the bot and will not attempt to change the conversation in any way.

A **dialog-based proactive message** is more complex than an ad hoc proactive message. Before it can inject this type of proactive message into the conversation, the bot must identify the context of the existing conversation and decide how (or if) it will resume that conversation after the message interrupts.

For example, consider a bot that needs to initiate a survey at a given point in time. When that time arrives, the bot stops the existing conversation with the user and redirects the user to a `SurveyDialog`. The `SurveyDialog` is added to the top of the dialog stack and takes control of the conversation. When the user finishes all required tasks at the `SurveyDialog`, the `SurveyDialog` closes, returning control to the previous dialog, where the user can continue with the prior topic of conversation.

A dialog-based proactive message is more than just simple notification. In sending the notification, the bot changes the topic of the existing conversation. It then must decide whether to resume that conversation later, or to abandon that conversation altogether by resetting the dialog stack.

Send an ad hoc proactive message

The following code samples show how to send an ad hoc proactive message with the Bot Builder SDK for .NET.

To be able to send an ad hoc message to a user, the bot must first collect and store some information about the user from the current conversation.

```

public virtual async Task MessageReceivedAsync(IDialogContext context, IAwaitable<IMessageActivity> result)
{
    var message = await result;

    // Extract data from the user's message that the bot will need later to send an ad hoc message to the
    user.

    // Store the extracted data in a custom class "ConversationStarter" (not shown here).

    ConversationStarter.toId = message.From.Id;
    ConversationStarter.toName = message.From.Name;
    ConversationStarter.fromId = message.Recipient.Id;
    ConversationStarter.fromName = message.Recipient.Name;
    ConversationStarter.serviceUrl = message.ServiceUrl;
    ConversationStarter.channelId = message.ChannelId;
    ConversationStarter.conversationId = message.Conversation.Id;

    // (Save this information somewhere that it can be accessed later, such as in a database.)

    await context.PostAsync("Hello user, good to meet you! I now know your address and can send you
notifications in the future.");
    context.Wait(MessageReceivedAsync);
}

```

NOTE

For simplicity, this example does not specify how to store the user data. It does not matter how the data is stored as long as the bot can retrieve it later.

Now that the data has been stored, the bot can simply retrieve the data, construct the ad hoc proactive message, and send it.

```

// Use the data stored previously to create the required objects.
var userAccount = new ChannelAccount(toId,toName);
var botAccount = new ChannelAccount(fromId, fromName);
var connector = new ConnectorClient(new Uri(serviceUrl));

// Create a new message.
IMessageActivity message = Activity.CreateMessageActivity();
if (!string.IsNullOrEmpty(conversationId) && !string.IsNullOrEmpty(channelId))
{
    // If conversation ID and channel ID was stored previously, use it.
    message.ChannelId = channelId;
}
else
{
    // Conversation ID was not stored previously, so create a conversation.
    // Note: If the user has an existing conversation in a channel, this will likely create a new conversation
    window.
    conversationId = (await connector.Conversations.CreateDirectConversationAsync( botAccount,
userAccount)).Id;
}

// Set the address-related properties in the message and send the message.
message.From = botAccount;
message.Recipient = userAccount;
message.Conversation = new ConversationAccount(id: conversationId);
message.Text = "Hello, this is a notification";
message.Locale = "en-us";
await connector.Conversations.SendToConversationAsync((Activity)message);

```

NOTE

If the bot specifies a conversation ID that was stored previously, the message will likely be delivered to the user in the existing conversation window on the client. If the bot generates a new conversation ID, the message will be delivered to the user in a new conversation window on the client, provided that the client supports multiple conversation windows.

Send a dialog-based proactive message

The following code samples show how to send a dialog-based proactive message by using the Bot Builder SDK for .NET.

To be able to send a dialog-based proactive message to a user, the bot must first collect and save information from the current conversation.

```
public virtual async Task MessageReceivedAsync(IDialogContext context, IAwaitable<IMessageActivity> result)
{
    var message = await result;

    // Store information about this specific point the conversation, so that the bot can resume this
    // conversation later.
    var conversationReference = message.ToConversationReference();
    ConversationStarter.conversationReference = JsonConvert.SerializeObject(conversationReference);

    await context.PostAsync("Greetings, user! I now know how to start a proactive message to you.");
    context.Wait(MessageReceivedAsync);
}
```

When it is time to send the message, the bot creates a new dialog and adds it to the top of the dialog stack. The new dialog takes control of the conversation, delivers the proactive message, closes, and then returns control to the previous dialog in the stack.

```
public static async Task Resume()
{
    // Recreate the message from the conversation reference that was saved previously.
    var message = JsonConvert.DeserializeObject<ConversationReference>(conversationReference).GetPostToBotMessage();
    var client = new ConnectorClient(new Uri(message.ServiceUrl));

    // Create a scope that can be used to work with state from bot framework.
    using (var scope = DialogModule.BeginLifetimeScope(Conversation.Container, message))
    {
        var botData = scope.Resolve<IBotData>();
        await botData.LoadAsync(CancellationToken.None);

        // This is the dialog stack.
        var stack = scope.Resolve<IDialogStack>();

        // Create the new dialog and add it to the stack.
        var dialog = new SurveyDialog();
        stack.Call(dialog.Void<object, IMessageActivity>(), null);
        await stack.PollAsync(CancellationToken.None);

        // Flush the dialog stack back to its state store.
        await botData.FlushAsync(CancellationToken.None);
    }
}
```

The `SurveyDialog` controls the conversation until it finishes. When its task is finished, it calls `context.Done` and closes, returning control to the previous dialog.

```

[Serializable]
public class SurveyDialog : IDialog<object>
{
    public async Task StartAsync(IDialogContext context)
    {
        await context.PostAsync("Hello, I'm the survey dialog. I'm interrupting your conversation to ask you a question. Type \"done\" to resume");

        context.Wait(this.MessageReceivedAsync);
    }
    public virtual async Task MessageReceivedAsync(IDialogContext context, IAwaitable<IMessageActivity> result)
    {
        if ((await result).Text == "done")
        {
            await context.PostAsync("Great, back to the original conversation!");
            context.Done(String.Empty); // Finish this dialog.
        }
        else
        {
            await context.PostAsync("I'm still on the survey until you type \"done\"");
            context.Wait(MessageReceivedAsync); // Not done yet.
        }
    }
}

```

Sample code

For a complete sample that shows how to send proactive messages using the Bot Builder SDK for .NET, see the [Proactive Messages sample](#) in GitHub. Within the Proactive Messages sample, [simpleSendMessage](#) shows how to send an ad-hoc proactive message and [startNewDialog](#) shows how to send a dialog-based proactive message.

Additional resources

- [Design and control conversation flow](#)
- [Bot Builder SDK for .NET Reference](#)
- [Proactive Messages sample \(GitHub\)](#)

Dialogs in the Bot Builder SDK for .NET

10/30/2017 • 8 min to read • [Edit Online](#)

When you create a bot using the Bot Builder SDK for .NET, you can use dialogs to model a conversation and manage [conversation flow](#). Each dialog is an abstraction that encapsulates its own state in a C# class that implements `IDialog`. A dialog can be composed with other dialogs to maximize reuse, and a dialog context maintains the [stack of dialogs](#) that are active in the conversation at any point in time.

A conversation that comprises dialogs is portable across computers, which makes it possible for your bot implementation to scale. When you use dialogs in the Bot Builder SDK for .NET, conversation state (the dialog stack and the state of each dialog in the stack) is automatically stored using the Bot Framework State service. This enables your bot to be stateless, much like a web application that does not need to store session state in web server memory.

Echo bot example

Consider this echo bot example, which describes how to change the bot that's created in the [Quickstart](#) tutorial so that it uses dialogs to exchange messages with the user.

TIP

To follow along with this example, use the instructions in the [Quickstart](#) tutorial to create a bot, and then update its `MessagesController.cs` file as described below.

MessagesController.cs

In the Bot Builder SDK for .NET, the [Builder](#) library enables you to implement dialogs. To access the relevant classes, import the `Dialogs` namespace.

```
using Microsoft.Bot.Builder.Dialogs;
```

Next, add this `EchoDialog` class to `MessagesController.cs` to represent the conversation.

```
[Serializable]
public class EchoDialog : IDialog<object>
{
    public async Task StartAsync(IDialogContext context)
    {
        context.Wait(MessageReceivedAsync);
    }

    public async Task MessageReceivedAsync(IDialogContext context, IAwaitable<IMessageActivity> argument)
    {
        var message = await argument;
        await context.PostAsync("You said: " + message.Text);
        context.Wait(MessageReceivedAsync);
    }
}
```

Then, wire the `EchoDialog` class to the `Post` method by calling the `Conversation.SendAsync` method.

```

public virtual async Task<HttpResponseMessage> Post([FromBody] Activity activity)
{
    // Check if activity is of type message
    if (activity != null && activity.GetActivityType() == ActivityTypes.Message)
    {
        await Conversation.SendAsync(activity, () => new EchoDialog());
    }
    else
    {
        HandleSystemMessage(activity);
    }
    return new HttpResponseMessage(System.Net.HttpStatusCode.Accepted);
}

```

Implementation details

The `Post` method is marked `async` because Bot Builder uses the C# facilities for handling asynchronous communication. It returns a `Task` object, which represents the task that is responsible for sending replies to the passed-in message. If there is an exception, the `Task` that is returned by the method will contain the exception information.

The `Conversation.SendAsync` method is key to implementing dialogs with the Bot Builder SDK for .NET. It follows the [dependency inversion principle](#) and performs these steps:

1. Instantiates the required components
2. Deserializes the conversation state (the dialog stack and the state of each dialog in the stack) from `IBotDataStore`
3. Resumes the conversation process where the bot suspended and waits for a message
4. Sends the replies
5. Serializes the updated conversation state and saves it back to `IBotDataStore`

When the conversation first starts, the dialog does not contain state, so `Conversation.SendAsync` constructs `EchoDialog` and calls its `StartAsync` method. The `StartAsync` method calls `IDialogContext.Wait` with the continuation delegate to specify the method that should be called when a new message is received (`MessageReceivedAsync`).

The `MessageReceivedAsync` method waits for a message, posts a response, and waits for the next message. Every time `IDialogContext.Wait` is called, the bot enters a suspended state and can be restarted on any computer that receives the message.

A bot that's created by using the code samples above will reply to each message that the user sends by simply echoing back the user's message prefixed with the text 'You said: '. Because the bot is created using dialogs, it can evolve to support more complex conversations without having to explicitly manage state.

Echo bot with state example

This next example builds upon the one above by adding the ability to track dialog state. When the `EchoDialog` class is updated as shown in the code sample below, the bot will reply to each message that the user sends by echoing back the user's message prefixed with a number (`count`) followed by the text 'You said: '. The bot will continue to increment `count` with each reply, until the user elects to reset the count.

MessagesController.cs

```

[Serializable]
public class EchoDialog : IDialog<object>
{
    protected int count = 1;

    public async Task StartAsync(IDialogContext context)
    {
        context.Wait(MessageReceivedAsync);
    }

    public virtual async Task MessageReceivedAsync(IDialogContext context, IAwaitable<IMessageActivity> argument)
    {
        var message = await argument;
        if (message.Text == "reset")
        {
            PromptDialog.Confirm(
                context,
                AfterResetAsync,
                "Are you sure you want to reset the count?",
                "Didn't get that!",
                promptStyle: PromptStyle.None);
        }
        else
        {
            await context.PostAsync($"{this.count++}: You said {message.Text}");
            context.Wait(MessageReceivedAsync);
        }
    }

    public async Task AfterResetAsync(IDialogContext context, IAwaitable<bool> argument)
    {
        var confirm = await argument;
        if (confirm)
        {
            this.count = 1;
            await context.PostAsync("Reset count.");
        }
        else
        {
            await context.PostAsync("Did not reset count.");
        }
        context.Wait(MessageReceivedAsync);
    }
}

```

Implementation details

As in the first example, the `MessageReceivedAsync` method is called when a new message is received. This time though, the `MessageReceivedAsync` method evaluates the user's message before responding. If the user's message is "reset", the built-in `PromptDialog.Confirm` prompt spawns a sub-dialog that asks the user to confirm the count reset. The sub-dialog has its own private state that does not interfere with the parent dialog's state. When the user responds to the prompt, the result of the sub-dialog is passed to the `AfterResetAsync` method, which sends a message to the user to indicate whether or not the count was reset and then calls `IDialogContext.Wait` with a continuation back to `MessageReceivedAsync` on the next message.

Dialog context

The `IDialogContext` interface that is passed into each dialog method provides access to the services that a dialog requires to save state and communicate with the channel. The `IDialogContext` interface comprises three interfaces: [Internals.IBotData](#), [Internals.IBotToUser](#), and [Internals.IDialogStack](#).

[Internals.IBotData](#)

The `Internals.IBotData` interface provides access to the per-user, per-conversation, and private conversation state data that's maintained by Connector. Per-user state data is useful for storing user data that is not related to a specific conversation, while per-conversation data is useful for storing general data about a conversation, and private conversation data is useful for storing user data that is related to a specific conversation.

Internals.IBotToUser

`Internals.IBotToUser` provides methods to send a message from bot to user. Messages may be sent inline with the response to the web API method call or directly by using the [Connector client](#). Sending and receiving messages through the dialog context ensures that the `Internals.IBotData` state is passed through the Connector.

Internals.IDialogStack

`Internals.IDialogStack` provides methods to manage the [dialog stack](#). Most of the time, the dialog stack will automatically be managed for you. However, there may be cases where you want to explicitly manage the stack. For example, you might want to call a child dialog and add it to the top of the dialog stack, mark the current dialog as complete (thereby removing it from the dialog stack and returning the result to the prior dialog in the stack), suspend the current dialog until a message from the user arrives, or even reset the dialog stack altogether.

Serialization

The dialog stack and the state of all active dialogs are serialized to the per-user, per-conversation `IBotDataBag`. The serialized blob is persisted in the messages that the bot sends to and receives from the [Connector](#). To be serialized, a `Dialog` class must include the `[Serializable]` attribute. All `IDialog` implementations in the [Builder](#) library are marked as serializable.

The [Chain methods](#) provide a fluent interface to dialogs that is usable in LINQ query syntax. The compiled form of LINQ query syntax often uses anonymous methods. If these anonymous methods do not reference the environment of local variables, then these anonymous methods have no state and are trivially serializable. However, if the anonymous method captures any local variable in the environment, the resulting closure object (generated by the compiler) is not marked as serializable. In this situation, Bot Builder will throw a `ClosureCaptureException` to identify the issue.

To use reflection to serialize classes that are not marked as serializable, the Builder library includes a reflection-based serialization surrogate that you can use to register with [Autofac](#).

```
var builder = new ContainerBuilder();
builder.RegisterModule(new DialogModule());
builder.RegisterModule(new ReflectionSurrogateModule());
```

Dialog chains

While you can explicitly manage the stack of active dialogs by using `IDialogStack.Call<R>` and `IDialogStack.Done<R>`, you can also implicitly manage the stack of active dialogs by using these fluent [Chain](#) methods.

METHOD	TYPE	NOTES
<code>Chain.Select<T, R></code>	LINQ	Supports "select" and "let" in LINQ query syntax.
<code>Chain.SelectMany<T, C, R></code>	LINQ	Supports successive "from" in LINQ query syntax.

METHOD	TYPE	NOTES
Chain.Where	LINQ	Supports "where" in LINQ query syntax.
Chain.From	Chains	Instantiates a new instance of a dialog.
Chain.Return	Chains	Returns a constant value into the chain.
Chain.Do	Chains	Allows for side-effects within the chain.
Chain.ContinueWith<T, R>	Chains	Simple chaining of dialogs.
Chain.Unwrap	Chains	Unwrap a dialog nested in a dialog.
Chain.DefaultIfException	Chains	Swallows an exception from the previous result and returns default(T).
Chain.Loop	Branch	Loops the entire chain of dialogs.
Chain.Fold	Branch	Folds results from an enumeration of dialogs into a single result.
Chain.Switch<T, R>	Branch	Supports branching into different dialog chains.
Chain.PostToUser	Message	Posts a message to the user.
Chain.WaitToBot	Message	Waits for a message to the bot.
Chain.PostToChain	Message	Starts a chain with a message from the user.

Examples

The LINQ query syntax uses the `chain.Select<T, R>` method.

```
var query = from x in new PromptDialog.PromptString(Prompt, Prompt, attempts: 1)
            let w = new string(x.Reverse().ToArray())
            select w;
```

Or the `Chain.SelectMany<T, C, R>` method.

```
var query = from x in new PromptDialog.PromptString("p1", "p1", 1)
            from y in new PromptDialog.PromptString("p2", "p2", 1)
            select string.Join(" ", x, y);
```

The `Chain.PostToUser<T>` and `Chain.WaitToBot<T>` methods post messages from the bot to the user and vice versa.

```
query = query.PostToUser();
```

The `Chain.Switch<T, R>` method branches the conversation dialog flow.

```

var logic =
    toBot
    .Switch
    (
        new RegexCase<string>(new Regex("^hello"), (context, text) =>
        {
            return "world!";
        }),
        new Case<string, string>((txt) => txt == "world", (context, text) =>
        {
            return "!";
        }),
        new DefaultCase<string, string>((context, text) =>
        {
            return text;
        })
    )
);

```

If `Chain.Switch<T, R>` returns a nested `IDialog<IDialog<T>>`, then the inner `IDialog<T>` can be unwrapped with `Chain.Unwrap<T>`. This allows branching conversations to different paths of chained dialogs, possibly of unequal length. This example shows a more complete example of branching dialogs written in the fluent chain style with implicit stack management.

```

var joke = Chain
    .PostToChain()
    .Select(m => m.Text)
    .Switch
    (
        Chain.Case
        (
            new Regex("^chicken"),
            (context, text) =>
                Chain
                    .Return("why did the chicken cross the road?")
                    .PostToUser()
                    .WaitToBot()
                    .Select(ignoreUser => "to get to the other side")
        ),
        Chain.Default<string, IDialog<string>>(
            (context, text) =>
                Chain
                    .Return("why don't you like chicken jokes?")
        )
    )
    .Unwrap()
    .PostToUser().
    Loop();

```

Next steps

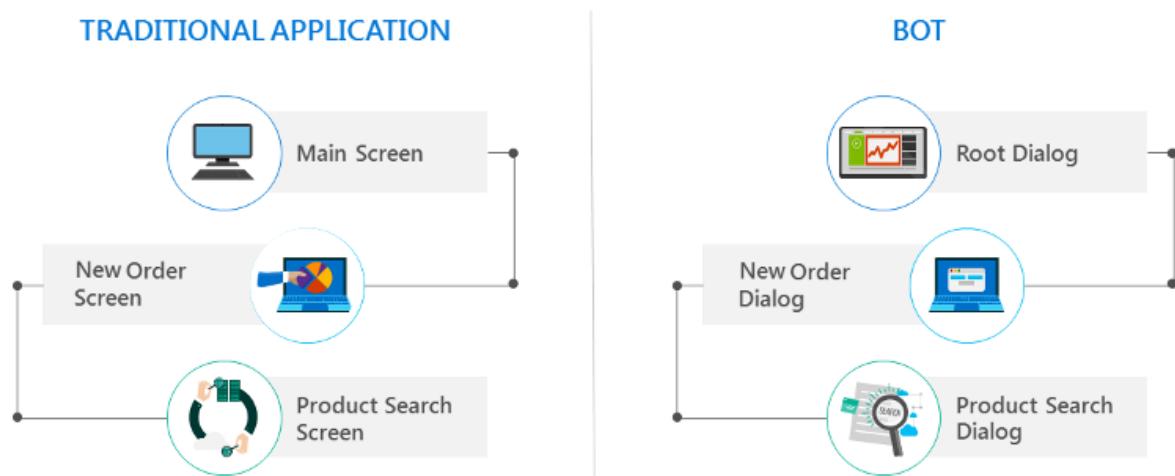
Dialogs manage conversation flow between a bot and a user. A dialog defines how to interact with a user. A bot can use many dialogs organized in stacks to guide the conversation with the user. In the next section, see how the dialog stack grows and shrinks as you create and dismiss dialogs in the stack.

[Manage conversation flow with dialogs](#)

Manage conversation flow with dialogs

11/2/2017 • 2 min to read • [Edit Online](#)

This diagram shows the screen flow of a traditional application compared to the dialog flow of a bot.



In a traditional application, everything begins with the **main screen**. The **main screen** invokes the **new order screen**. The **new order screen** remains in control until it either closes or invokes other screens. If the **new order screen** closes, the user is returned to the **main screen**.

In a bot, everything begins with the **root dialog**. The **root dialog** invokes the **new order dialog**. At that point, the **new order dialog** takes control of the conversation and remains in control until it either closes or invokes other dialogs. If the **new order dialog** closes, control of the conversation is returned back to the **root dialog**.

This article describes how to model this conversation flow by using [dialogs](#) and the Bot Builder SDK for .NET.

Invoke the root dialog

First, the bot controller invokes the "root dialog". The following example shows how to wire the basic HTTP GET call to a controller and then invoke the root dialog.

```
public class MessagesController : ApiController
{
    public async Task<HttpResponseMessage> Post([FromBody]Activity activity)
    {
        // Redirect to the root dialog.
        await Conversation.SendAsync(activity, () => new RootDialog());
        ...
    }
}
```

Invoke the 'New Order' dialog

Next, the root dialog invokes the 'New Order' dialog.

```

[Serializable]
public class RootDialog : IDialog<object>
{
    public async Task StartAsync(IDialogContext context)
    {
        // Root dialog initiates and waits for the next message from the user.
        // When a message arrives, call MessageReceivedAsync.
        context.Wait(this.MessageReceivedAsync);
    }

    public virtual async Task MessageReceivedAsync(IDialogContext context, IAwaitable<IMessageActivity> result)
    {
        var message = await result; // We've got a message!
        if (message.Text.ToLower().Contains("order"))
        {
            // User said 'order', so invoke the New Order Dialog and wait for it to finish.
            // Then, call ResumeAfterNewOrderDialog.
            await context.Forward(new NewOrderDialog(), this.ResumeAfterNewOrderDialog, message,
CancellationToken.None);
        }
        // User typed something else; for simplicity, ignore this input and wait for the next message.
        context.Wait(this.MessageReceivedAsync);
    }

    private async Task ResumeAfterNewOrderDialog(IDialogContext context, IAwaitable<string> result)
    {
        // Store the value that NewOrderDialog returned.
        // (At this point, new order dialog has finished and returned some value to use within the root
dialog.)
        var resultFromNewOrder = await result;

        await context.PostAsync($"New order dialog just told me this: {resultFromNewOrder}");

        // Again, wait for the next message from the user.
        context.Wait(this.MessageReceivedAsync);
    }
}

```

Dialog lifecycle

When a dialog is invoked, it takes control of the conversation flow. Every new message will be subject to processing by that dialog until it either closes or redirects to another dialog.

In C#, you can use `context.Wait()` to specify the callback to invoke the next time the user sends a message. To close a dialog and remove it from the stack (thereby sending the user back to the prior dialog in the stack), use `context.Done()`. You must end every dialog method with `context.Wait()`, `context.Fail()`, `context.Done()`, or some redirection directive such as `context.Forward()` or `context.Call()`. A dialog method that does not end with one of these will result in an error (because the framework does not know what action to take the next time the user sends a message).

Sample code

For a complete sample that shows how to manage a conversation by using dialogs in the Bot Builder SDK for .NET, see the [Basic Multi-Dialog sample](#) in GitHub.

Additional resources

- [Dialogs](#)
- [Design and control conversation flow](#)

- [Basic Multi-Dialog sample \(GitHub\)](#)
- [Bot Builder SDK for .NET Reference](#)

Global message handlers using scorables

9/28/2017 • 3 min to read • [Edit Online](#)

Users attempt to access certain functionality within a bot by using words like "help," "cancel," or "start over" in the middle of a conversation when the bot is expecting a different response. You can design your bot to gracefully handle such requests using scorable dialogs.

Scorable dialogs monitor all incoming messages and determine whether a message is actionable in some way. Messages that are scorable are assigned a score between [0 – 1] by each scorable dialog. The scorable dialog that determines the highest score is added to the top of the dialog stack and then hands the response to the user. After the scorable dialog completes execution, the conversation continues from where it left off.

Scorables enable you to create more flexible conversations by allowing your users to 'interrupt' the normal conversation flow you find in regular dialogs.

Create a scorable dialog

First, define a new [dialog](#). The following code uses a dialog that is derived from the `IDialog<object>` interface.

```
public class SampleDialog : IDialog<object>
{
    public async Task StartAsync(IDialogContext context)
    {
        await context.PostAsync("This is a Sample Dialog which is Scorable. Reply with anything to return to
the prior prior dialog.");

        context.Wait(this.MessageReceived);
    }

    private async Task MessageReceived(IDialogContext context, IAwaitable<IMessageActivity> result)
    {
        var message = await result;

        if ((message.Text != null) && (message.Text.Trim().Length > 0))
        {
            context.Done<object>(null);
        }
        else
        {
            context.Fail(new Exception("Message was not a string or was an empty string."));
        }
    }
}
```

To make a scorable dialog, create a class that inherits from the `ScorableBase` abstract class. The following code shows a `SampleScorable` class.

```

using Microsoft.Bot.Builder.Dialogs;
using Microsoft.Bot.Builder.Dialogs.Internals;
using Microsoft.Bot.Builder.Internals.Fibers;
using Microsoft.Bot.Builder.Scorables.Internals;

public class SampleScorable : ScorableBase<IActivity, string, double>
{
    private readonly IDialogTask task;

    public SampleScorable(IDialogTask task)
    {
        SetField.NotNull(out this.task, nameof(task), task);
    }
}

```

The `ScorableBase` abstract class inherits from the `IScorable` interface. You will need to implement the following `IScorable` methods in your class:

- `PrepareAsync` is the first method that is called in the scorable instance. It accepts incoming message activity, analyzes and sets the dialog's state, which is passed to all the other methods of the `IScorable` interface.

```

protected override async Task<string> PrepareAsync(IACTIVITY item, CancellationToken token)
{
    // TODO: insert your code here
}

```

- The `HasScore` method checks the state property to determine if the scorable dialog should provide a score for the message. If it returns false, the message will be ignored by the scorable dialog.

```

protected override bool HasScore(IACTIVITY item, string state)
{
    // TODO: insert your code here
}

```

- `GetScore` will only trigger if `HasScore` returns true. You'll provision the logic in this method to determine the score for a message between 0 - 1.

```

protected override double GetScore(IACTIVITY item, string state)
{
    // TODO: insert your code here
}

```

- In the `PostAsync` method, define core actions to be performed for the scorable class. All scorable dialogs will monitor incoming messages, and assign scores to valid messages based on the scorables' `GetScore` method. The scorable class which determines the highest score (between 0 - 1.0) will then trigger that scorable's `PostAsync` method.

```

protected override Task PostAsync(IACTIVITY item, string state, CancellationToken token)
{
    // TODO: insert your code here
}

```

- `DoneAsync` is called after the scoring process is complete. Use this method to dispose of any scoped resources.

```
protected override Task DoneAsync(IActivity item, string state, CancellationToken token)
{
    //TODO: insert your code here
}
```

Create a module to register the IScorable service

Next, define a `Module` that will register the `SampleScorable` class as a component. This will provision the `IScorable` service.

```
public class GlobalMessageHandlersBotModule : Module
{
    protected override void Load(ContainerBuilder builder)
    {
        base.Load(builder);

        builder
            .Register(c => new SampleScorable(c.Resolve<IDialogTask>()))
            .As<IScorable<IActivity, double>>()
            .InstancePerLifetimeScope();
    }
}
```

Register the module

The last step in the process is to apply the `SampleScorable` to the bot's Conversation Container. This will register the scorable service within the Bot Framework's message handling pipeline. The following code shows to update the `Conversation.Container` within the bot app's initialization in **Global.asax.cs**:

```
public class WebApiApplication : System.Web.HttpApplication
{
    protected void Application_Start()
    {
        this.RegisterBotModules();
        GlobalConfiguration.Configure(WebApiConfig.Register);
    }

    private void RegisterBotModules()
    {
        var builder = new ContainerBuilder();
        builder.RegisterModule(new ReflectionSurrogateModule());

        //Register the module within the Conversation container
        builder.RegisterModule<GlobalMessageHandlersBotModule>();

        builder.Update(Conversation.Container);
    }
}
```

Additional resources

- [Global Message Handlers sample](#)
- [Simple Scorable Bot sample](#)
- [Dialogs overview](#)
- [AutoFac](#)

Basic features of FormFlow

11/2/2017 • 11 min to read • [Edit Online](#)

[Dialogs](#) are very powerful and flexible, but handling a guided conversation such as ordering a sandwich can require a lot of effort. At each point in the conversation, there are many possibilities of what will happen next. For example, you may need to clarify an ambiguity, provide help, go back, or show progress. By using **FormFlow** within the Bot Builder SDK for .NET, you can greatly simplify the process of managing a guided conversation like this.

FormFlow automatically generates the dialogs that are necessary to manage a guided conversation, based upon guidelines that you specify. Although using FormFlow sacrifices some of the flexibility that you might otherwise get by creating and managing dialogs on your own, designing a guided conversation using FormFlow can significantly reduce the time it takes to develop your bot. Additionally, you may construct your bot using a combination of FormFlow-generated dialogs and other types of dialogs. For example, a FormFlow dialog may guide the user through the process of completing a form, while a [LuisDialog](#) may evaluate user input to determine intent.

This article describes how to create a bot that uses the basic features of FormFlow to collect information from a user.

Forms and fields

To create a bot using FormFlow, you must specify the information that the bot needs to collect from the user. For example, if the bot's objective is to obtain a user's sandwich order, then you must define a form that contains fields for the data that the bot needs to fulfill the order. You can define the form by creating a C# class that contains one or more public properties to represent the data that the bot will collect from the user. Each property must be one of these data types:

- Integral (sbyte, byte, short, ushort, int, uint, long, ulong)
- Floating point (float, double)
- String
- DateTime
- Enumeration
- List of enumerations

Any of the data types may be nullable, which you can use to model that the field does not have a value. If a form field is based on an enumeration property that is not nullable, the value **0** in the enumeration represents **null** (i.e., indicates that the field does not have a value), and you should start your enumeration values at **1**. FormFlow ignores all other property types and methods.

For complex objects, you must create a form for the top-level C# class and another form for the complex object. You can compose the forms together by using typical [dialog](#) semantics. It is also possible to define a form directly by implementing [Advanced.IField](#) or using [Advanced.Field](#) and populating the dictionaries within it.

NOTE

You can define a form by using either a C# class or JSON schema. This article describes how to define a form using a C# class. For more information about using JSON schema, see [Define a form using JSON schema](#).

Simple sandwich bot

Consider this example of a simple sandwich bot that is designed to obtain a user's sandwich order.

Create the form

The `SandwichOrder` class defines the form and the enumerations define the options for building a sandwich. The class also includes the static `BuildForm` method that uses `FormBuilder` to create the form and define a simple welcome message.

To use FormFlow, you must first import the `Microsoft.Bot.Builder.FormFlow` namespace.

```

using Microsoft.Bot.Builder.FormFlow;
using System;
using System.Collections.Generic;

// The SandwichOrder class represents the form that you want to complete
// using information that is collected from the user.
// It must be serializable so the bot can be stateless.
// The order of fields defines the default sequence in which the user is asked questions.

// The enumerations define the valid options for each field in SandwichOrder, and the order
// of the values represents the sequence in which they are presented to the user in a conversation.

namespace Microsoft.Bot.Sample.SimpleSandwichBot
{
    public enum SandwichOptions
    {
        BLT, BlackForestHam, BuffaloChicken, ChickenAndBaconRanchMelt, ColdCutCombo, MeatballMarinara,
        OvenRoastedChicken, RoastBeef, RotisserieStyleChicken, SpicyItalian, SteakAndCheese,
        SweetOnionTeriyaki, Tuna,
        TurkeyBreast, Veggie
    };
    public enum LengthOptions { SixInch, FootLong };
    public enum BreadOptions { NineGrainWheat, NineGrainHoneyOat, Italian, ItalianHerbsAndCheese, Flatbread
    };
    public enum CheeseOptions { American, MontereyCheddar, Pepperjack };
    public enum ToppingOptions
    {
        Avocado, BananaPeppers, Cucumbers, GreenBellPeppers, Jalapenos,
        Lettuce, Olives, Pickles, RedOnion, Spinach, Tomatoes
    };
    public enum SauceOptions
    {
        ChipotleSouthwest, HoneyMustard, LightMayonnaise, RegularMayonnaise,
        Mustard, Oil, Pepper, Ranch, SweetOnion, Vinegar
    };

    [Serializable]
    public class SandwichOrder
    {
        public SandwichOptions? Sandwich;
        public LengthOptions? Length;
        public BreadOptions? Bread;
        public CheeseOptions? Cheese;
        public List<ToppingOptions> Toppings;
        public List<SauceOptions> Sauce;

        public static IForm<SandwichOrder> BuildForm()
        {
            return new FormBuilder<SandwichOrder>()
                .Message("Welcome to the simple sandwich order bot!")
                .Build();
        }
    };
}

```

Connect the form to the framework

To connect the form to the framework, you must add it to the controller. In this example, the `Conversation.SendAsync` method calls the static `MakeRootDialog` method, which in turn, calls the `FormDialog.FromForm` method to create the `SandwichOrder` form.

```

internal static IDialog<SandwichOrder> MakeRootDialog()
{
    return Chain.From(() => FormDialog.FromForm(SandwichOrder.BuildForm));
}

[ResponseType(typeof(void))]
public virtual async Task<HttpResponseMessage> Post([FromBody] Activity activity)
{
    if (activity != null)
    {
        switch (activity.GetActivityType())
        {
            case ActivityTypes.Message:
                await Conversation.SendAsync(activity, MakeRootDialog);
                break;

            case ActivityTypes.ConversationUpdate:
            case ActivityTypes.ContactRelationUpdate:
            case ActivityTypes.Typing:
            case ActivityTypes.DeleteUserData:
            default:
                Trace.TraceError($"Unknown activity type ignored: {activity.GetActivityType()}");
                break;
        }
    }
    ...
}

```

See it in action

By simply defining the form with a C# class and connecting it to the framework, you have enabled FormFlow to automatically manage the conversation between bot and user. The example interactions shown below demonstrate the capabilities of a bot that is created by using the basic features of FormFlow. In each interaction, a > symbol indicates the point at which the user enters a response.

Display the first prompt

This form populates the `SandwichOrder.Sandwich` property. The form automatically generates the prompt, "Please select a sandwich", where the word "sandwich" in the prompt derives from the property name `Sandwich`. The `SandwichOptions` enumeration defines the choices that are presented to the user, with each enumeration value being automatically broken into words based upon changes in case and underscores.

```

Please select a sandwich
1. BLT
2. Black Forest Ham
3. Buffalo Chicken
4. Chicken And Bacon Ranch Melt
5. Cold Cut Combo
6. Meatball Marinara
7. Oven Roasted Chicken
8. Roast Beef
9. Rotisserie Style Chicken
10. Spicy Italian
11. Steak And Cheese
12. Sweet Onion Teriyaki
13. Tuna
14. Turkey Breast
15. Veggie
>

```

Provide guidance

The user can enter "help" at any point in the conversation to get guidance with filling out the form. For example, if the user enters "help" at the sandwich prompt, the bot will respond with this guidance.

```
> help
* You are filling in the sandwich field. Possible responses:
* You can enter a number 1-15 or words from the descriptions. (BLT, Black Forest Ham, Buffalo Chicken,
Chicken And Bacon Ranch Melt, Cold Cut Combo, Meatball Marinara, Oven Roasted Chicken, Roast Beef,
Rotisserie Style Chicken, Spicy Italian, Steak And Cheese, Sweet Onion Teriyaki, Tuna, Turkey Breast, and
Veggie)
* Back: Go back to the previous question.
* Help: Show the kinds of responses you can enter.
* Quit: Quit the form without completing it.
* Reset: Start over filling in the form. (With defaults from your previous entries.)
* Status: Show your progress in filling in the form so far.
* You can switch to another field by entering its name. (Sandwich, Length, Bread, Cheese, Toppings, and
Sauce).
```

Advance to the next prompt

If the user enters "2" in response to the initial sandwich prompt, the bot then displays a prompt for the next property that is defined by the form: `SandwichOrder.Length`.

```
Please select a sandwich
1. BLT
2. Black Forest Ham
3. Buffalo Chicken
4. Chicken And Bacon Ranch Melt
5. Cold Cut Combo
6. Meatball Marinara
7. Oven Roasted Chicken
8. Roast Beef
9. Rotisserie Style Chicken
10. Spicy Italian
11. Steak And Cheese
12. Sweet Onion Teriyaki
13. Tuna
14. Turkey Breast
15. Veggie
> 2
Please select a length (1. Six Inch, 2. Foot Long)
>
```

Return to the previous prompt

If the user enters "back" at this point in the conversation, the bot will return the previous prompt. The prompt shows the user's current choice ("Black Forest Ham"); the user may change that selection by entering a different number or confirm that selection by entering "c".

```
> back
Please select a sandwich(current choice: Black Forest Ham)
1. BLT
2. Black Forest Ham
3. Buffalo Chicken
4. Chicken And Bacon Ranch Melt
5. Cold Cut Combo
6. Meatball Marinara
7. Oven Roasted Chicken
8. Roast Beef
9. Rotisserie Style Chicken
10. Spicy Italian
11. Steak And Cheese
12. Sweet Onion Teriyaki
13. Tuna
14. Turkey Breast
15. Veggie
> c
Please select a length (1. Six Inch, 2. Foot Long)
>
```

Clarify user input

If the user responds with text (instead of a number) to indicate a choice, the bot will automatically ask for clarification if user input matches more than one choice.

```
Please select a bread
1. Nine Grain Wheat
2. Nine Grain Honey Oat
3. Italian
4. Italian Herbs And Cheese
5. Flatbread
> nine grain
By "nine grain" bread did you mean (1. Nine Grain Honey Oat, 2. Nine Grain Wheat)
> 1
```

If user input does not directly match any of the valid choices, the bot will automatically prompt the user for clarification.

```
Please select a cheese (1. American, 2. Monterey Cheddar, 3. Pepperjack)
> american
"american" is not a cheese option.
> american smoked
For cheese I understood American. "smoked" is not an option.
```

If user input specifies multiple choices for a property and the bot does not understand any of the specified choices, it will automatically prompt the user for clarification.

```
Please select one or more toppings
1. Banana Peppers
2. Cucumbers
3. Green Bell Peppers
4. Jalapenos
5. Lettuce
6. Olives
7. Pickles
8. Red Onion
9. Spinach
10. Tomatoes
> peppers, lettuce and tomato
By "peppers" toppings did you mean (1. Green Bell Peppers, 2. Banana Peppers)
> 1
```

Show current status

If the user enters "status" at any point in the order, the bot's response will indicate which values have already been specified and which values remain to be specified.

```
Please select one or more sauce
1. Honey Mustard
2. Light Mayonnaise
3. Regular Mayonnaise
4. Mustard
5. Oil
6. Pepper
7. Ranch
8. Sweet Onion
9. Vinegar
> status
* Sandwich: Black Forest Ham
* Length: Six Inch
* Bread: Nine Grain Honey Oat
* Cheese: American
* Toppings: Lettuce, Tomatoes, and Green Bell Peppers
* Sauce: Unspecified
```

Confirm selections

When the user completes the form, the bot will ask the user to confirm their selections.

```
Please select one or more sauce
1. Honey Mustard
2. Light Mayonnaise
3. Regular Mayonnaise
4. Mustard
5. Oil
6. Pepper
7. Ranch
8. Sweet Onion
9. Vinegar
> 1
Is this your selection?
* Sandwich: Black Forest Ham
* Length: Six Inch
* Bread: Nine Grain Honey Oat
* Cheese: American
* Toppings: Lettuce, Tomatoes, and Green Bell Peppers
* Sauce: Honey Mustard
>
```

If the user responds by entering "no", the bot allows the user to update any of the prior selections. If the user responds by entering "yes", the form has been completed and control is returned to the calling dialog.

```
Is this your selection?
* Sandwich: Black Forest Ham
* Length: Six Inch
* Bread: Nine Grain Honey Oat
* Cheese: American
* Toppings: Lettuce, Tomatoes, and Green Bell Peppers
* Sauce: Honey Mustard
> no

What do you want to change?
1. Sandwich(Black Forest Ham)
2. Length(Six Inch)
3. Bread(Nine Grain Honey Oat)
4. Cheese(American)
5. Toppings(Lettuce, Tomatoes, and Green Bell Peppers)
6. Sauce(Honey Mustard)
> 2
Please select a length (current choice: Six Inch) (1. Six Inch, 2. Foot Long)
> 2

Is this your selection?
* Sandwich: Black Forest Ham
* Length: Foot Long
* Bread: Nine Grain Honey Oat
* Cheese: American
* Toppings: Lettuce, Tomatoes, and Green Bell Peppers
* Sauce: Honey Mustard
> y
```

Handling quit and exceptions

If the user enters "quit" in the form or an exception occurs at some point in the conversation, your bot will need to know the step in which the event occurred, the state of the form when the event occurred, and which steps of the form were successfully completed prior to the event. The form returns this information via the `FormCanceledException<T>` class.

This code example shows how to catch the exception and display a message according to the event that occurred.

```

internal static IDialog<SandwichOrder> MakeRootDialog()
{
    return Chain.From(() => FormDialog.FromForm(SandwichOrder.BuildLocalizedForm))
        .Do(async (context, order) =>
    {
        try
        {
            var completed = await order;
            // Actually process the sandwich order...
            await context.PostAsync("Processed your order!");
        }
        catch (FormCanceledException<SandwichOrder> e)
        {
            string reply;
            if (e.InnerException == null)
            {
                reply = $"You quit on {e.Last} -- maybe you can finish next time!";
            }
            else
            {
                reply = "Sorry, I've had a short circuit. Please try again.";
            }
            await context.PostAsync(reply);
        }
    });
}

```

Summary

This article has described how to use the basic features of FormFlow to create a bot that can:

- Automatically generate and manage the conversation
- Provide clear guidance and help
- Understand both numbers and textual entries
- Provide feedback to the user regarding what is understood and what is not
- Ask clarifying questions when necessary
- Allow the user to navigate between steps

Although basic FormFlow functionality is sufficient in some cases, you should consider the potential benefits of incorporating some of the more advanced features of FormFlow into your bot. For more information, see [Advanced features of FormFlow](#) and [Customize a form using FormBuilder](#).

Sample code

For complete samples that show how to implement FormFlow using the Bot Builder SDK for .NET, see the [Multi-Dialog Bot sample](#) and the [Contoso Flowers Bot sample](#) in GitHub.

Next steps

FormFlow simplifies dialog development. The advanced features of FormFlow let you customize how a FormFlow object behaves.

[Advanced features of FormFlow](#)

Additional resources

- [Customize a form using FormBuilder](#)
- [Localize form content](#)

- [Define a form using JSON schema](#)
- [Customize user experience with pattern language](#)
- [Bot Builder SDK for .NET Reference](#)

Advanced features of FormFlow

11/2/2017 • 9 min to read • [Edit Online](#)

[Basic features of FormFlow](#) describes a basic FormFlow implementation that delivers a fairly generic user experience. To deliver a more customized user experience using FormFlow, you can specify initial form state, add business logic to manage interdependencies between fields and process user input, and use attributes to customize prompts, override templates, designate optional fields, match user input, and validate user input.

Specify initial form state and entities

When you launch a [FormDialog](#), you may optionally pass in an instance of your state. If you do pass in an instance of your state, then by default, FormFlow will skip steps for any fields that already contain values; the user will not be prompted for those fields. To force the form to prompt the user for all fields (including those fields that already contain values in the initial state), pass in [FormOptions.PromptFieldsWithValues](#) when you launch the [FormDialog](#). If a field contains an initial value, the prompt will use that value as the default value.

You can also pass in [LUIS](#) entities to bind to the state. If the `EntityRecommendation.Type` is a path to a field in your C# class, the `EntityRecommendation.Entity` will be passed through the recognizer to bind to your field. FormFlow will skip steps for any fields that are bound to an entity; the user will not be prompted for those fields.

Add business logic

To handle interdependencies between form fields or apply specific logic during the process of getting or setting a field value, you can specify business logic within a validation function. A validation function lets you manipulate the state and return a [ValidateResult](#) object that can contain:

- a feedback string that describes the reason that a value is invalid
- a transformed value
- a set of choices for clarifying a value

This code example shows a validation function for the `Toppings` field. If input for the field contains the `ToppingOptions.Everything` enumeration value, the function ensures that the `Toppings` field value contains the full list of toppings.

```

public static IForm<SandwichOrder> BuildForm()
{
    ...
    return new FormBuilder<SandwichOrder>()
        .Message("Welcome to the sandwich order bot!")
        .Field(nameof(Sandwich))
        .Field(nameof(Length))
        .Field(nameof(Bread))
        .Field(nameof(Cheese))
        .Field(nameof(Toppings),
            validate: async (state, value) =>
        {
            var values = ((List<object>)value).OfType<ToppingOptions>();
            var result = new ValidateResult { IsValid = true, Value = values };
            if (values != null && values.Contains(ToppingOptions.Everything))
            {
                result.Value = (from ToppingOptions topping in Enum.GetValues(typeof(ToppingOptions))
                                where topping != ToppingOptions.Everything && !values.Contains(topping)
                                select topping).ToList();
            }
            return result;
        })
        .Message("For sandwich toppings you have selected {Toppings}.")
        ...
        .Build();
}

```

In addition to the validation function, you can add the [Term](#) attribute to match user expressions such as "everything" or "not".

```

public enum ToppingOptions
{
    // This starts at 1 because 0 is the "no value" value
    [Terms("except", "but", "not", "no", "all", "everything")]
    Everything = 1,
    ...
}

```

Using the validation function shown above, this snippet shows the interaction between bot and user when the user requests "everything but Jalapenos."

```

Please select one or more toppings (current choice: No Preference)
1. Everything
2. Avocado
3. Banana Peppers
4. Cucumbers
5. Green Bell Peppers
6. Jalapenos
7. Lettuce
8. Olives
9. Pickles
10. Red Onion
11. Spinach
12. Tomatoes
> everything but jalapenos
For sandwich toppings you have selected Avocado, Banana Peppers, Cucumbers, Green Bell Peppers, Lettuce, Olives, Pickles, Red Onion, Spinach, and Tomatoes.

```

FormFlow attributes

You can add these C# attributes to your class to customize behavior of a FormFlow dialog.

ATTRIBUTE	PURPOSE
Describe	Alter how a field or a value is shown in a template or card
Numeric	Restrict the accepted values of a numeric field
Optional	Mark a field as optional
Pattern	Define a regular expression to validate a string field
Prompt	Define the prompt for a field
Template	Define the template to use to generate prompts or values in prompts
Terms	Define the input terms that match a field or value

Customize prompts using the Prompt attribute

Default prompts are automatically generated for each field in your form, but you can specify a custom prompt for any field by using the `Prompt` attribute. For example, if the default prompt for the `SandwichOrder.Sandwich` field is "Please select a sandwich", you can add the `Prompt` attribute to specify a custom prompt for that field.

```
[Prompt("What kind of {&} would you like? {|||}")]
public SandwichOptions? Sandwich;
```

This example uses [pattern language](#) to dynamically populate the prompt with form data at runtime: `{&}` is replaced with the description of the field and `{|||}` is replaced with the list of choices in the enumeration.

NOTE

By default, the description of a field is generated from the field's name. To specify a custom description for a field, add the `Describe` attribute.

This snippet shows the customized prompt that is specified by the example above.

```
What kind of sandwich would you like?
1. BLT
2. Black Forest Ham
3. Buffalo Chicken
4. Chicken And Bacon Ranch Melt
5. Cold Cut Combo
6. Meatball Marinara
7. Oven Roasted Chicken
8. Roast Beef
9. Rotisserie Style Chicken
10. Spicy Italian
11. Steak And Cheese
12. Sweet Onion Teriyaki
13. Tuna
14. Turkey Breast
15. Veggie
>
```

A `Prompt` attribute may also specify parameters that affect how the form displays the prompt. For example, the

`ChoiceFormat` parameter determines how the form renders the list of choices.

```
[Prompt("What kind of {&} would you like? {|||}", ChoiceFormat="{1}")]
public SandwichOptions? Sandwich;
```

In this example, the value of the `ChoiceFormat` parameter indicates that the choices should be displayed as a bulleted list (instead of a numbered list).

```
What kind of sandwich would you like?
- BLT
- Black Forest Ham
- Buffalo Chicken
- Chicken And Bacon Ranch Melt
- Cold Cut Combo
- Meatball Marinara
- Oven Roasted Chicken
- Roast Beef
- Rotisserie Style Chicken
- Spicy Italian
- Steak And Cheese
- Sweet Onion Teriyaki
- Tuna
- Turkey Breast
- Veggie
>
```

Customize prompts using the `Template` attribute

While the `Prompt` attribute enables you to customize the prompt for a single field, the `Template` attribute enables you to replace the default templates that FormFlow uses to automatically generate prompts. This code example uses the `Template` attribute to redefine how the form handles all enumeration fields. The attribute indicates that the user may select only one item, sets the prompt text by using [pattern language](#), and specifies that the form should display only one item per line.

```
[Template(TemplateUsage.EnumSelectOne, "What kind of {&} would you like on your sandwich? {|||}", ChoiceStyle =
ChoiceStyleOptions.PerLine)]
public class SandwichOrder
```

This snippet shows the resulting prompts for the `Bread` field and `Cheese` field.

```
What kind of bread would you like on your sandwich?
1. Nine Grain Wheat
2. Nine Grain Honey Oat
3. Italian
4. Italian Herbs And Cheese
5. Flatbread
>
```

```
What kind of cheese would you like on your sandwich?
1. American
2. Monterey Cheddar
3. Pepperjack
>
```

If you use the `Template` attribute to replace the default templates that FormFlow uses to generate prompts, you may want to interject some variation into the prompts and messages that the form generates. To do so, you can define multiple text strings using [pattern language](#), and the form will randomly choose from the available options

each time it needs to display a prompt or message.

This code example redefines the `TemplateUsage.NotUnderstood` template to specify two different variations of message. When the bot needs to communicate that it does not understand a user's input, it will determine message contents by randomly selecting one of the two text strings.

```
[Template(TemplateUsage.NotUnderstood, "I do not understand \"{0}\".", "Try again, I don't get \"{0}\".")]  
[Template(TemplateUsage.EnumSelectOne, "What kind of {&} would you like on your sandwich? {||}")]  
public class SandwichOrder
```

This snippet shows an example of the resulting interaction between bot and user.

```
What size of sandwich do you want? (1. Six Inch, 2. Foot Long)  
> two feet  
I do not understand "two feet".  
> two feet  
Try again, I don't get "two feet"  
>
```

Designate a field as optional using the `Optional` attribute

To designate a field as optional, use the `Optional` attribute. This code example specifies that the `Cheese` field is optional.

```
[Optional]  
public CheeseOptions? Cheese;
```

If a field is optional and no value has been specified, the current choice will be displayed as "No Preference".

```
What kind of cheese would you like on your sandwich? (current choice: No Preference)  
1. American  
2. Monterey Cheddar  
3. Pepperjack  
>
```

If a field is optional and the user has specified a value, "No Preference" will be displayed as the last choice in the list.

```
What kind of cheese would you like on your sandwich? (current choice: American)  
1. American  
2. Monterey Cheddar  
3. Pepperjack  
4. No Preference  
>
```

Match user input using the `Terms` attribute

When a user sends a message to a bot that is built using FormFlow, the bot attempts to identify the meaning of the user's input by matching the input to a list of terms. By default, the list of terms is generated by applying these steps to the field or value:

1. Break on case changes and underscore (_).
2. Generate each `n`-gram up to a maximum length.
3. Add "s?" to the end of each word (to support plurals).

For example, the value "AngusBeefAndGarlicPizza" would generate these terms:

- 'angus?'
- 'beefs?'
- 'garlics?'
- 'pizzas?'
- 'angus? beefs?'
- 'garlics? pizzas?'
- 'angus beef and garlic pizza'

To override this default behavior and define the list of terms that are used to match user input to a field or a value in a field, use the `Terms` attribute. For example, you may use the `Terms` attribute (with a regular expression) to account for the fact that users are likely to misspell the word "rotisserie."

```
[Terms(@"rotis\w* style chicken", MaxPhrase = 3)]  
RotisserieStyleChicken, SpicyItalian, SteakAndCheese, SweetOnionTeriyaki, Tuna,...
```

By using the `Terms` attribute, you increase the likelihood of being able to match user input with one of the valid choices. The `Terms.MaxPhrase` parameter in this example causes the `Language.GenerateTerms` to generate additional variations of terms.

This snippet shows the resulting interaction between bot and user when the user misspells "Rotisserie."

```
What kind of sandwich would you like?  
1. BLT  
2. Black Forest Ham  
3. Buffalo Chicken  
4. Chicken And Bacon Ranch Melt  
5. Cold Cut Combo  
6. Meatball Marinara  
7. Oven Roasted Chicken  
8. Roast Beef  
9. Rotisserie Style Chicken  
10. Spicy Italian  
11. Steak And Cheese  
12. Sweet Onion Teriyaki  
13. Tuna  
14. Turkey Breast  
15. Veggie  
> rotissary checkin  
For sandwich I understood Rotisserie Style Chicken. "checkin" is not an option.
```

Validate user input using the Numeric attribute or Pattern attribute

To restrict the range of allowed values for a numeric field, use the `Numeric` attribute. This code example uses the `Numeric` attribute to specify that input for the `Rating` field must be a number between 1 and 5.

```
[Numeric(1, 5)]  
public double? Rating;
```

To specify the required format for the value of a particular field, use the `Pattern` attribute. This code example uses the `Pattern` attribute to specify the required format for the value of the `PhoneNumber` field.

```
[Pattern(@"(<Undefined control sequence>\d)?\s*\d{3}(-|\s*)\d{4}")]
public string PhoneNumber;
```

Summary

This article has described how to deliver a customized user experience with FormFlow by specifying initial form state, adding business logic to manage interdependencies between fields and process user input, and using attributes to customize prompts, override templates, designate optional fields, match user input, and validate user input. For information about additional ways to customize the user experience with FormFlow, see [Customize a form using FormBuilder](#).

Sample code

For complete samples that show how to implement FormFlow using the Bot Builder SDK for .NET, see the [Multi-Dialog Bot sample](#) and the [Contoso Flowers Bot sample](#) in GitHub.

Additional resources

- [Basic features of FormFlow](#)
- [Customize a form using FormBuilder](#)
- [Localize form content](#)
- [Define a form using JSON schema](#)
- [Customize user experience with pattern language](#)
- [Bot Builder SDK for .NET Reference](#)

Customize a form using FormBuilder

11/2/2017 • 4 min to read • [Edit Online](#)

[Basic features of FormFlow](#) describes a basic FormFlow implementation that delivers a fairly generic user experience, and [Advanced features of FormFlow](#) describes how you can customize user experience by using business logic and attributes. This article describes how you can use [FormBuilder](#) to customize user experience even further, by specifying the sequence in which the form executes steps and dynamically defining field values, confirmations, and messages.

Dynamically define field values, confirmations, and messages

Using FormBuilder, you can dynamically define field values, confirmations, and messages.

Dynamically define field values

A sandwich bot that is designed to add a free drink or cookie to any order that specifies a foot-long sandwich uses the `Sandwich.Specials` field to store data about free items. In this case, the value of the `Sandwich.Specials` field must be dynamically set for each order according to whether or not the order contains a foot-long sandwich.

The `Specials` field is specified as optional and "None" is designated as text for the choice that indicates no preference.

```
[Optional]
[Template(TemplateUsage.NoPreference, "None")]
public string Specials;
```

This code example shows how to dynamically set the value of the `Specials` field.

```
.Field(new FieldReflector<SandwichOrder>(nameof(Specials))
    .SetType(null)
    .SetActive((state) => state.Length == LengthOptions.FootLong)
    .SetDefine(async (state, field) =>
{
    field
        .AddDescription("cookie", "Free cookie")
        .AddTerms("cookie", "cookie", "free cookie")
        .AddDescription("drink", "Free large drink")
        .AddTerms("drink", "drink", "free drink");
    return true;
}))
```

In this example, the [Advanced.Field.SetType](#) method specifies the field type (`null` represents an enumeration field). The [Advanced.Field.SetActive](#) method specifies that the field should only be enabled if the length of the sandwich is `Length.FootLong`. Finally, the [Advanced.Field.SetDefine](#) method specifies an async delegate that defines the field. The delegate is passed the current state object and the [Advanced.Field](#) that is being dynamically defined. The delegate uses the field's fluent methods to dynamically define values. In this example, the values are strings and the `AddDescription` and `AddTerms` methods specify the descriptions and terms for each value.

NOTE

To dynamically define a field value, you can implement [Advanced.IField](#) yourself, or streamline the process by using the [Advanced.FieldReflector](#) class as shown in the example above.

Dynamically define messages and confirmations

Using FormBuilder, you can also dynamically define messages and confirmations. Each message and confirmation runs only when prior steps in the form are inactive or completed.

This code example shows a dynamically generated confirmation that computes the cost of the sandwich.

```
.Confirm(async (state) =>
{
    var cost = 0.0;
    switch (state.Length)
    {
        case LengthOptions.SixInch: cost = 5.0; break;
        case LengthOptions.FootLong: cost = 6.50; break;
    }
    return new PromptAttribute($"Total for your sandwich is {cost:C2} is that ok?");
})
```

Customize a form using FormBuilder

This code example uses FormBuilder to define the steps of the form, [validate selections](#), and [dynamically define a field value and confirmation](#). By default, steps in the form will be executed in the sequence in which they are listed. However, steps might be skipped for fields that already contain values or if explicit navigation is specified.

```
public static IForm<SandwichOrder> BuildForm()
{
    OnCompletionAsyncDelegate<SandwichOrder> processOrder = async (context, state) =>
    {
        await context.PostAsync("We are currently processing your sandwich. We will message you the status.");
    };

    return new FormBuilder<SandwichOrder>()
        .Message("Welcome to the sandwich order bot!")
        .Field(nameof(Sandwich))
        .Field(nameof(Length))
        .Field(nameof(Bread))
        .Field(nameof(Cheese))
        .Field(nameof(Toppings),
            validate: async (state, value) =>
        {
            var values = ((List<object>)value).OfType<ToppingOptions>();
            var result = new ValidateResult { IsValid = true, Value = values };
            if (values != null && values.Contains(ToppingOptions.Everything))
            {
                result.Value = (from ToppingOptions topping in Enum.GetValues(typeof(ToppingOptions))
                                where topping != ToppingOptions.Everything && !values.Contains(topping)
                                select topping).ToList();
            }
            return result;
        })
        .Message("For sandwich toppings you have selected {Toppings}.")
        .Field(nameof(SandwichOrder.Sauces))
        .Field(new FieldReflector<SandwichOrder>(nameof(Specials))
            .SetType(null)
            .SetActive((state) => state.Length == LengthOptions.FootLong)
            .SetDefine(async (state, field) =>
        {
            field
                .AddDescription("cookie", "Free cookie")
                .AddTerms("cookie", "cookie", "free cookie")
                .AddDescription("drink", "Free large drink")
                .AddTerms("drink", "drink", "free drink");
            return true;
        })
        .SetType(null)
        .SetActive((state) => state.Length == LengthOptions.FootLong)
        .SetDefine(async (state, field) =>
    {
        field
            .AddDescription("cookie", "Free cookie")
            .AddTerms("cookie", "cookie", "free cookie")
            .AddDescription("drink", "Free large drink")
            .AddTerms("drink", "drink", "free drink");
        return true;
    })
    .SetType(null)
    .SetActive((state) => state.Length == LengthOptions.FootLong)
    .SetDefine(async (state, field) =>
{
    field
        .AddDescription("cookie", "Free cookie")
        .AddTerms("cookie", "cookie", "free cookie")
        .AddDescription("drink", "Free large drink")
        .AddTerms("drink", "drink", "free drink");
    return true;
})
})
```

```

        })
        .Confirm(async (state) =>
    {
        var cost = 0.0;
        switch (state.Length)
        {
            case LengthOptions.SixInch: cost = 5.0; break;
            case LengthOptions.FootLong: cost = 6.50; break;
        }
        return new PromptAttribute($"Total for your sandwich is {cost:C2} is that ok?");
    })
    .Field(nameof(SandwichOrder.DeliveryAddress),
    validate: async (state, response) =>
    {
        var result = new ValidateResult { IsValid = true, Value = response };
        var address = (response as string).Trim();
        if (address.Length > 0 && (address[0] < '0' || address[0] > '9'))
        {
            result.Feedback = "Address must start with a number.";
            result.IsValid = false;
        }
        return result;
    })
    .Field(nameof(SandwichOrder.DeliveryTime), "What time do you want your sandwich delivered? {}")
    .Confirm("Do you want to order your {Length} {Sandwich} on {Bread} {&Bread} with {{Cheese}{Toppings}{Sauces}} to be sent to {DeliveryAddress} {at {DeliveryTime:t}}?")
    .AddRemainingFields()
    .Message("Thanks for ordering a sandwich!")
    .OnCompletion(processOrder)
    .Build();
}

```

In this example, the form executes these steps:

- Shows a welcome message.
- Fills in `SandwichOrder.Sandwich`.
- Fills in `SandwichOrder.Length`.
- Fills in `SandwichOrder.Bread`.
- Fills in `SandwichOrder.Cheese`.
- Fills in `SandwichOrder.Toppings` and adds missing values if the user selected `ToppingOptions.Everything`. Shows a message that confirms the selected toppings.
- Fills in `SandwichOrder.Sauces`.
- Dynamically defines** the field value for `SandwichOrder.Specials`.
- Dynamically defines** the confirmation for cost of the sandwich.
- Fills in `SandwichOrder.DeliveryAddress` and **verifies** the resulting string. If the address does not start with a number, the form returns a message.
- Fills in `SandwichOrder.DeliveryTime` with a custom prompt.
- Confirms the order.
- Adds any remaining fields that were defined in the class but not explicitly referenced by `Field`. (If the example did not call the `AddRemainingFields` method, the form would not include any fields that were not explicitly referenced.)
- Shows a thank you message.
- Defines an `onCompletionAsync` handler to process the order.

Sample code

For complete samples that show how to implement FormFlow using the Bot Builder SDK for .NET, see the [Multi-Dialog Bot sample](#) and the [Contoso Flowers Bot sample](#) in GitHub.

Additional resources

- [Basic features of FormFlow](#)
- [Advanced features of FormFlow](#)
- [Localize form content](#)
- [Define a form using JSON schema](#)
- [Customize user experience with pattern language](#)
- [Bot Builder SDK for .NET Reference](#)

Customize user experience with pattern language

11/2/2017 • 5 min to read • [Edit Online](#)

When you customize a prompt or override a default template, you can use pattern language to specify the contents and/or format of the prompt.

Prompts and templates

A [prompt](#) defines the message that is sent to the user to request a piece of information or ask for confirmation. You can customize a prompt by using the [Prompt attribute](#) or implicitly through [IFormBuilder.Field](#).

Forms use templates to automatically construct prompts and other things such as help. You can override the default template of a class or field by using the [Template attribute](#).

TIP

The [FormConfiguration.Templates](#) class defines a set of built-in templates that provide good examples of how to use pattern language.

Elements of pattern language

Pattern language uses curly braces (`{ }`) to identify elements that will be replaced at runtime with actual values.

This table lists the elements of pattern language.

ELEMENT	DESCRIPTION
<code>{<format>}</code>	Shows the value of the current field (the field that the attribute applies to).
<code>{&}</code>	Shows the description of the current field (unless otherwise specified, this is the name of the field).
<code>{<field><format>}</code>	Shows the value of the named field.
<code>{&<field>}</code>	Shows the description of the named field.
<code>{ }</code>	Shows the current choice(s), which could be the current value of a field, "no preference" or the values of an enumeration.
<code>{[<field><format>} ...]}</code>	Shows a list of values from the named fields using Separator and LastSeparator to separate the individual values in the list.
<code>{*}</code>	Shows one line for each active field; each line contains the field description and current value.
<code>{*filled}</code>	Shows one line for each active field that contains an actual value; each line contains the field description and current value.

ELEMENT	DESCRIPTION
{<nth><format>}	A regular C# format specifier that applies to the nth argument of a template. For the list of available arguments, see TemplateUsage .
{?<textOrPatternElement>...}	Conditional substitution. If all referred to pattern elements have values, the values are substituted and the whole expression is used.

For the elements listed above:

- The `<field>` placeholder is the name of a field in your form class. For example, if your class contains a field with the name `Size`, you could specify `{Size}` as the pattern element.
- An ellipsis (`"..."`) within a pattern element indicates that the element may contain multiple values.
- The `<format>` placeholder is a C# format specifier. For example, if the class contains a field with the name `Rating` and of type `double`, you could specify `{Rating:F2}` as the pattern element to show two digits of precision.
- The `<nth>` placeholder references the nth argument of a template.

Pattern language within a Prompt attribute

This example uses the `{&}` element to show the description of the `Sandwich` field and the `{||}` element to show the list of choices for that field.

```
[Prompt("What kind of {&} would you like? {||}")]
public SandwichOptions? Sandwich;
```

This is the resulting prompt:

```
What kind of sandwich would you like?
1. BLT
2. Black Forest Ham
3. Buffalo Chicken
4. Chicken And Bacon Ranch Melt
5. Cold Cut Combo
6. Meatball Marinara
7. Oven Roasted Chicken
8. Roast Beef
9. Rotisserie Style Chicken
10. Spicy Italian
11. Steak And Cheese
12. Sweet Onion Teriyaki
13. Tuna
14. Turkey Breast
15. Veggie
>
```

Formatting parameters

Prompts and templates support these formatting parameters.

USAGE	DESCRIPTION
-------	-------------

USAGE	DESCRIPTION
<code>AllowDefault</code>	Applies to <code>{ }</code> pattern elements. Determines whether the form should show the current value of the field as a possible choice. If <code>true</code> , the current value is shown as a possible value. The default is <code>true</code> .
<code>ChoiceCase</code>	Applies to <code>{ }</code> pattern elements. Determines whether the text of each choice is normalized (e.g., whether the first letter of each word is capitalized). The default is <code>CaseNormalization.None</code> . For possible values, see CaseNormalization .
<code>ChoiceFormat</code>	Applies to <code>{ }</code> pattern elements. Determines whether to show a list of choices as a numbered list or a bulleted list. For a numbered list, set <code>ChoiceFormat</code> to <code>{0}</code> (default). For a bulleted list, set <code>ChoiceFormat</code> to <code>{1}</code> .
<code>ChoiceLastSeparator</code>	Applies to <code>{ }</code> pattern elements. Determines whether an inline list of choices includes a separator before the last choice.
<code>ChoiceParens</code>	Applies to <code>{ }</code> pattern elements. Determines whether an inline list of choices is shown within parentheses. If <code>true</code> , the list of choices is shown within parentheses. The default is <code>true</code> .
<code>ChoiceSeparator</code>	Applies to <code>{ }</code> pattern elements. Determines whether an inline list of choices includes a separator before every choice except the last choice.
<code>ChoiceStyle</code>	Applies to <code>{ }</code> pattern elements. Determines whether the list of choices is shown inline or per line. The default is <code>ChoiceStyleOptions.Auto</code> which determines at runtime whether to show the choice inline or in a list. For possible values, see ChoiceStyleOptions .
<code>Feedback</code>	Applies to prompts only. Determines whether the form echoes the user's choice to indicate that the form understood the selection. The default is <code>FeedbackOptions.Auto</code> which echoes the user's input only if part of it is not understood. For possible values, see FeedbackOptions .
<code>FieldCase</code>	Determines whether the text of the field's description is normalized (e.g., whether the first letter of each word is capitalized). The default is <code>CaseNormalization.Lower</code> which converts the description to lowercase. For possible values, see CaseNormalization .
<code>LastSeparator</code>	Applies to <code>{[]}</code> pattern elements. Determines whether an array of items includes a separator before the last item.
<code>Separator</code>	Applies to <code>{[]}</code> pattern elements. Determines whether an array of items includes a separator before every item in the array except the last item.

USAGE	DESCRIPTION
<pre>ValueCase</pre>	<p>Determines whether the text of the field's value is normalized (e.g., whether the first letter of each word is capitalized). The default is <code>CaseNormalization.InitialUpper</code> which converts the first letter of each word to uppercase. For possible values, see CaseNormalization.</p>

Prompt attribute with formatting parameter

This example uses the `ChoiceFormat` parameter to specify that the list of choices should be displayed as a bulleted list.

```
[Prompt("What kind of {&} would you like? {||}", ChoiceFormat="{1}")]
public SandwichOptions? Sandwich;
```

This is the resulting prompt:

```
What kind of sandwich would you like?
* BLT
* Black Forest Ham
* Buffalo Chicken
* Chicken And Bacon Ranch Melt
* Cold Cut Combo
* Meatball Marinara
* Oven Roasted Chicken
* Roast Beef
* Rotisserie Style Chicken
* Spicy Italian
* Steak And Cheese
* Sweet Onion Teriyaki
* Tuna
* Turkey Breast
* Veggie
>
```

Sample code

For complete samples that show how to implement FormFlow using the Bot Builder SDK for .NET, see the [Multi-Dialog Bot sample](#) and the [Contoso Flowers Bot sample](#) in GitHub.

Additional resources

- [Basic features of FormFlow](#)
- [Advanced features of FormFlow](#)
- [Customize a form using FormBuilder](#)
- [Localize form content](#)
- [Define a form using JSON schema](#)
- [Bot Builder SDK for .NET Reference](#)

Localize form content

11/2/2017 • 6 min to read • [Edit Online](#)

A form's localization language is determined by the current thread's [CurrentUICulture](#) and [CurrentCulture](#). By default, the culture derives from the **Locale** field of the current message, but you can override that default behavior. Depending on how your bot is constructed, localized information may come from up to three different sources:

- the built-in localization for **PromptDialog** and **FormFlow**
- a resource file that you generate for the static strings in your form
- a resource file that you create with strings for dynamically-computed fields, messages or confirmations

Generate a resource file for the static strings in your form

Static strings in a form include the strings that the form generates from the information in your C# class and the strings that you specify as prompts, templates, messages or confirmations. Strings that are generated from built-in templates are not considered static strings, since those strings are already localized. Since many of the strings in a form are automatically generated, it is not feasible to use normal C# resource strings directly. Instead, you can generate a resource file for the static strings in your form either by calling `IFormBuilder.SaveResources` or by using the **RView** tool that is included with the BotBuilder SDK for .NET.

Use `IFormBuilder.SaveResources`

You can generate a resource file by calling `IFormBuilder.SaveResources` on your form to save the strings to a .resx file.

Use **RView**

Alternatively, you can generate a resource file that is based upon your .dll or .exe by using the **RView** tool that is included in the BotBuilder SDK for .NET. To generate the .resx file, execute **rview** and specify the assembly that contains your static form-building method and the path to that method. This snippet shows how to generate the `Microsoft.Bot.Sample.AnnotatedSandwichBot.SandwichOrder.resx` resource file using **RView**.

```
rview -g Microsoft.Bot.Sample.AnnotatedSandwichBot.dll  
Microsoft.Bot.Sample.AnnotatedSandwichBot.SandwichOrder.BuildForm
```

This excerpt shows part of the .resx file that is generated by executing this **rview** command.

```

<data name="Specials_description;VALUE" xml:space="preserve">
<value>Specials</value>
</data>
<data name="DeliveryAddress_description;VALUE" xml:space="preserve">
<value>Delivery Address</value>
</data>
<data name="DeliveryTime_description;VALUE" xml:space="preserve">
<value>Delivery Time</value>
</data>
<data name="PhoneNumber_description;VALUE" xml:space="preserve">
<value>Phone Number</value>
</data>
<data name="Rating_description;VALUE" xml:space="preserve">
<value>your experience today</value>
</data>
<data name="message0;LIST" xml:space="preserve">
<value>Welcome to the sandwich order bot!</value>
</data>
<data name="Sandwich_terms;LIST" xml:space="preserve">
<value>sandwichs?</value>
</data>

```

Configure your project

After you have generated a resource file, add it to your project and then set the neutral language by completing these steps:

1. Right-click on your project and select **Application**.
2. Click **Assembly Information**.
3. Select the **Neutral Language** value that corresponds to the language in which you developed your bot.

When your form is created, the [IFormBuilder.Build](#) method will automatically look for resources that contain your form type name and use them to localize the static strings in your form.

NOTE

Dynamically-computed fields that are defined using [Advanced.Field.SetDefine](#) (as described in [Using Dynamic Fields](#)) cannot be localized in the same manner as static fields, since strings for dynamically-computed fields are constructed at the time the form is populated. However, you can localize dynamically-computed fields by using normal C# localization mechanisms.

Localize resource files

After you have added resource files to your project, you can localize them by using the [Multilingual App Toolkit \(MAT\)](#). Install **MAT**, then enable it for your project by completing these steps:

1. Select your project in the Visual Studio Solution Explorer.
2. Click **Tools**, **Multilingual App Toolkit**, and **Enable**.
3. Right-click the project and select **Multilingual App Toolkit, Add Translations** to select the translations. This will create industry-standard [XLF](#) files that you can automatically or manually translate.

NOTE

Although this article describes how to use the Multilingual App Toolkit to localize content, you may implement localization via a variety of other means.

See it in action

This code example builds upon the one in [Customize a form using FormBuilder](#) to implement localization as described above. In this example, the `DynamicSandwich` class (not shown here) contains localization information for dynamically-computed fields, messages and confirmations.

```
public static IForm<SandwichOrder> BuildLocalizedForm()
{
    var culture = Thread.CurrentThread.CurrentCulture;
    IForm<SandwichOrder> form;
    if (!_forms.TryGetValue(culture, out form))
    {
        OnCompletionAsyncDelegate<SandwichOrder> processOrder = async (context, state) =>
        {
            await context.PostAsync(DynamicSandwich.Processing);
        };
        // FormBuilder uses the thread culture to automatically switch framework strings and static strings.
        // Dynamically defined fields must do their own localization.
        var builder = new FormBuilder<SandwichOrder>()
            .Message("Welcome to the sandwich order bot!")
            .Field(nameof(Sandwich))
            .Field(nameof(Length))
            .Field(nameof(Bread))
            .Field(nameof(Cheese))
            .Field(nameof(Toppings),
                validate: async (state, value) =>
            {
                var values = ((List<object>)value).OfType<ToppingOptions>();
                var result = new ValidationResult { IsValid = true, Value = values };
                if (values != null && values.Contains(ToppingOptions.Everything))
                {
                    result.Value = (from ToppingOptions topping in
Enum.GetValues(typeof(ToppingOptions))
                        where topping != ToppingOptions.Everything &&
!values.Contains(topping)
                        select topping).ToList();
                }
                return result;
            })
            .Message("For sandwich toppings you have selected {Toppings}.")
            .Field(nameof(SandwichOrder.Sauces))
            .Field(new FieldReflector<SandwichOrder>(nameof(Specials))
                .SetType(null)
                .SetActive((state) => state.Length == LengthOptions.FootLong)
                .SetDefine(async (state, field) =>
            {
                field
                    .AddDescription("cookie", DynamicSandwich.FreeCookie)
                    .AddTerms("cookie", Language.GenerateTerms(DynamicSandwich.FreeCookie, 2))
                    .AddDescription("drink", DynamicSandwich.FreeDrink)
                    .AddTerms("drink", Language.GenerateTerms(DynamicSandwich.FreeDrink, 2));
                return true;
            }))
            .Confirm(async (state) =>
            {
                var cost = 0.0;
                switch (state.Length)
                {
                    case LengthOptions.SixInch: cost = 5.0; break;
                    case LengthOptions.FootLong: cost = 6.50; break;
                }
                return new PromptAttribute(string.Format(DynamicSandwich.Cost, cost) + "||");
            })
            .Field(nameof(SandwichOrder.DeliveryAddress),
                validate: async (state, response) =>
            {
                var result = new ValidationResult { IsValid = true, Value = response };
                var address = (response as string).Trim();
                if (address.Length > 0 && address[0] < '0' || address[0] > '9')

```

```
        {
            result.Feedback = DynamicSandwich.BadAddress;
            result.IsValid = false;
        }
        return result;
    })
    .Field(nameof(SandwichOrder.DeliveryTime), "What time do you want your sandwich delivered?
{|||}")
    .Confirm("Do you want to order your {Length} {Sandwich} on {Bread} {&Bread} with {{Cheese}
{Toppings} {Sauces}}} to be sent to {DeliveryAddress} {at {DeliveryTime:t}}?")
    .AddRemainingFields()
    .Message("Thanks for ordering a sandwich!")
    .OnCompletion(processOrder);
builder.Configuration.DefaultPrompt.ChoiceStyle = ChoiceStyleOptions.Auto;
form = builder.Build();
_forms[culture] = form;
}
return form;
}
```

This snippet shows the resulting interaction between bot and user when `CurrentUICulture` is **French**.

Bienvenue sur le bot d'ordre "sandwich" !

Quel genre de "sandwich" vous souhaitez sur votre "sandwich"?

1. BLT
2. Jambon Forêt Noire
3. Poulet Buffalo
4. Faire fondre le poulet et Bacon Ranch
5. Combo de coupe à froid
6. Boulette de viande Marinara
7. Poulet rôti au four
8. Rôti de boeuf
9. Rotisserie poulet
10. Italienne piquante
11. Bifteck et fromage
12. Oignon doux Teriyaki
13. Thon
14. Poitrine de dinde
15. Veggie

> 2

Quel genre de longueur vous souhaitez sur votre "sandwich"?

1. Six pouces
2. Pied Long

> ?

* Vous renseignez le champ longueur.Réponses possibles:

* Vous pouvez saisir un numéro 1-2 ou des mots de la description. (Six pouces, ou Pied Long)

* Retourner à la question précédente.

* Assistance: Montrez les réponses possibles.

* Abandonner: Abandonner sans finir

* Recommencer remplir le formulaire. (Vos réponses précédentes sont enregistrées.)

* Statut: Montrer le progrès en remplissant le formulaire jusqu'à présent.

* Vous pouvez passer à un autre champ en entrant son nom. ("Sandwich", Longueur, Pain, Fromage, Nappages, Sauces, Adresse de remise, Délai de livraison, ou votre expérience aujourd'hui).

Quel genre de longueur vous souhaitez sur votre "sandwich"?

1. Six pouces
2. Pied Long

> 1

Quel genre de pain vous souhaitez sur votre "sandwich"?

1. Neuf grains de blé
2. Neuf grains miel avoine
3. Italien
4. Fromage et herbes italiennes
5. Pain plat

> neuf

Par pain "neuf" vouliez-vous dire (1. Neuf grains miel avoine, ou 2. Neuf grains de blé)

Sample code

For complete samples that show how to implement FormFlow using the Bot Builder SDK for .NET, see the [Multi-Dialog Bot sample](#) and the [Contoso Flowers Bot sample](#) in GitHub.

Additional resources

- [Basic features of FormFlow](#)
- [Advanced features of FormFlow](#)
- [Customize a form using FormBuilder](#)
- [Define a form using JSON schema](#)
- [Customize user experience with pattern language](#)
- [Bot Builder SDK for .NET Reference](#)

Define a form using JSON schema

11/2/2017 • 7 min to read • [Edit Online](#)

If you use a [C# class](#) to define the form when you create a bot with FormFlow, the form derives from the static definition of your type in C#. As an alternative, you may instead define the form by using [JSON schema](#). A form that is defined using JSON schema is purely data-driven; you can change the form (and therefore, the behavior of the bot) simply by updating the schema.

The JSON schema describes the fields within your [JObject](#) and includes annotations that control prompts, templates, and terms. To use JSON schema with FormFlow, you must add the

`Microsoft.Bot.Builder.FormFlow.Json` NuGet package to your project and import the

`Microsoft.Bot.Builder.FormFlow.Json` namespace.

Standard keywords

FormFlow supports these standard [JSON Schema](#) keywords:

KEYWORD	DESCRIPTION
type	Defines the type of data that the field contains.
enum	Defines the valid values for the field.
minimum	Defines the minimum numeric value allowed for the field (as described in NumericAttribute).
maximum	Defines the maximum numeric value allowed for the field (as described in NumericAttribute).
required	Defines which fields are required.
pattern	Validates string values (as described in PatternAttribute).

Extensions to JSON Schema

FormFlow extends the standard [JSON Schema](#) to support several additional properties.

Additional properties at the root of the schema

PROPERTY	VALUE
OnCompletion	C# script with arguments <code>(IDialogContext context, JObject state)</code> for completing the form.
References	References to include in scripts. For example, <code>[assemblyReference, ...]</code> . Paths should be absolute or relative to the current directory. By default, the script includes <code>Microsoft.Bot.Builder.dll</code> .

PROPERTY	VALUE
Imports	Imports to include in scripts. For example, <code>[import, ...]</code> . By default, the script includes the <code>Microsoft.Bot.Builder</code> , <code>Microsoft.Bot.Builder.Dialogs</code> , <code>Microsoft.Bot.Builder.FormFlow</code> , <code>Microsoft.Bot.Builder.FormFlow.Advanced</code> , <code>System.Collections.Generic</code> , and <code>System.Linq</code> namespaces.

Additional properties at the root of the schema or as peers of the type property

PROPERTY	VALUE
Templates	<code>{ TemplateUsage: { Patterns: [string, ...], <args> }, ... }</code>
Prompt	<code>{ Patterns:[string, ...] <args> }</code>

To specify templates and prompts in JSON schema, use the same vocabulary as defined by [TemplateAttribute](#) and [PromptAttribute](#). Property names and values in the schema should match the property names and values in the underlying C# enumeration. For example, this schema snippet defines a template that overrides the `TemplateUsage.NotUnderstood` template and specifies a `TemplateBaseAttribute.ChoiceStyle`:

```
"Templates":{ "NotUnderstood": { "Patterns": ["I don't get it"], "ChoiceStyle":"Auto"}}
```

Additional properties as peers of the type property

PROPERTY	CONTENTS	DESCRIPTION
DateTime	bool	Indicates whether field is a <code>DateTime</code> field.
Describe	string or object	Description of a field as described in DescribeAttribute .
Terms	<code>[string,...]</code>	Regular expressions for matching a field value as described in TermsAttribute .
MaxPhrase	int	Runs your terms through <code>Language.GenerateTerms(string, int)</code> to expand them.
Values	<code>'{ string: {Describe:string</code> <code>...}'</code>	object, Terms:[string, ...], MaxPhrase}, ...`
Active	script	C# script with arguments <code>(JObject state)->bool</code> to test whether the field, message, or confirmation is active.

PROPERTY	CONTENTS	DESCRIPTION
Validate	script	C# script with arguments (JObject state, object value)->ValidateResult for validating a field value.
Define	script	C# script with arguments (JObject state, Field<JObject> field) for dynamically defining a field.
Next	script	C# script with arguments (object value, JObject state) for determining the next step after filling in a field.
Before	`[confirm]	message, ...`
After	`[confirm]	message, ...`
Dependencies	[string, ...]	Fields that this field, message, or confirmation depends on.

Use `{Confirm:script|[string, ...], ...templateArgs}` within the value of the **Before** property or the **After** property to define a confirmation by using either a C# script with argument `(JObject state)` or a set of patterns that will be randomly selected with optional template arguments.

Use `{Message:script|[string, ...] ...templateArgs}` within the value of the **Before** property or the **After** property to define a message by using either a C# script with argument `(JObject state)` or a set of patterns that will be randomly selected with optional template arguments.

Scripts

Several of the properties that are described above contain a script as the property value. A script can be any snippet of C# code that you might normally find in the body of a method. You can add references by using the **References** property and/or the **Imports** property. Special global variables include:

VARIABLE	DESCRIPTION
choice	Internal dispatch for the script to execute.
state	<code>JObject</code> form state bound for all scripts.
ifield	<code>IField<JObject></code> to allow reasoning over the current field for all scripts except Message/Confirm prompt builders.
value	Object value to be validated for Validate .
field	<code>Field<JObject></code> to allow dynamically updating a field in Define .
context	<code>IDialogContext</code> context to allow posting results in OnCompletion .

Fields that are defined via JSON schema have the same ability to extend or override the definitions programmatically as any other field. They can also be localized in the same way.

JSON schema example

The simplest way to define a form is to define everything, including any C# code, directly in the JSON schema. This example shows the JSON schema for the annotated sandwich bot that is described in [Customize a form using FormBuilder](#).

```
{
  "References": [ "Microsoft.Bot.Sample.AnnotatedSandwichBot.dll" ],
  "Imports": [ "Microsoft.Bot.Sample.AnnotatedSandwichBot.Resource" ],
  "type": "object",
  "required": [
    "Sandwich",
    "Length",
    "Ingredients",
    "DeliveryAddress"
  ],
  "Templates": {
    "NotUnderstood": {
      "Patterns": [ "I do not understand \'{0}\'.", "Try again, I don't get \'{0}\'." ]
    },
    "EnumSelectOne": {
      "Patterns": [ "What kind of {&} would you like on your sandwich? {||}" ],
      "ChoiceStyle": "Auto"
    }
  },
  "properties": {
    "Sandwich": {
      "Prompt": { "Patterns": [ "What kind of {&} would you like? {||}" ] },
      "Before": [ { "Message": [ "Welcome to the sandwich order bot!" ] } ],
      "Describe": { "Image": "https://placeholdit.imgix.net/~text?txtsize=16&txt=Sandwich&w=125&h=40&txttrack=0&txtclr=000&txtfont=bold" },
      "type": [
        "string",
        "null"
      ],
      "enum": [
        "BLT",
        "BlackForestHam",
        "BuffaloChicken",
        "ChickenAndBaconRanchMelt",
        "ColdCutCombo",
        "MeatballMarinara",
        "OvenRoastedChicken",
        "RoastBeef",
        "RotisserieStyleChicken",
        "SpicyItalian",
        "SteakAndCheese",
        "SweetOnionTeriyaki",
        "Tuna",
        "TurkeyBreast",
        "Veggie"
      ],
      "Values": {
        "RotisserieStyleChicken": {
          "Terms": [ "rotis\\w* style chicken" ],
          "MaxPhrase": 3
        }
      }
    },
    "Length": {
      "Prompt": {
        "Patterns": [ "What size of sandwich do you want? {||}" ]
      }
    }
  }
}
```

```

    "type": [
        "string",
        "null"
    ],
    "enum": [
        "SixInch",
        "FootLong"
    ]
},
"Ingredients": {
    "type": "object",
    "required": [ "Bread" ],
    "properties": {
        "Bread": {
            "type": [
                "string",
                "null"
            ],
            "Describe": {
                "Title": "Sandwich Bot",
                "SubTitle": "Bread Picker"
            },
            "enum": [
                "NineGrainWheat",
                "NineGrainHoneyOat",
                "Italian",
                "ItalianHerbsAndCheese",
                "Flatbread"
            ]
        },
        "Cheese": {
            "type": [
                "string",
                "null"
            ],
            "enum": [
                "American",
                "MontereyCheddar",
                "Pepperjack"
            ]
        },
        "Toppings": {
            "type": "array",
            "items": {
                "type": "integer",
                "enum": [
                    "Everything",
                    "Avocado",
                    "BananaPeppers",
                    "Cucumbers",
                    "GreenBellPeppers",
                    "Jalapenos",
                    "Lettuce",
                    "Olives",
                    "Pickles",
                    "RedOnion",
                    "Spinach",
                    "Tomatoes"
                ],
                "Values": {
                    "Everything": { "Terms": [ "except", "but", "not", "no", "all", "everything" ] }
                }
            },
            "Validate": "var values = ((List<object>) value).OfType<string>(); var result = new ValidateResult {IsValid = true, Value = values} ; if (values != null && values.Contains(\"Everything\")) { result.Value = (from topping in new string[] { \"Avocado\", \"BananaPeppers\", \"Cucumbers\", \"GreenBellPeppers\", \"Jalapenos\", \"Lettuce\", \"Olives\", \"Pickles\", \"RedOnion\", \"Spinach\", \"Tomatoes\"} where !values.Contains(topping) select topping).ToList();} return result;",
            "After": [ { "Message": [ "For sandwich toppings you have selected {Ingredients.Toppings}." ] } ]
        }
    }
}

```

```

    "Sauces": [
      "type": [
        "array",
        "null"
      ],
      "items": {
        "type": "string",
        "enum": [
          "ChipotleSouthwest",
          "HoneyMustard",
          "LightMayonnaise",
          "RegularMayonnaise",
          "Mustard",
          "Oil",
          "Pepper",
          "Ranch",
          "SweetOnion",
          "Vinegar"
        ]
      }
    }
  },
  "Specials": {
    "Templates": {
      "NoPreference": { "Patterns": [ "None" ] }
    },
    "type": [
      "string",
      "null"
    ],
    "Active": "return (string) state[\"Length\"] == \"FootLong\";",
    "Define": "field.GetType(null).AddDescription(\"cookie\",  

DynamicSandwich.FreeCookie).AddTerms(\"cookie\", Language.GenerateTerms(DynamicSandwich.FreeCookie,  

2)).AddDescription(\"drink\", DynamicSandwich.FreeDrink).AddTerms(\"drink\",  

Language.GenerateTerms(DynamicSandwich.FreeDrink, 2)); return true;",
    "After": [ { "Confirm": "var cost = 0.0; switch ((string) state[\"Length\"]) { case \"SixInch\": cost =  

5.0; break; case \"FootLong\": cost=6.50; break;} return new PromptAttribute($\"Total for your sandwich is  

{cost:C2} is that ok?\"); } ]
  },
  "DeliveryAddress": {
    "type": [
      "string",
      "null"
    ],
    "Validate": "var result = new ValidationResult{ IsValid = true, Value = value}; var address = (value as  

string).Trim(); if (address.Length > 0 && (address[0] < '0' || address[0] > '9')) {result.Feedback =  

DynamicSandwich.BadAddress; result.IsValid = false; } return result;"
  },
  "PhoneNumber": {
    "type": [ "string", "null" ],
    "pattern": "(\\(\\d{3}\\))?\\s*\\d{3}(-|\\s*)\\d{4}"
  },
  "DeliveryTime": {
    "Templates": {
      "StatusFormat": {
        "Patterns": [ "{&}:{:t}" ],
        "FieldCase": "None"
      }
    },
    "DateTime": true,
    "type": [
      "string",
      "null"
    ],
    "After": [ { "Confirm": [ "Do you want to order your {Length} {Sandwich} on {Ingredients.Bread}  

{&Ingredients.Bread} with {[{Ingredients.Cheese} {Ingredients.Toppings} {Ingredients.Sauces}] to be sent to  

{DeliveryAddress} at {DeliveryTime}?" ] ]
  }
]
}

```

```
    "DeliveryAddress": {
        "Rating": {
            "Describe": "your experience today",
            "type": [
                "number",
                "null"
            ],
            "minimum": 1,
            "maximum": 5,
            "After": [ { "Message": [ "Thanks for ordering your sandwich!" ] } ]
        }
    },
    "OnCompletion": "await context.PostAsync(\"We are currently processing your sandwich. We will message you the status.\");"
}
```

Implement FormFlow with JSON schema

To implement FormFlow with a JSON schema, use `FormBuilderJson`, which supports the same fluent interface as `FormBuilder`. This code example shows how to implement the JSON schema for the annotated sandwich bot that is described in [Customize a form using FormBuilder](#).

```
public static IForm< JObject> BuildJsonForm()
{
    using (var stream =
Assembly.GetExecutingAssembly().GetManifestResourceStream("Microsoft.Bot.Sample.AnnnotatedSandwichBot.Annotate
dSandwich.json"))
    {
        var schema = JObject.Parse(new StreamReader(stream).ReadToEnd());
        return new FormBuilderJson(schema)
            .AddRemainingFields()
            .Build();
    }
    ...
}
```

Sample code

For complete samples that show how to implement FormFlow using the Bot Builder SDK for .NET, see the [Multi-Dialog Bot sample](#) and the [Contoso Flowers Bot sample](#) in GitHub.

Additional resources

- [Basic features of FormFlow](#)
- [Advanced features of FormFlow](#)
- [Customize a form using FormBuilder](#)
- [Localize form content](#)
- [Customize user experience with pattern language](#)
- [Bot Builder SDK for .NET Reference](#)

Build a speech-enabled bot with Cortana skills

11/2/2017 • 7 min to read • [Edit Online](#)

The Bot Builder SDK for .NET enables you to build speech-enabled bot by connecting it to the Cortana channel as a Cortana skill.

TIP

For more information on what a skill is, and what they can do, see [The Cortana Skills Kit](#).

Creating a Cortana skill using Bot Framework requires very little Cortana-specific knowledge and primarily consists of building a bot. One of the likely key differences from other bots that you may have created in the past is that Cortana has both a visual and an audio component. For the visual component, Cortana provides an area of the canvas for rendering content such as cards. For the audio component, you provide text or SSML in your bot's messages, which Cortana reads to the user, giving your bot a voice.

NOTE

Cortana is available on many different devices. Some have a screen while others, like a standalone speaker, might not. You should make sure that your bot is capable of handling both scenarios. See [Cortana-specific entities](#) to learn how to check device information.

Adding speech to your bot

Spoken messages from your bot are represented as Speech Synthesis Markup Language (SSML). The Bot Builder SDK lets you include SSML in your bot's responses to control what the bot says, in addition to what it shows. You can also control the state of Cortana's microphone, by specifying whether your bot is accepting, expecting, or ignoring user input.

Set the `Speak` property of the `IMessageActivity` object to specify a message for Cortana to say. If you specify plain text, Cortana determines how the words are pronounced.

```
Activity reply = activity.CreateReply("This is the text that Cortana displays.");
reply.Speak = "This is the text that Cortana will say.;"
```

If you want more control over pitch, tone, and emphasis, format the `Speak` property as [Speech Synthesis Markup Language \(SSML\)](#).

The following code example specifies that the word "text" should be spoken with a moderate amount of emphasis:

```
Activity reply = activity.CreateReply("This is the text that will be displayed.");
reply.Speak = "<speak version=\"1.0\" xmlns=\"http://www.w3.org/2001/10/synthesis\" xml:lang=\"en-US\">This is
the <emphasis level=\"moderate\">text</emphasis> that will be spoken.</speak>";
```

The `InputHint` property helps indicate to Cortana whether your bot is expecting input. The default value is `ExpectingInput` for a prompt, and `AcceptingInput` for other types of responses.

VALUE	DESCRIPTION
AcceptingInput	Your bot is passively ready for input but is not waiting on a response. Cortana accepts input from the user if the user holds down the microphone button.
ExpectingInput	Indicates that the bot is actively expecting a response from the user. Cortana listens for the user to speak into the microphone.
IgnoringInput	Cortana is ignoring input. Your bot may send this hint if it is actively processing a request and will ignore input from users until the request is complete.

This example shows how to let Cortana know that user input is expected. The microphone will be left open.

```
// Add an InputHint to let Cortana know to expect user input
Activity reply = activity.CreateReply("This is the text that will be displayed.");
reply.Speak = "This is the text that will be spoken.";
reply.InputHint = InputHints.ExpectingInput;
```

Display cards in Cortana

In addition to spoken responses, Cortana can also display card attachments. Cortana supports the following rich cards:

CARD TYPE	DESCRIPTION
HeroCard	A card that typically contains a single large image, one or more buttons, and text.
ThumbnailCard	A card that typically contains a single thumbnail image, one or more buttons, and text.
ReceiptCard	A card that enables a bot to provide a receipt to the user. It typically contains the list of items to include on the receipt, tax and total information, and other text.
SignInCard	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.

See [Card design best practices](#) to see what these cards look like inside Cortana. For an example of how to use a rich card in a bot, see [Add rich card attachments to messages](#).

Sample: RollerSkill

The code in the following sections comes from a sample Cortana skill for rolling dice. Download the full code for the bot from the [BotBuilder-Samples repository](#).

You invoke the skill by saying its [invocation name](#) to Cortana. For the roller skill, after you [add the bot to the Cortana channel](#) and register it as a Cortana skill, you can invoke it by telling Cortana "Ask Roller" or "Ask Roller to roll dice".

Explore the code

To invoke the appropriate dialogs, the activity handlers defined in `RootDispatchDialog.cs` use regular expressions to match the user's input. For example, the handler in the following example is triggered if the user says something like "I'd like to roll some dice". Synonyms are included in the regular expression so that similar utterances will trigger the dialog.

```
[RegexPattern("(roll|role|throw|shoot).*(dice|die|dye|bones)")]  
[RegexPattern("new game")]  
[ScorableGroup(1)]  
public async Task NewGame(IDialogContext context, IActivity activity)  
{  
    context.Call(new CreateGameDialog(), AfterGameCreated);  
}
```

The `CreateGameDialog` dialog sets up a custom game for the bot to play. It uses a `PromptDialog` to ask the user how many sides they want the dice to have and then how many should be rolled. Note that the `PromptOptions` object that is used to initialize the prompt contains a `speak` property for the spoken version of the prompt.

```

[Serializable]
public class CreateGameDialog : IDialog<GameData>
{
    public async Task StartAsync(IDialogContext context)
    {
        context.UserData.SetValue<GameData>(Utils.GameDataKey, new GameData());

        var descriptions = new List<string>() { "4 Sides", "6 Sides", "8 Sides", "10 Sides", "12 Sides",
"20 Sides" };
        var choices = new Dictionary<string, IReadOnlyList<string>>()
        {
            { "4", new List<string> { "four", "for", "4 sided", "4 sides" } },
            { "6", new List<string> { "six", "sex", "6 sided", "6 sides" } },
            { "8", new List<string> { "eight", "8 sided", "8 sides" } },
            { "10", new List<string> { "ten", "10 sided", "10 sides" } },
            { "12", new List<string> { "twelve", "12 sided", "12 sides" } },
            { "20", new List<string> { "twenty", "20 sided", "20 sides" } }
        };

        var promptOptions = new PromptOptions<string>(
            Resources.ChooseSides,
            choices: choices,
            descriptions: descriptions,
            speak: SSMLHelper.Speak(Utils.RandomPick(Resources.ChooseSidesSSML))); // spoken prompt

        PromptDialog.Choice(context, this.DiceChoiceReceivedAsync, promptOptions);
    }

    private async Task DiceChoiceReceivedAsync(IDialogContext context, IAwaitable<string> result)
    {
        GameData game;
        if (context.UserData.TryGetValue<GameData>(Utils.GameDataKey, out game))
        {
            int sides;
            if (int.TryParse(await result, out sides))
            {
                game.Sides = sides;
                context.UserData.SetValue<GameData>(Utils.GameDataKey, game);
            }

            var promptText = string.Format(Resources.ChooseCount, sides);

            var promptOption = new PromptOptions<long>(promptText, choices: null, speak:
SSMLHelper.Speak(Utils.RandomPick(Resources.ChooseCountSSML)));

            var prompt = new PromptDialog.PromptInt64(promptOption);
            context.Call<long>(prompt, this.DiceNumberReceivedAsync);
        }
    }

    private async Task DiceNumberReceivedAsync(IDialogContext context, IAwaitable<long> result)
    {
        GameData game;
        if (context.UserData.TryGetValue<GameData>(Utils.GameDataKey, out game))
        {
            game.Count = await result;
            context.UserData.SetValue<GameData>(Utils.GameDataKey, game);
        }

        context.Done(game);
    }
}

```

The `PlayGameDialog` renders the results both by displaying them in a `HeroCard` and building a spoken message to say using the `Speak` method.

```

[Serializable]
public class PlayGameDialog : IDialog<object>
{
    private const string RollAgainOptionValue = "roll again";

    private const string NewGameOptionValue = "new game";

    private GameData gameData;

    public PlayGameDialog(GameData gameData)
    {
        this.gameData = gameData;
    }

    public async Task StartAsync(IDialogContext context)
    {
        if (this.gameData == null)
        {
            if (!context.UserData.TryGetValue<GameData>(Utils.GameDataKey, out this.gameData))
            {
                // User started session with "roll again" so let's just send them to
                // the 'CreateGameDialog'
                context.Done<object>(null);
            }
        }

        int total = 0;
        var randomGenerator = new Random();
        var rolls = new List<int>();

        // Generate Rolls
        for (int i = 0; i < this.gameData.Count; i++)
        {
            var roll = randomGenerator.Next(1, this.gameData.Sides);
            total += roll;
            rolls.Add(roll);
        }

        // Format rolls results
        var result = string.Join(" . ", rolls.ToArray());
        bool multiLine = rolls.Count > 5;

        var card = new HeroCard()
        {
            Subtitle = string.Format(
                this.gameData.Count > 1 ? Resources.CardSubtitlePlural : Resources.CardSubtitleSingular,
                this.gameData.Count,
                this.gameData.Sides),
            Buttons = new List<CardAction>()
            {
                new CardAction(ActionTypes.ImBack, "Roll Again", value: RollAgainOptionValue),
                new CardAction(ActionTypes.ImBack, "New Game", value: NewGameOptionValue)
            }
        };

        if (multiLine)
        {
            card.Text = result;
        }
        else
        {
            card.Title = result;
        }

        var message = context.MakeMessage();
        message.Attachments = new List<Attachment>()
        {
            card.ToAttachment()
        };
    }
}

```

```

};

// Determine bots reaction for speech purposes
string reaction = "normal";

var min = this.gameData.Count;
var max = this.gameData.Count * this.gameData.Sides;
var score = total / max;
if (score == 1)
{
    reaction = "Best";
}
else if (score == 0)
{
    reaction = "Worst";
}
else if (score <= 0.3)
{
    reaction = "Bad";
}
else if (score >= 0.8)
{
    reaction = "Good";
}

// Check for special craps rolls
if (this.gameData.Type == "Craps")
{
    switch (total)
    {
        case 2:
        case 3:
        case 12:
            reaction = "CrapsLose";
            break;
        case 7:
            reaction = "CrapsSeven";
            break;
        case 11:
            reaction = "CrapsEleven";
            break;
        default:
            reaction = "CrapsRetry";
            break;
    }
}

// Build up spoken response
var spoken = string.Empty;
if (this.gameData.Turns == 0)
{
    spoken +=
    Utils.RandomPick(Resources.ResourceManager.GetString($"Start{this.gameData.Type}GameSSML"));
}

spoken += Utils.RandomPick(Resources.ResourceManager.GetString($"{reaction}RollReactionSSML"));

message.Speak = SSMLHelper.Speak(spoken);

// Increment number of turns and store game to roll again
this.gameData.Turns++;
context.UserData.SetValue<GameData>(Utils.GameDataKey, this.gameData);

// Send card and bots reaction to user.
message.InputHint = InputHints.AcceptingInput;
await context.PostAsync(message);

context.Done<object>(null);
}

```

}

Next steps

If your bot is running locally or deployed in the cloud, you can invoke it from Cortana. See [Test a Cortana skill](#) for the steps required to try out your Cortana skill.

Additional resources

- [The Cortana Skills Kit](#)
- [Add speech to messages](#)
- [SSML Reference](#)
- [Voice design best practices for Cortana](#)
- [Card design best practices for Cortana](#)
- [Cortana Dev Center](#)
- [Testing and debugging best practices for Cortana](#)
- [Bot Builder SDK for .NET Reference](#)

Conduct audio calls with Skype

11/2/2017 • 3 min to read • [Edit Online](#)

If you are building a bot for Skype, your bot can communicate with users via audio call. Audio calls are useful when the user does not want to or cannot provide input by typing, tapping, or clicking.

A bot may support other user controls such as rich cards or text in addition to audio calls, or communicate through audio calls only.

The architecture for a bot that supports audio calls is very similar to that of a typical bot. The following code samples show how to enable support for audio calls via Skype with the Bot Builder SDK for .NET.

Enable support for audio calls

To enable a bot to support audio calls, define the `CallingController`.

```
[BotAuthentication]
[RoutePrefix("api/calling")]
public class CallingController : ApiController
{
    public CallingController() : base()
    {
        CallingConversation.RegisterCallingBot(callingBotService => new IVRBot(callingBotService));
    }

    [Route("callback")]
    public async Task<HttpResponseMessage> ProcessCallingEventAsync()
    {
        return await CallingConversation.SendAsync(this.Request, CallRequestType.CallingEvent);
    }

    [Route("call")]
    public async Task<HttpResponseMessage> ProcessIncomingCallAsync()
    {
        return await CallingConversation.SendAsync(this.Request, CallRequestType.IncomingCall);
    }
}
```

NOTE

In addition to the `CallingController`, which supports audio calls, a bot may also contain a `MessagesController` to support messages. Providing both options allows users to interact with the bot in the way that they prefer.

Answer the call

The `ProcessIncomingCallAsync` task will execute whenever a user initiates a call to this bot from Skype. The constructor registers the `IVRBot` class, which has a predefined handler for the `incomingCallEvent`.

The first action within the workflow should determine if the bot answers or rejects the incoming call. This workflow instructs the bot to answer the incoming call and then play a welcome message.

```

private Task OnIncomingCallReceived(IncomingCallEvent incomingCallEvent)
{
    this.callStateMap[incomingCallEvent.IncomingCall.Id] = new
CallState(incomingCallEvent.IncomingCall.Participants);

    incomingCallEvent.ResultingWorkflow.Actions = new List<ActionBase>
{
    new Answer { OperationId = Guid.NewGuid().ToString() },
    GetPromptForText(WelcomeMessage)
};

    return Task.FromResult(true);
}

```

After the bot answers

If the bot answers the call, subsequent actions specified within the workflow will instruct the **Skype Bot Platform for Calling** to play prompt, record audio, recognize speech, or collect digits from a dial pad. The final action of the workflow might be to end the call.

This code sample defines a handler that will set up a menu after the welcome message completes.

```

private Task OnPlayPromptCompleted(PlayPromptOutcomeEvent playPromptOutcomeEvent)
{
    var callState = this.callStateMap[playPromptOutcomeEvent.ConversationResult.Id];
    SetupInitialMenu(playPromptOutcomeEvent.ResultingWorkflow);
    return Task.FromResult(true);
}

```

The `CreateIvrOptions` method defines that menu that will be presented to the user.

```

private static Recognize CreateIvrOptions(string textToBeRead, int numberofOptions, bool includeBack)
{
    if (numberofOptions > 9)
    {
        throw new Exception("too many options specified");
    }

    var choices = new List<RecognitionOption>();

    for (int i = 1; i <= numberofOptions; i++)
    {
        choices.Add(new RecognitionOption { Name = Convert.ToString(i), DtmfVariation = (char)('0' + i) });
    }

    if (includeBack)
    {
        choices.Add(new RecognitionOption { Name = "#", DtmfVariation = '#' });
    }

    var recognize = new Recognize
    {
        OperationId = Guid.NewGuid().ToString(),
        PlayPrompt = GetPromptForText(textToBeRead),
        BargeInAllowed = true,
        Choices = choices
    };

    return recognize;
}

```

The `RecognitionOption` class defines both the spoken answer as well as the corresponding Dual-Tone Multi-Frequency (DTMF) variation. DTMF enables the user to answer by typing the corresponding digits on the keypad instead of speaking.

The `OnRecognizeCompleted` method processes the user's selection, and the input parameter `recognizeOutcomeEvent` contains the value of the user's selection.

```
private Task OnRecognizeCompleted(RecognizeOutcomeEvent recognizeOutcomeEvent)
{
    var callState = this.callStateMap[recognizeOutcomeEvent.ConversationResult.Id];
    ProcessMainMenuSelection(recognizeOutcomeEvent, callState);
    return Task.FromResult(true);
}
```

Support natural language

The bot can also be designed to support natural language responses. The **Bing Speech API** enables the bot to recognize words in the user's spoken reply.

```
private async Task OnRecordCompleted(RecordOutcomeEvent recordOutcomeEvent)
{
    recordOutcomeEvent.ResultingWorkflow.Actions = new List<ActionBase>
    {
        GetPromptForText(EndingMessage),
        new Hangup { OperationId = Guid.NewGuid().ToString() }
    };

    // Convert the audio to text
    if (recordOutcomeEvent.RecordOutcome.Outcome == Outcome.Success)
    {
        var record = await recordOutcomeEvent.RecordedContent;
        string text = await this.GetTextFromAudioAsync(record);

        var callState = this.callStateMap[recordOutcomeEvent.ConversationResult.Id];

        await this.SendSTTResultToUser("We detected the following audio: " + text, callState.Participants);
    }

    recordOutcomeEvent.ResultingWorkflow.Links = null;
    this.callStateMap.Remove(recordOutcomeEvent.ConversationResult.Id);
}
```

Sample code

For a complete sample that shows how to support audio calls with Skype using the Bot Builder SDK for .NET, see the [Skype Calling Bot sample](#) in GitHub.

Additional resources

- [Bot Builder SDK for .NET Reference](#)
- [Skype Calling Bot sample \(GitHub\)](#)

Real-time media calling with Skype

8/7/2017 • 4 min to read • [Edit Online](#)

The Real-Time Media Platform for Bots adds a new dimension to how bots can interact with users by enabling real-time voice, video and screen sharing modalities. This provides the capability to build compelling and interactive entertainment, educational, and assistance bots. Users communicate with real-time media bots using Skype.

This is an advanced capability which allows the bot to send and receive voice and video content *frame by frame*. The bot has "raw" access to the voice, video and screen-sharing real-time modalities. For example, as the user speaks, the bot will receive 50 audio frames per second, with each frame containing 20 milliseconds (ms) of audio. The bot can perform real-time speech recognition as the audio frames are received, rather than having to wait for a recording after the user has stopped speaking. The bot can also send high-definition-resolution video to the user at 30 frames per second, along with audio.

The Real-Time Media Platform for Bots works with the Skype Calling Cloud to take care of call setup and media session establishment, enabling the bot to engage in a voice and (optionally) video conversation with a Skype caller. The platform provides a simple "socket"-like API for the bot to send and receive media, and handles the real-time encoding and decoding of media, using codecs such as SILK for audio and H.264 for video. A real-time media bot may also participate in Skype group video calls with multiple participants.

This article introduces key concepts related to building a bot that can conduct real-time audio and video calls with Skype and provides links to relevant developer resources.

Media session

When a real-time media bot answers an incoming Skype call, it decides whether to support both audio and video modalities, or just audio. For each supported modality, the bot can either both send and receive media, receive only, or send only. For example, a bot may wish to both send and receive audio, but only send video (as it does not want to receive the video of the Skype caller). The set of audio and video modalities established between the Skype caller and the bot is called the **media session**.

There are two video modalities supported: "main video" and "screen sharing". Main video transmits the video the bot generates, or plays back, to the caller, and transmits the video of the caller (typically from the user's webcam) to the bot. The screen sharing modality allows the caller to share his or her screen (as a video) with the bot. The bot cannot send screen sharing video to the user.

When joined to a multiparty Skype **group video call**, the real-time media bot can support receiving multiple main video streams simultaneously. This allows the bot to "see" more than one participant in the group video call.

Frames and frame rate

A real-time media bot interacts directly with the audio and video modalities of a Skype call. This means the bot is sending and/or receiving media as a sequence of **frames**, where each frame represents a unit of content. One second of audio may be transmitted as a sequence of 50 frames, with each frame containing 20 milliseconds (ms) (1/50th of a second) of content. One second worth of video may be sliced as a sequence of 30 still images, each intended to be viewed for just 33ms (1/30th of a second) before the next video frame is displayed. The number of frames transmitted or rendered per second is called the **frame rate**. "30fps" indicates 30 frames per second.

Audio format

Each second of audio is represented as 16,000 **samples**, with each sample storing 16-bits of data. A 20ms audio

frame contains 320 samples (640 bytes of data).

Video format

There are several formats supported for video. Two key properties of a video format are its **frame size** and **color format**. Supported frame sizes include 640x360 ("360p"), 1280x720 ("720p"), and 1920x1080 ("1080p"). Supported color formats include NV12 (12 bits per pixel) and RGB24 (24 bits per pixel).

A "720p" video frame contains 921,600 pixels (1280 times 720). In the RGB24 color format, each pixel is represented as 3 bytes (24-bits) comprised of one byte each of red, green, and blue color components. Therefore, a single 720p RGB24 video frame requires 2,764,800 bytes of data (921,600 pixels times 3 bytes/pixel). At a frame rate of 30fps, sending 720p RGB24 video frames means processing approximately 80 MB/s of content (which is substantially compressed by the H.264 video codec before network transmission).

Active and dominant speakers

When joined to a Skype group video call consisting of multiple participants, the bot can identify which participants are currently speaking. **Active speakers** identify which participants are being heard in each received audio frame. **Dominant speakers** identify which participants are currently most active (or "dominant") in the group conversation, even though their voice may not be heard in every audio frame. The set of dominant speakers can change as different participants take turns speaking.

Video subscription

In a call with a single Skype caller, the bot will automatically receive the video of the caller (if the bot is receive enabled for main video). In a multiparty Skype group video call, the bot must indicate to the real-time media platform which participants it wants to see. A video subscription is a request by the bot to receive a participant's video. As the participants in a group video call conduct their conversation, a bot may modify its desired video subscriptions based on updates of the dominant speaker set.

Developer resources

SDKs

To develop a real-time media bot, you must install these NuGet packages within your Visual Studio project:

- [Bot Builder SDK for .NET](#)
- [Bot Builder Real-Time Media Calling for .NET](#)
- [Microsoft.Skype.Bots.Media .NET library](#)

Code samples

The [BotBuilder-RealTimeMediaCalling](#) GitHub repository contains samples that show how to build real-time media bots for Skype.

Requirements and considerations for real-time media bots

6/13/2017 • 3 min to read • [Edit Online](#)

Not all guidance that applies to developing messaging and IVR calling bots applies equally to building real-time media bots. This article describes some of the important requirements and considerations for developing real-time media bots.

NOTE

Because the Real-Time Media Platform for Bots is a Preview technology, the guidance in this article is subject to change.

Real-time media bot development requires C#/.NET and Windows Server

- The real-time media bot requires the `Microsoft.Skype.Bots.Media` .NET library (available via [NuGet](#)) to access the audio and video media, and the bot must be deployed on a Windows Server machine (or Windows Server guest OS in Azure). Therefore, the bot should be developed with C# and the Standard .NET Framework, and deployed in Azure. You cannot use C++ and Node.js APIs to access real-time media.
- The real-time media bot must be running on a recent version of the `Microsoft.Skype.Bots.Media` .NET library. The bot should use either the newest available version of the [NuGet](#) package, or a version which is not more than three months old. Older versions of the library will be deprecated and will not work after a few months.

Real-time media calls stay on the machine where they were created

- Real-time media bots are very stateful. The real-time media call is pinned to the virtual machine (VM) instance which accepted the incoming call: voice and video media from the Skype caller will flow to that VM instance, and media the bot sends back to the caller must also originate from that VM.
- If there are any real-time media calls in progress when the VM is stopped, those calls will be abruptly terminated. If the bot can know of the pending VM shutdown, it can try to "gracefully" end the calls.

Real-time media bots must be directly accessible on the Internet

- Each VM instance hosting a real-time media bot must be directly accessible from the Internet. For real-time media bots hosted in Azure, each VM instance must have an instance-level public IP address (ILPIP). For information about obtaining and configuring an ILPIP, see [Instance level public IP \(Classic\) overview](#). By default, an Azure subscription can obtain 5 ILPIP addresses; please contact Azure support to increase your subscription's quota.
- Because of the public IP address requirement, real-time media bots must be hosted in either an "IaaS" Azure Virtual Machine, or a "classic" Azure Cloud Service. Other runtime environments, such as Azure Bot Service or Service Fabric with VM Scale Sets, are not supported, as these do not support ILPIP.
- Real-time media bots are also not supported by the [Bot Framework Emulator](#).

Scalability and performance considerations

- Real-time media bots require more compute and network (bandwidth) capacity than messaging bots and may incur significantly higher operational costs. A real-time media bot developer must carefully measure the bot's scalability, and ensure the bot does not accept more simultaneous calls than it can manage. A video-enabled bot may be able to sustain only one or two concurrent real-time media calls per CPU core.
- The current Preview release of Real-Time Media Platform for Bots has certain scalability limitations the bot developer must be aware of (these may be improved in future releases):
 1. A single VM instance may not have more than 10 video sockets created at any single time.
 2. The Real-Time Media Platform does not currently take advantage of any Graphics Processing Unit (GPU) available on the VM to off-load H.264 video encoding/decoding. Instead, video encode and decode are done in software on the CPU. If a GPU is available, the bot may take advantage of it for its own graphics rendering (e.g., if the bot is using a 3D graphics engine).
- The VM instance hosting the real-time media bot must have at least 2 CPU cores. For Azure, a Dv2-series virtual machine is recommended. Detailed information about Azure VM types is available in the [Azure documentation](#).

Build a real-time media bot for Skype

8/7/2017 • 10 min to read • [Edit Online](#)

The Real-Time Media Platform for Bots is an advanced capability which allows the bot to send and receive voice and video content frame by frame. The bot has "raw" access to the voice, video and screen-sharing real-time modalities. This article provides an overview of building an audio/video calling bot and accessing the real-time modalities.

In this article, the bot is running in an Azure Cloud Service, as either a Web Role or a Worker Role self-hosting the ASP.NET Web API framework.

IMPORTANT

This article content is preliminary; please refer to the Samples folder in the [BotBuilder-RealTimeMediaCalling GitHub](#) repository for the full code of sample real-time media bots.

Configure the service hosting the real-time media bot

In order to use the Real-Time Media Platform, the following service configurations are necessary.

- The bot must know the Fully-Qualified Domain Name (FQDN) of its service. This is not provided by the Azure RoleEnvironment API; instead, the FQDN must be stored in the bot's Cloud Service configuration and read during service startup.
- The bot service must have a certificate issued by a recognized Certificate Authority. The thumbprint of the certificate must be stored in the bot's Cloud Service configuration and read during service startup.
- A public [instance input endpoint](#) must be provisioned. This assigns a unique public port to each virtual machine (VM) instance in the bot's service. This port is used by the Real-Time Media Platform to communicate with the Skype Calling Cloud.

```
<InstanceInputEndpoint name="InstanceMediaControlEndpoint" protocol="tcp" localPort="20100">
  <AllocatePublicPortFrom>
    <FixedPortRange max="20200" min="20101" />
  </AllocatePublicPortFrom>
</InstanceInputEndpoint>
```

It is also useful to create another instance input endpoint for call related callbacks and notifications. Using an instance input endpoint ensures the callbacks and notifications are delivered to the same VM instance in the service deployment that is hosting the real-time media session for the call.

```
<InstanceInputEndpoint name="InstanceCallControlEndpoint" protocol="tcp" localPort="10100">
  <AllocatePublicPortFrom>
    <FixedPortRange max="10200" min="10101" />
  </AllocatePublicPortFrom>
</InstanceInputEndpoint>
```

- Each VM instance must have an instance-level public IP address (ILPIP). During startup, the bot must discover the ILPIP address assigned to each service instance. See [ILPIP](#) for more information about obtaining and configuring an ILPIP.

```

<NetworkConfiguration>
  <AddressAssignments>
    <InstanceAddress roleName="WorkerRole">
      <PublicIPs>
        <PublicIP name="InstancePublicIP" domainNameLabel="InstancePublicIP" />
      </PublicIPs>
    </InstanceAddress>
  </AddressAssignments>
</NetworkConfiguration>

```

- During service instance startup, the script `MediaPlatformStartupScript.bat` (provided as a part of Nuget package) needs to be run as a Startup task under elevated privileges. The script execution must complete before the platform's initialization method is called.

```

<Startup>
  <Task commandLine="MediaPlatformStartupScript.bat" executionContext="elevated" taskType="simple" />
</Startup>

```

Initialize the media platform on service startup

During service instance startup, the Real-Time Media Platform must be initialized. This must be done only once before the bot can accept any audio/video Skype calls on that instance. Initialization of the media platform requires providing various service configuration settings, including the service's FQDN, the public ILPIP address, the instance input endpoint port value, and the bot's **Microsoft App ID**.

```

var mediaPlatformSettings = new MediaPlatformSettings()
{
  MediaPlatformInstanceSettings = new MediaPlatformInstanceSettings()
  {
    CertificateThumbprint = certificateThumbprint,
    InstanceInternalPort = instanceMediaControlEndpointInternalPort,
    InstancePublicIPAddress = instancePublicIPAddress,
    InstancePublicPort = instanceMediaControlEndpointPublicPort,
    ServiceFqdn = serviceFqdn
  },
  ApplicationId = MicrosoftAppId
};

MediaPlatform.Initialize(mediaPlatformSettings);

```

Register to receive incoming call requests

Define a `CallController` class. This enables the bot service to register for incoming Skype calls, and ensure callback and notification requests are forwarded to the appropriate `RealTimeMediaCall` object.

```

[BotAuthentication]
[RoutePrefix("api/calling")]
public class CallController : ApiController
{
    static CallController()
    {
        RealTimeMediaCalling.RegisterRealTimeMediaCallingBot(
            c => { return new RealTimeMediaCall(c); },
            new RealTimeMediaCallingBotServiceSettings()
        );
    }

    [Route("call")]
    public async Task<HttpResponseMessage> OnIncomingCallAsync()
    {
        // forwards the incoming call to the associated RealTimeMediaCall object
        return await RealTimeMediaCalling.SendAsync(this.Request, RealTimeMediaCallRequestType.IncomingCall);
    }

    [Route("callback")]
    public async Task<HttpResponseMessage> OnCallbackAsync()
    {
        // forwards the incoming callback to the associated RealTimeMediaCall object
        return await RealTimeMediaCalling.SendAsync(this.Request, RealTimeMediaCallRequestType.CallingEvent);
    }

    [Route("notification")]
    public async Task<HttpResponseMessage> OnNotificationAsync()
    {
        // forwards the incoming notification to the associated RealTimeMediaCall object
        return await RealTimeMediaCalling.SendAsync(this.Request,
            RealTimeMediaCallRequestType.NotificationEvent);
    }
}

```

`RealTimeMediaCallingBotServiceSettings` implements `IRealTimeMediaCallServiceSettings` and provides webhook links for callback and notification events.

Register for incoming events for the call

Define a `RealTimeMediaCall` class that implements `IRealTimeMediaCall`. For each call that is received by the bot, an instance of `RealTimeMediaCall` is created by the Bot Framework. The `IRealTimeMediaCallService` object passed to the constructor allows the bot to register for events to handle events associated with the real-time media call.

```

internal class RealTimeMediaCall : IRealTimeMediaCall
{
    public RealTimeMediaCall(IRealTimeMediaCallService callService)
    {
        if (callService == null)
            throw new ArgumentNullException(nameof(callService));

        CallService = callService;
        CorrelationId = callService.CorrelationId;
        CallId = CorrelationId + ":" + Guid.NewGuid().ToString();

        // Register for the call events
        CallService.OnIncomingCallReceived += OnIncomingCallReceived;
        CallService.OnAnswerAppHostedMediaCompleted += OnAnswerAppHostedMediaCompleted;
        CallService.OnCallStateChangeNotification += OnCallStateChangeNotification;
        CallService.OnRosterUpdateNotification += OnRosterUpdateNotification;
        CallService.OnCallCleanup += OnCallCleanup;
    }
}

```

Create audio and video sockets

Before the bot can accept an incoming audio/video Skype call, it must create `AudioSocket` and `VideoSocket` objects in order to support sending and receiving real-time media. (If the bot does not want to support video, then it should create only an `AudioSocket`.)

The bot must decide upfront which modalities it wants to support, and create the appropriate `AudioSocket` and `VideoSocket` objects. The bot cannot change what modalities it supports for the call after the incoming call is accepted.

For each `AudioSocket` and `VideoSocket`, the bot specifies whether the media socket is to support both sending and receiving media, or sending or receiving only. For example, the bot may wish to send and receive audio ("Sendrecv"), but send only for video ("Sendonly"). The bot must also specify what media formats are supported for each media socket. For an `AudioSocket`, the currently supported format is "Pcm16K": (signed) 16-bit PCM encoding, 16KHz sampling rate. For a `VideoSocket`, media formats for sending and receiving are specified separately. Only the "NV12" format is supported for receiving video, while several different formats are supported for sending.

```

\_audioSocket = new AudioSocket(new AudioSocketSettings
{
    StreamDirections = StreamDirection.Sendrecv,
    SupportedAudioFormat = AudioFormat.Pcm16K,
    CallId = correlationId
});

\_videoSocket = new VideoSocket(new VideoSocketSettings
{
    StreamDirections = StreamDirection.Sendrecv,
    ReceiveColorFormat = VideoColorFormat.NV12,
    SupportedSendVideoFormats = new VideoFormat\[\]
    {
        VideoFormat.Yuy2\_1280x720\_30Fps,
        VideoFormat.Yuy2\_720x1280\_30Fps,
    },
    CallId = correlationId
});

\_audioSocket.AudioMediaReceived += OnAudioMediaReceived;
\_audioSocket.AudioSendStatusChanged += OnAudioSendStatusChanged;
\_audioSocket.DominantSpeakerChanged += OnDominantSpeakerChanged;
\_videoSocket.VideoMediaReceived += OnVideoMediaReceived;
\_videoSocket.VideoSendStatusChanged += OnVideoSendStatusChanged;

```

Create a MediaConfiguration

Once the media sockets have been created, the bot must next create a [MediaConfiguration](#) object which is needed to associate the media sockets with an incoming audio/video Skype call.

```
var mediaConfiguration = MediaPlatform.CreateMediaConfiguration(\_audioSocket, \_videoSocket);
```

Answer an incoming audio/video call

The [OnIncomingCallReceived](#) event is invoked to allow the bot to accept the incoming Skype audio/video call. To do so, the bot creates an [AnswerAppHostedMedia](#) object with the [MediaConfiguration](#) object. The bot registers for notifications associated with this Skype call.

```

private Task OnIncomingCallReceived(RealTimeMediaIncomingCallEvent incomingCallEvent)
{
    // ... create Audio/VideoSocket objects and MediaConfiguration ...

    incomingCallEvent.RealTimeMediaWorkflow.Actions = new ActionBase\[\]
    {
        new AnswerAppHostedMedia
        {
            MediaConfiguration = mediaConfiguration,
            OperationId = Guid.NewGuid().ToString()
        }
    };
}

// subscribe for roster and call state changes
incomingCallEvent.RealTimeMediaWorkflow.NotificationSubscriptions = new NotificationType\[\]
{
    NotificationType.CallStateChange,
    NotificationType.RosterUpdate
};
}

```

Outcome of the call

`OnAnswerAppHostedMediaCompleted` is raised when the `AnswerAppHostedMedia` action completes. The `outcome` property in the `AnswerAppHostedMediaOutcomeEvent` indicates success or failure. If the call cannot be established, the bot should dispose the `AudioSocket` and `VideoSocket` objects it created for the call.

Receive audio media

If the `AudioSocket` was created with the ability to receive audio, then the `AudioMediaReceived` event will be invoked each time a frame of audio is received. The bot should expect to handle this event approximately 50 times per second, regardless of the peer that could be sourcing audio content (since comfort noise buffers are generated locally if no audio is received from the peer). Each packet of audio content is delivered in an `AudioMediaBuffer` object. This object contains a pointer to a native heap-allocated memory buffer containing the decoded audio content.

```
void OnAudioMediaReceived(
    object sender,
    AudioMediaReceivedEventArgs args)
{
    var buffer = args.Buffer;

    // native heap-allocated memory containing decoded content
    IntPtr rawData = buffer.Data;
}
```

The event handler must return quickly. It is recommended that the application queue the `AudioMediaBuffer` to be processed asynchronously. `OnAudioMediaReceived` events will be serialized by the Real-Time Media Platform (that is, the next event will not be raised until the current one returns). Once an `AudioMediaBuffer` has been consumed, the application must call the buffer's `Dispose` method so that the underlying unmanaged memory can be reclaimed by the media platform.

```
// release/dispose buffer when done
buffer.Dispose();
```

IMPORTANT

The bot must not call the buffer's `Dispose` method until it is finished accessing the buffer.

Receive video media

Receiving video is similar to receiving audio except that the number of buffers per second will depend on value of the frame rate. `VideoMediaBuffer` has `VideoFormat` and `OriginalVideoFormat` properties. `OriginalVideoFormat` represents the original format of the buffer when it was sourced. It is only available when receiving video buffers via the `IVideoSocket.VideoMediaReceived` event handler. If the buffer was resized before being transmitted, the `OriginalVideoFormat` property will have the original Width and Height, whereas `VideoFormat` will have the current ones after the resize. If the Width and Height properties of `OriginalVideoFormat` differ from the `VideoFormat` property, the consumer of the `VideoMediaBuffer` raised in the `VideoMediaReceived` event should resize the buffer to fit the `originalVideoFormat` size. Currently only NV12 format is supported for receive.

Send audio media

If the `AudioSocket` is configured to send media, the bot should register for the `AudioSendStatusChanged` event

handler on the `AudioSocket` to get notifications about sending status changes. The bot should start sending audio only after the send status changes to Active.

```
void AudioSocket_OnSendStatusChanged(
    object sender,
    AudioSendStatusChangedEventArgs args)
{
    switch (args.MediaSendStatus)
    {
        case MediaSendStatus.Active:
            // notify bot to begin sending audio
            break;

        case MediaSendStatus.Inactive:
            // notify bot to stop sending audio
            break;
    }
}
```

To send audio media, it is assumed the PCM audio content is contained within a native heap-allocated buffer. The bot must derive from the `AudioMediaBuffer` abstract class. Media is sent asynchronously by the `AudioSocket`'s `Send` method and the platform will call `Dispose` on the `AudioMediaBuffer` when the send completes. The bot should not release (or return to a pool allocator) the unmanaged resources when the `Send` returns. It must wait for `Dispose` to be called.

Send video media

Sending video media is similar to that of audio media. The bot should register for `VideoSendStatusChanged` event and wait for `MediaSendStatus` to be `Active`. The bot must not release or reclaim the buffer's unmanaged resources until the `Dispose` method is called. RGB24, NV12, and Yuy2 color formats are supported.

While multiple `VideoFormat`s are supported for sending video, the `videoFormat` that is currently preferred is communicated to the bot via the `VideoSendStatusChanged` event. The preferred `VideoFormat` for sending video may change due to network bandwidth conditions, or the peer client resizing its video window.

```
void VideoSocket_OnSendStatusChanged(
    object sender,
    VideoSendStatusChangedEventArgs args)
{
    VideoFormat preferredVideoFormat;

    switch (args.MediaSendStatus)
    {
        case MediaSendStatus.Active:
            // notify bot to begin sending audio
            // bot is recommended to use this format for sourcing video content.
            preferredVideoFormat = args.PreferredVideoSourceFormat;
            break;

        case MediaSendStatus.Inactive:
            // notify bot to stop sending audio
            break;
    }
}
```

Each `VideoMediaBuffer` has a `VideoFormat` property to indicate the format of the video content for that particular buffer. While the `VideoFormat` property does not have to match the `PreferredVideoSourceFormat` property indicated in the `VideoSendStatusChanged` event, it is strongly recommended to use the indicated `PreferredVideoSourceFormat` to avoid wasteful CPU cycles spent on video frame resizing.

Call notifications

Call state changes

The bot can get call state change notifications by subscribing to the `NotificationType.CallStateChange` in `NotificationSubscriptions` of `RealTimeMediaIncomingCallEvent.RealTimeMediaWorkflow`.

```
private Task OnCallStateChangeNotification(CallStateChangeNotification callStateChangeNotification)
{
    if (callStateChangeNotification.CurrentState == CallState.Terminated)
    {
        // stop sending media and dispose the media sockets
        _audioSocket.Dispose();
        _videoSocket.Dispose();
    }

    return Task.CompletedTask;
}
```

Roster update

If the bot is added to a conference, it can listen to the roster of the conference by subscribing to `NotificationType.RosterUpdate` in `NotificationSubscriptions` of `RealTimeMediaIncomingCallEvent.RealTimeMediaWorkflow`. The `RosterUpdateNotification` has the details of all participants in the conference. The bot could choose to wait for a notification with a participant sending video and then `Subscribe` to the participant's video on one of its `VideoSocket` objects.

```
private async Task OnRosterUpdateNotification(RosterUpdateNotification rosterUpdateNotification)
{
    // Find a video source in the conference to subscribe
    foreach (RosterParticipant participant in rosterUpdateNotification.Participants)
    {
        if (participant.MediaType == ModalityType.Video
            && (participant.MediaStreamDirection == "sendonly" || participant.MediaStreamDirection == "sendrecv"))
        {
            var videoSubscription = new VideoSubscription
            {
                ParticipantIdentity = participant.Identity,
                OperationId = Guid.NewGuid().ToString(),
                SocketId = _videoSocket.SocketId,
                VideoModality = ModalityType.Video,
                VideoResolution = ResolutionFormat.Hd720p
            };

            await CallService.Subscribe(videoSubscription).ConfigureAwait(false);
            break;
        }
    }
}
```

End the call

Handle call termination from the caller

When the user disconnects the call to the bot in a peer to peer conversation or when the bot is removed from the conference, the bot gets a call state change notification with the CallState as Terminated. The bot should dispose any media sockets it created.

Terminate the call from the bot

The bot can choose to end the call by calling `EndCall` on `IRealTimeMediaCallService`.

Handle call clean up by the Bot Framework

On error conditions (for example, if `AnswerAppHostedMediaOutcomeEvent` is not received within a reasonable time), the Bot Framework may terminate the call. The bot should register for `onCallCleanup` event and dispose the media sockets.

Deploy a real-time media bot from Visual Studio to Azure

8/7/2017 • 2 min to read • [Edit Online](#)

Real-time media bots can be hosted in either an "IaaS" Azure Virtual Machine or a "classic" Azure Cloud Service. This article shows how to deploy a bot, hosted in an Azure Cloud Service Worker Role, from Visual Studio using its built-in publish capability.

Prerequisites

You must have a Microsoft Azure subscription before you can deploy a bot to Azure. If you do not already have a subscription, you can register for a [free trial](#). Additionally, the process described by this article requires Visual Studio. If you do not already have Visual Studio, you can download [Visual Studio 2017 Community](#) for free.

Certificate from a valid certificate authority

The bot needs to be configured with a valid certificate from a trusted Certificate Authority (CA). The Subject Name (SN) or the last entry of the Subject Alternative Name (SAN) of the certificate should be the name of the cloud service. Wild-card certificates are currently not supported. If a CNAME is used to point to the cloud service, the CNAME should be the SN or the last SAN entry of the certificate.

Configure application settings

For your bot to function properly in the cloud, you must ensure that its application settings are correct. If you've already [registered](#) your bot with the Bot Framework, update the Microsoft App Id and Microsoft App Password values in your application's configuration settings as part of the deployment process. Specify the **app ID** and **password** values that were generated for your bot during registration.

TIP

Set the following key values in the app.config file of your worker role:

- MicrosoftAppId
- MicrosoftAppPassword

Create worker role in the Azure Portal

Step 1: Create Cloud Service(classic)

Log on to [Azure Portal](#). Click + on the left side of the screen and choose **Cloud services (classic)**. Provide the required information in the form and click **Create**.

The screenshot shows the Azure portal interface for creating a new Cloud service (classic). On the left, a sidebar lists various services like New, Dashboard, Resource groups, etc. The main area is titled 'Cloud services (classic)' and shows a list of subscriptions. A central panel displays a 'Create Cloud services (classic)' form. The form fields include:

- * DNS name: huebot
- * Subscription: [selected]
- * Resource group:
 - Create new (radio button selected)
 - Use existinghuebotResource
- * Location: East US
- Package (Optional) Select a package
- Certificates (Optional) Add certificates

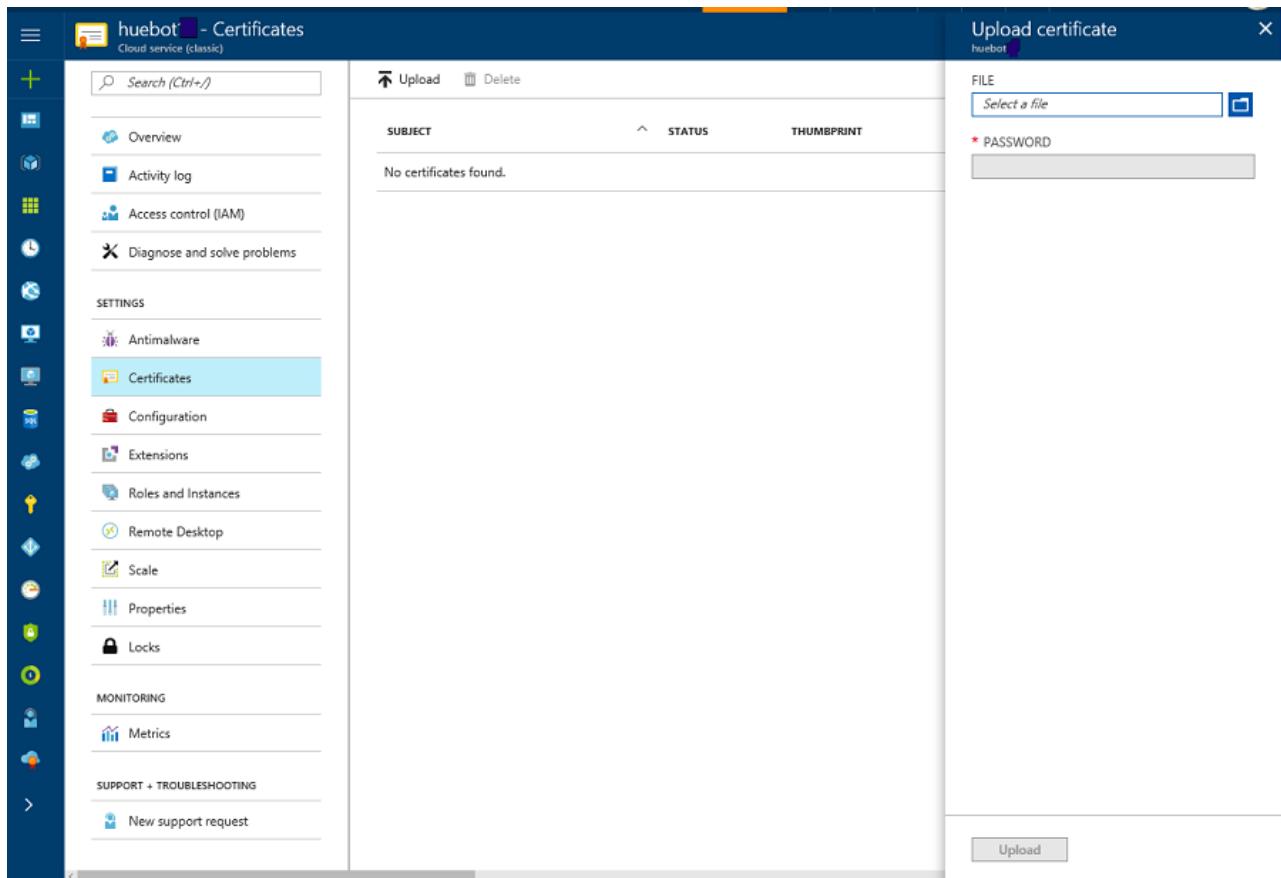
At the bottom of the form are 'Create' and 'Automation options' buttons.

NOTE

The DNS name of the bot should be supplied in the url for bot registration.

Step 2: Upload the certificate for the bot

Once the bot is created, upload the certificate for the bot.



Modify service configuration with worker role details

The Fully-Qualified Domain Name (FQDN) of the bot is not available through the Azure RoleEnvironment APIs. Therefore, the bot must be provided with its FQDN. It also needs to know about the certificate for HTTPS. These can be configured in the service configuration (.cscfg) file of the worker role.

TIP

If you are deploying a sample from BotBuilder-RealTimeMediaCalling git repository,

- Substitute the \$DnsName\$ with either the cloud service name or the CNAME if one is used in the service configuration.

```
<Setting name="ServiceDnsName" value="$DnsName$" />
```

- Substitute \$CertThumbprint\$ with the thumbprint of the certificate uploaded to the bot in the following lines from the configuration.

```
<Setting name="DefaultCertificate" value="$CertThumbprint$" />
<Certificate name="Default" thumbprint="$CertThumbprint$" thumbprintAlgorithm="sha1" />
```

Publish the bot from Visual Studio

Step 1: Launch the Microsoft Azure Publishing Wizard in Visual Studio

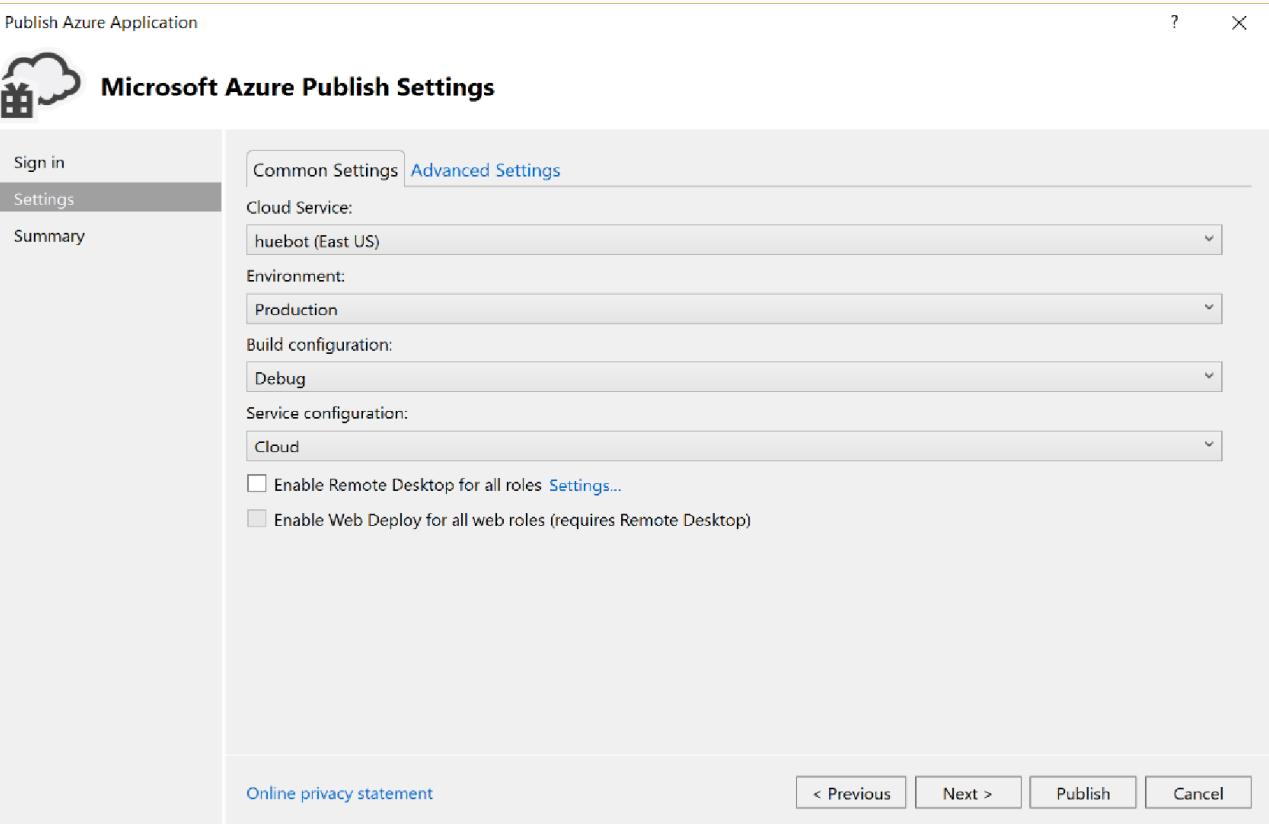
Open your project in Visual Studio. In Solution Explorer, right-click the cloud service project and select **Publish**. This starts the Microsoft Azure publishing wizard. Use your credentials to sign in to the appropriate subscription.



The screenshot shows the Microsoft Azure Publish Sign In dialog box. On the left is a sidebar with three tabs: "Sign in" (selected), "Settings", and "Summary". The main area contains a large blacked-out input field and a dropdown menu labeled "Choose your subscription:" with a single option also blacked out. At the bottom are buttons for "Online privacy statement", "< Previous" (disabled), "Next >" (highlighted in blue), "Publish" (disabled), and "Cancel".

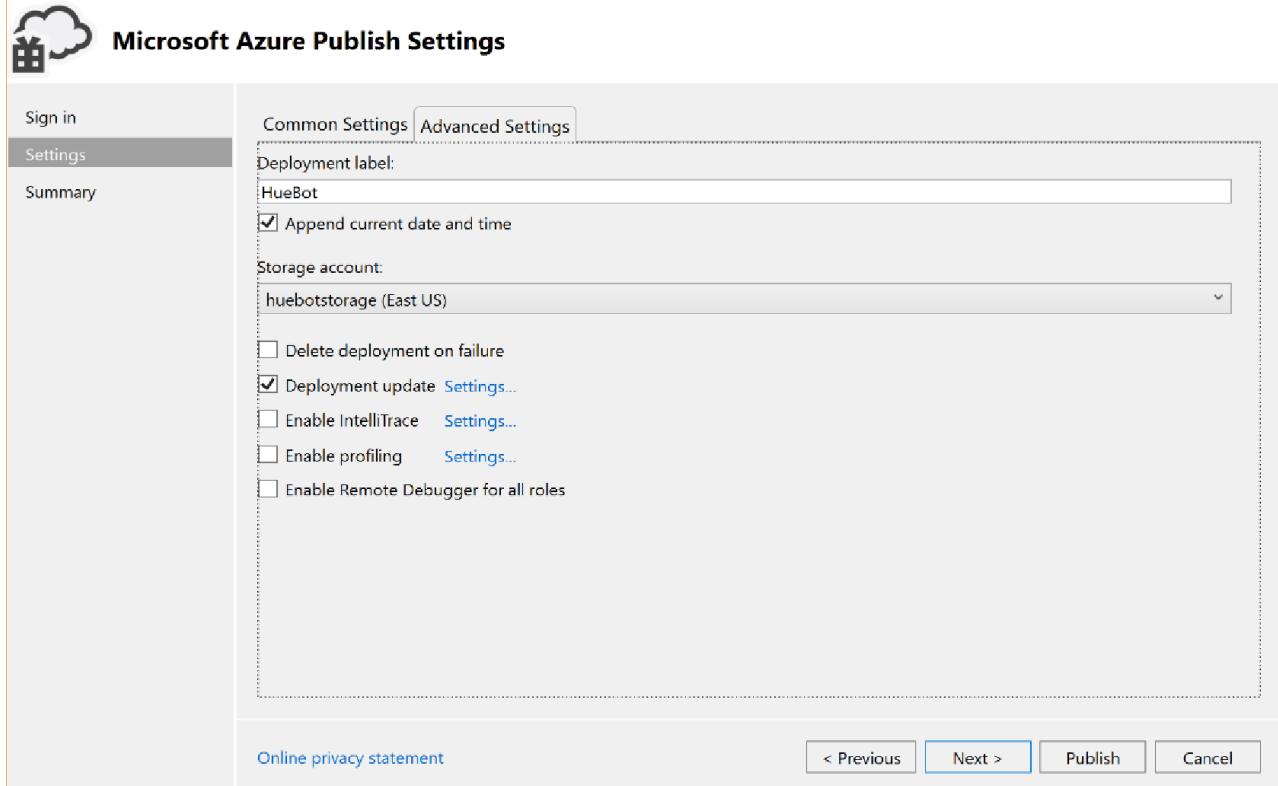
Step 2: Publish the bot

Click **Next**. This will open the **Settings** tab. Specify the Cloud Service, Environment, Build Configuration and the Service Configuration for deploying the bot.



The screenshot shows the Microsoft Azure Publish Settings dialog box. On the left is a sidebar with three tabs: "Sign in", "Settings" (selected), and "Summary". The main area has two tabs at the top: "Common Settings" (selected) and "Advanced Settings". Under "Common Settings", there are four dropdown menus: "Cloud Service" (set to "huebot (East US)"), "Environment" (set to "Production"), "Build configuration" (set to "Debug"), and "Service configuration" (set to "Cloud"). Below these are two checkboxes: "Enable Remote Desktop for all roles" (unchecked) and "Enable Web Deploy for all web roles (requires Remote Desktop)" (unchecked). At the bottom are buttons for "Online privacy statement", "< Previous" (disabled), "Next >" (disabled), "Publish" (disabled), and "Cancel".

You can optionally choose **Advanced Settings** and specify the Storage account for the deployment logs (which you can use to debug issues).



The screenshot shows the Microsoft Azure Publish Settings dialog box. On the left, there's a sidebar with 'Sign in', 'Settings' (which is selected), and 'Summary'. The main area has two tabs at the top: 'Common Settings' (selected) and 'Advanced Settings'. Under 'Common Settings', there are sections for 'Deployment label' (set to 'HueBot'), 'Append current date and time' (checkbox checked), 'Storage account' (set to 'huebotstorage (East US)'), and several deployment options: 'Delete deployment on failure' (checkbox unselected), 'Deployment update' (checkbox checked, with a 'Settings...' link), 'Enable IntelliTrace' (checkbox unselected, with a 'Settings...' link), 'Enable profiling' (checkbox unselected, with a 'Settings...' link), and 'Enable Remote Debugger for all roles' (checkbox unselected). At the bottom, there's an 'Online privacy statement' link, and buttons for '< Previous', 'Next >', 'Publish' (highlighted in blue), and 'Cancel'.

Verify the configuration in the **Summary** tab and click **Publish** to deploy the bot to Microsoft Azure.

Manage state data

11/2/2017 • 5 min to read • [Edit Online](#)

The Bot Framework State service enables your bot to store and retrieve state data that is associated with a user, a conversation, or a specific user within the context of a specific conversation. State data can be used for many purposes, such as determining where the prior conversation left off or simply greeting a returning user by name. If you store a user's preferences, you can use that information to customize the conversation the next time you chat. For example, you might alert the user to a news article about a topic that interests her, or alert a user when an appointment becomes available.

If your bot uses [dialogs](#), conversation state (the dialog stack and the state of each dialog in the stack) is automatically stored using the Bot Framework State service.

Bot state methods

This table lists the methods within Bot state service that you can use to manage state data.

METHOD	SCOPED TO	OBJECTIVE
<code>GetUserData</code>	User	Get state data that has previously been saved for the user on the specified channel
<code>GetConversationData</code>	Conversation	Get state data that has previously been saved for the conversation on the specified channel
<code>GetPrivateConversationData</code>	User and Conversation	Get state data that has previously been saved for the user within the conversation on the specified channel
<code>SetUserData</code>	User	Save state data for the user on the specified channel
<code>SetConversationData</code>	Conversation	Save state data for the conversation on the specified channel. Note: Because the <code>DeleteStateForUser</code> method does not delete data that has been stored using the <code>SetConversationData</code> method, you must NOT use this method to store a user's personally identifiable information (PII).
<code>SetPrivateConversationData</code>	User and Conversation	Save state data for the user within the conversation on the specified channel

METHOD	SCOPED TO	OBJECTIVE
<code>DeleteStateForUser</code>	User	<p>Delete state data for the user that has previously been stored by using either the <code>SetUserData</code> method or the <code>SetPrivateConversationData</code> method.</p> <p>Note: Your bot should call this method when it receives an activity of type <code>deleteUserData</code> or an activity of type <code>contactRelationUpdate</code> that indicates the bot has been removed from the user's contact list.</p>

If your bot saves state data by using one of the "Set...Data" methods, future messages that your bot receives in the same context will contain that data, which your bot can access by using the corresponding "Get...Data" method.

Useful properties for managing state data

Each `Activity` object contains properties that you will use to manage state data.

PROPERTY	DESCRIPTION	USE CASE
<code>From</code>	Uniquely identifies a user on a channel	Storing and retrieving state data that is associated with a user
<code>Conversation</code>	Uniquely identifies a conversation	Storing and retrieving state data that is associated with a conversation
<code>From</code> and <code>Conversation</code>	Uniquely identifies a user and conversation	Storing and retrieving state data that is associated with a specific user within the context of a specific conversation

NOTE

You may use these property values as keys even if you opt to store state data in your own database, rather than using the Bot Framework state data store.

Create a state client

The `StateClient` object enables you to manage state data using the Bot Builder SDK for .NET. If you have access to a message that belongs to the same context in which you want to manage state data, you can create a state client by calling the `GetStateClient` method on the `Activity` object.

```
StateClient stateClient = activity.GetStateClient();
```

If you do not have access to a message that belongs to the same context in which you want to manage state data, you can create a state client by simply creating a new instance of the `StateClient` class. In this example, `microsoftAppId` and `microsoftAppPassword` are the Bot Framework authentication credentials that you acquire for your bot during the [bot registration](#) process.

```
StateClient stateClient = new StateClient(new MicrosoftAppCredentials(microsoftAppId, microsoftAppPassword));
```

NOTE

The default state client is stored in a central service. For some channels, you may want to use a state API that is hosted within the channel itself, so that state data can be stored in a compliant store that the channel supplies.

Get state data

Each of the "**Get...Data**" methods returns a `BotData` object that contains the state data for the specified user and/or conversation. To get a specific property value from a `BotData` object, call the `GetProperty` method.

The following code example shows how to get a typed property from user data.

```
BotData userData = await stateClient.BotState.GetUserDataAsync(activity.ChannelId, activity.From.Id);
var sentGreeting = userData.GetProperty<bool>("SentGreeting");
```

The following code example shows how to get a property from a complex type within user data.

```
MyCustomType myUserData = new MyCustomType();
BotData botData = await botState.GetUserDataAsync(activity.ChannelId, activity.From.Id);
myUserData = botData.GetProperty<MyCustomType>("UserData");
```

If no state data exists for the user and/or conversation that is specified for a "**Get...Data**" method call, the `BotData` object that is returned will contain these property values:

- `BotData.Data` = null
- `BotData.ETag` = "*"

Save state data

To save state data, first get the `BotData` object by calling the appropriate "**Get...Data**" method, then update it by calling the `SetProperty` method for each property you want to update, and save it by calling the appropriate "**Set...Data**" method.

NOTE

You may store up to 32 kilobytes of data for each user on a channel, each conversation on a channel, and each user within the context of a conversation on a channel.

The following code example shows how to save a typed property in user data.

```
BotData userData = await stateClient.BotState.GetUserDataAsync(activity.ChannelId, activity.From.Id);
userData SetProperty<bool>("SentGreeting", true);
await stateClient.BotState.SetUserDataAsync(activity.ChannelId, activity.From.Id, userData);
```

The following code example shows how to save a property in a complex type within user data.

```
BotData userData = await stateClient.BotState.GetUserDataAsync(activity.ChannelId, activity.From.Id);
userData SetProperty<MyCustomType>("UserData", myUserData);
await stateClient.BotState.SetUserDataAsync(activity.ChannelId, activity.From.Id, userData);
```

Handle concurrency issues

Your bot may receive an error response with HTTP status code **412 Precondition Failed** when it attempts to save state data, if another instance of the bot has changed the data. You can design your bot to account for this scenario, as shown in the following code example.

```
try
{
    // get user data
    BotData userData = await stateClient.BotState.GetUserDataAdapter(activity.ChannelId, activity.From.Id);

    // modify a property within user data
    userData SetProperty<bool>("SentGreeting", true);

    // save updated user data
    await stateClient.BotState.SetUserDataAsync(activity.ChannelId, activity.From.Id, userData);
}
catch (HttpOperationException err)
{
    // handle error with HTTP status code 412 Precondition Failed
}
```

Manage data storage

Under the hood, the Bot Builder SDK for .NET stores state data using the Bot Connector State service, which is intended for prototyping only and is not designed for use by bots in a production environment. For performance and security reasons in the production environment, consider implementing one of the following data storage options:

1. [Manage state data with Cosmos DB](#)
2. [Manage state data with Table storage](#)

With either of these [Azure Extensions](#) options, the mechanism for setting and persisting data via the Bot Framework SDK for .NET remains the same as described previously in this article.

Additional resources

- [Bot Framework troubleshooting guide](#)
- [Bot Builder SDK for .NET Reference](#)

Manage custom state data with Azure Cosmos DB for .NET

9/22/2017 • 3 min to read • [Edit Online](#)

In this article, you'll implement Azure Cosmos DB to store and manage your bot's state data. The default Connector State Service used by bots is not intended for the production environment. You should either use [Azure Extensions](#) available on GitHub or implement a custom state client using data storage platform of your choice. Here are some of the reasons to use custom state storage:

- Higher state API throughput (more control over performance)
- Lower-latency for geo-distribution
- Control over where the data is stored
- Access to the actual state data
- Store more than 32kb of data

Prerequisites

You'll need:

- [Microsoft Azure Account](#)
- [Visual Studio 2015 or later](#)
- [Bot Builder Azure NuGet Package](#)
- [Autofac Web Api2 NuGet Package](#)
- [Bot Framework Emulator](#)

Create Azure account

If you don't have an Azure account, click [here](#) to sign up for a free trial.

Set up the Azure Cosmos DB database

1. After you've logged into the Azure portal, create a new *Azure Cosmos DB* database by clicking **New**.
2. Click **Databases**.
3. Find **Azure Cosmos DB** and click **Create**.
4. Fill in the fields. For the **API** field, select **SQL (DocumentDB)**. When done filling in all the fields, click the **Create** button at the bottom of the screen to deploy the new database.
5. After the new database is deployed, navigate to your new database. Click **Access keys** to find keys and connection strings. Your bot will use this information to call the storage service to save state data.

Install NuGet packages

1. Open an existing C# bot project, or create a new one using the Bot template in Visual Studio.
2. Install the following NuGet packages:
 - Microsoft.Bot.Builder.Azure
 - Autofac.WebApi2

Add connection string

Add the following entries into the Web.config file:

```
<add key="DocumentDbUrl" value="Your DocumentDB URI"/>
<add key="DocumentDbKey" value="Your DocumentDB Key"/>
```

You'll replace the value with your URI and Primary Key found in your Azure Cosmos DB. Save the Web.config file.

Modify your bot code

To use **Azure Cosmos DB** storage, add the following lines of code to your bot's **Global.asax.cs** file.

```
using System;
using Autofac;
using System.Web.Http;
using System.Configuration;
using Microsoft.Bot.Connector;
using Microsoft.Bot.Builder.Azure;
using Microsoft.Bot.Builder.Dialogs;
using Microsoft.Bot.Builder.Dialogs.Internals;

var uri = new Uri(ConfigurationManager.AppSettings["DocumentDbUrl"]);
var key = ConfigurationManager.AppSettings["DocumentDbKey"];
var store = new DocumentDbBotDataStore(uri, key);

Conversation.UpdateContainer(
    builder =>
{
    builder.Register(c => store)
        .Keyed<IBotDataStore<BotData>>(AzureModule.Key_DataStore)
        .AsSelf()
        .SingleInstance();

    builder.Register(c => new CachingBotDataStore(store,
CachingBotDataStoreConsistencyPolicy.ETagBasedConsistency))
        .As<IBotDataStore<BotData>>()
        .AsSelf()
        .InstancePerLifetimeScope();
});
```

Save the global.asax.cs file. Now you are ready to test the bot with the emulator.

Run your bot app

Run your bot in Visual Studio, the code you added will create the custom **botdata** table in Azure.

Connect your bot to the emulator

At this point, your bot is running locally. Next, start the emulator and then connect to your bot in the emulator:

1. Type `http://localhost:port-number/api/messages` into the address bar, where port-number matches the port number shown in the browser where your application is running. You can leave **Microsoft App ID** and **Microsoft App Password** fields blank for now. You'll get this information later when you register your bot.
2. Click **Connect**.
3. Test your bot by typing a few messages in the emulator.

View state data on Azure Portal

To view the state data, sign into your Azure portal and navigate to your database. Click **Data Explorer (preview)**

to verify that the state information from your bot is being saved.

Next steps

In this article, you used Cosmos DB for saving and managing your bot's data. Next, learn how to model conversation flow by using dialogs.

[Manage conversation flow](#)

Additional resources

If you are unfamiliar with Inversion of Control containers and Dependency Injection pattern used in the code above, visit [Autofac](#) site for information.

You can also download a [sample](#) from GitHub to learn more about using Cosmos DB for managing state.

Manage custom state data with Azure Table Storage for .NET

9/22/2017 • 3 min to read • [Edit Online](#)

In this article, you'll implement Azure Table storage to store and manage your bot's state data. The default Connector State Service used by bots is not intended for the production environment. You should either use [Azure Extensions](#) available on GitHub or implement a custom state client using data storage platform of your choice. Here are some of the reasons to use custom state storage:

- Higher state API throughput (more control over performance)
- Lower-latency for geo-distribution
- Control over where the data is stored
- Access to the actual state data
- Store more than 32kb of data

Prerequisites

You'll need:

- [Microsoft Azure Account](#)
- [Visual Studio 2015 or later](#)
- [Bot Builder Azure NuGet Package](#)
- [Autofac Web Api2 NuGet Package](#)
- [Bot Framework Emulator](#)
- [Azure Storage Explorer](#)

Create Azure account

If you don't have an Azure account, click [here](#) to sign up for a free trial.

Set up the Azure Table storage service

1. After you've logged into the Azure portal, create a new Azure Table storage service by clicking on **New**.
2. Search for **Storage account** that implements the Azure Table.
3. Fill in the fields, click the **Create** button at the bottom of the screen to deploy the new storage service. After the new storage service is deployed, it will display features and options available to you.
4. Select the **Access keys** tab on the left, and copy the connection string for later use. Your bot will use this connection string to call the storage service to save state data.

Install NuGet packages

1. Open an existing C# bot project, or create a new one using the C# Bot template in Visual Studio.
2. Install the following NuGet packages:
 - Microsoft.Bot.Builder.Azure
 - Autofac.WebApi2

Add connection string

Add the following entry in your Web.config file:

```
<connectionStrings>
<add name="StorageConnectionString"
      connectionString="YourConnectionString"/>
</connectionStrings>
```

Replace "YourConnectionString" with the connection string to the Azure Table storage you saved earlier. Save the Web.config file.

Modify your bot code

In the Global.asax.cs file, add the following `using` statements:

```
using Autofac;
using System.Configuration;
using Microsoft.Bot.Connector;
using Microsoft.Bot.Builder.Azure;
using Microsoft.Bot.Builder.Dialogs;
using Microsoft.Bot.Builder.Dialogs.Internals;
```

Create an instance of the `TableBotDataStore` class. The `TableBotDataStore` class implements the `IBotDataStore<BotData>` interface. The `IBotDataStore` interface allows you to override the default Connector State Service connection.

```
var store = new
TableBotDataStore(ConfigurationManager.ConnectionStrings["StorageConnectionString"].ConnectionString);
```

Register the service as shown below:

```
Conversation.UpdateContainer(
    builder =>
{
    builder.Register(c => store)
        .Keyed<IBotDataStore<BotData>>(AzureModule.Key_DataStore)
        .AsSelf()
        .SingleInstance();

    builder.Register(c => new CachingBotDataStore(store,
        CachingBotDataStoreConsistencyPolicy
        .ETagBasedConsistency))
        .As<IBotDataStore<BotData>>()
        .AsSelf()
        .InstancePerLifetimeScope();

});
```

Save the global.asax.cs file.

Run your bot app

Run your bot in Visual Studio, the code you added will create the custom **botdata** table in Azure.

Connect your bot to the emulator

At this point, your bot is running locally. Next, start the emulator and then connect to your bot in the emulator:

1. Type `http://localhost:port-number/api/messages` into the address bar, where port-number matches the port number shown in the browser where your application is running. You can leave **Microsoft App ID** and **Microsoft App Password** fields blank for now. You'll get this information later when you register your bot.
2. Click **Connect**.
3. Test your bot by typing a few messages in the emulator.

View data in Azure Table storage

To view the state data, open **Storage Explorer** and connect to Azure using your Azure Portal credential or connect directly to the table using the storage name and storage key then navigate to your table name.

Next steps

In this article, you implemented Azure Table storage for saving and managing your bot's data. Next, learn how to model conversation flow by using dialogs.

[Manage conversation flow](#)

Additional resources

If you are unfamiliar with Inversion of Control containers and Dependency Injection pattern used in the code above, go to [Autofac](#) site for information.

You can also download a complete [Azure Table storage](#) sample from GitHub.

Recognize intents and entities with LUIS using a prebuilt domain

11/2/2017 • 11 min to read • [Edit Online](#)

This article uses the example of a bot for taking notes, to demonstrate how Language Understanding Intelligent Service ([LUIS](#)) helps your bot respond appropriately to natural language input. A bot detects what a user wants to do by identifying their **intent**. This intent is determined from spoken or textual input, or **utterances**. The intent maps utterances to actions that the bot takes. For example, a note-taking bot recognizes a `Notes.Create` intent to invoke the functionality for creating a note. A bot may also need to extract **entities**, which are important words in utterances. In the example of a note-taking bot, the `Notes.Title` entity identifies the title of each note.

Create your LUIS app

The LUIS app, which is the web service you configure at [www.luis.ai](#) to provide the intents and entities to the bot. In this example, the LUIS app makes use of the **Notes** prebuilt domain, which is a collection of ready-to-use intents and entities. To create the LUIS app, follow these steps:

1. Log in to [www.luis.ai](#) using your Cognitive Services API account. If you don't have an account, you can create a free account in the [Azure portal](#).
2. In the **My Apps** page, click **New App**, enter a name like Notes in the **Name** field, and choose **Bootstrap Key** in the **Key to use** field.
3. In the **Intents** page, click **Add prebuilt domain intents** and select **Notes.Create**, **Notes.Delete** and **Notes.ReadAloud**.
4. In the **Intents** page, click on the **None** intent. This intent is meant for utterances that don't correspond to any other intents. Enter an example of an utterance unrelated to notes, like "Turn off the lights."
5. In the **Entities** page, click **Add prebuilt domain entities** and select **Notes.Title**.
6. In the **Train & Test** page, train your app.
7. In the **Publish** page, click **Publish**. After successful publish, copy the **Endpoint URL** from the **Publish App** page, to use later in your bot's code. The URL has a format similar to this example:

```
https://westus.api.cognitive.microsoft.com/luis/v2.0/apps/3889f7d0-9501-45c8-be5f-8635975eea8b?subscription-key=67073e45132a459db515ca04cea325d3&timezoneOffset=0&verbose=true&q=
```

TIP

You can also create the LUIS app by importing the **Notes** sample JSON. In [www.luis.ai](#), click **Import App** in the **My Apps** page and select the JSON file to import.

Create a note-taking bot integrated with the LUIS app

To create a bot that uses the LUIS app, you can first start with the sample bot that you create according to the steps in [Create a bot with the Bot Builder SDK for .NET](#), and edit the code to correspond to the examples in this article.

TIP

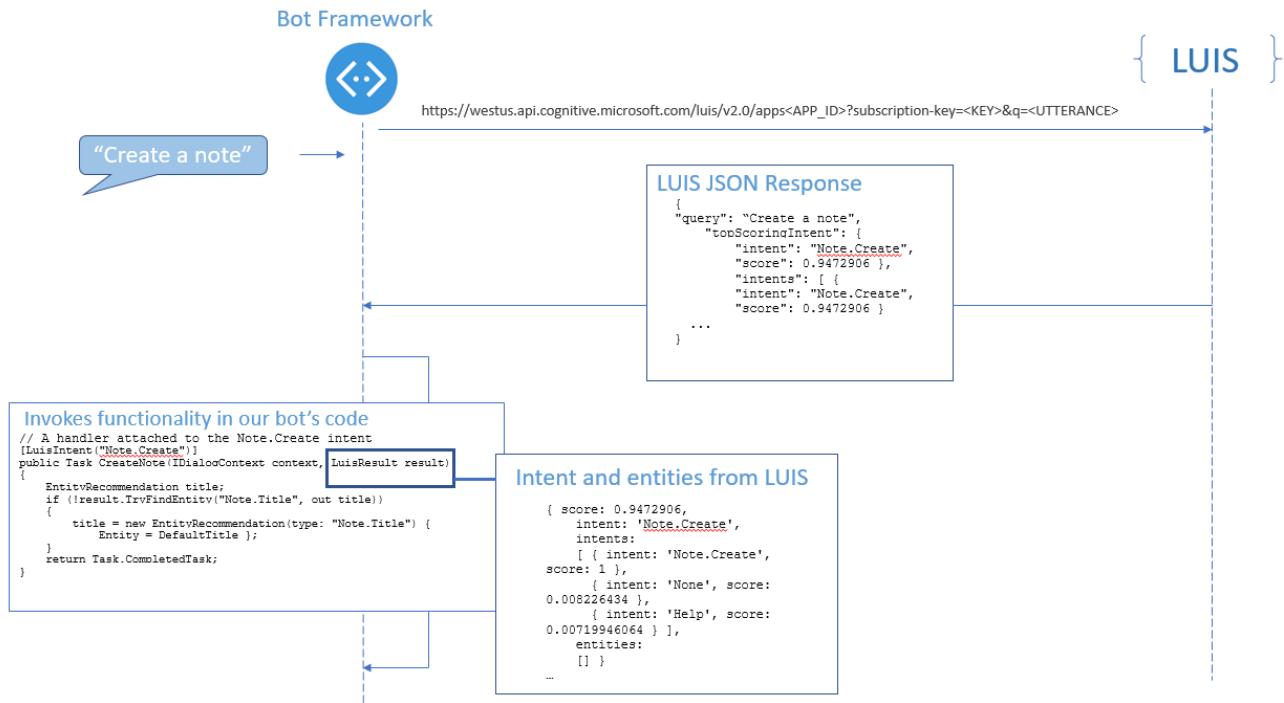
You can also find the sample code described in this article at [Notes bot sample](#).

How LUIS passes intents and entities to your bot

The following diagram shows the sequence of events that happen after the bot receives an utterance from the user. First, the bot passes the utterance to the LUIS app and gets a JSON result from LUIS that contains intents and entities. Next, your bot automatically invokes any matching handler in a [LuisDialog](#). The intent handler is associated with the high-scoring intent in

the LUIS result by using the [LuisIntent attribute](#). The full details of the match, including the list of intents and entities that LUIS detected, are passed as a [LuisResult](#) to the `result` parameter of the matching handler.

```
<p align=center>
```



Create a class that derives from LuisDialog

To create a [dialog](#) that uses LUIS, first create a class that derives from [LuisDialog](#) and specify the [LuisModel attribute](#). To populate the `modelID` and `subscriptionKey` parameters for the `LuisModel` attribute, use the `id` and `subscription-key` attribute values from your LUIS app's endpoint URL.

The `domain` parameter is determined by the Azure region to which your LUIS app is published. If not supplied, it defaults to `westus.api.cognitive.microsoft.com`. See [Regions and keys](#) for more information.

```
[LuisModel("<YOUR_LUIS_APP_ID>", "YOUR_SUBSCRIPTION_KEY", domain: "westus.api.cognitive.microsoft.com")]
[Serializable]
public class SimpleNoteDialog : LuisDialog<object>
{
    // ...
}
```

Create methods to handle intents

Within the class, create the methods that execute when your LUIS model matches a user's utterance to intent. To designate the method that runs when a specific intent is matched, specify the [LuisIntent attribute](#).

This code example defines the method that executes when the `Note.Delete` intent is matched.

```
[LuisIntent("Note.Delete")]
public async Task DeleteNote(IDialogContext context, LuisResult result)
{
    Note note;
    if (TryFindNote(result, out note))
    {
        this.noteByTitle.Remove(note.Title);
        await context.PostAsync($"Note {note.Title} deleted");
    }
    else
    {
        // Prompt the user for a note title
        PromptDialog.Text(context, After_DeleteTitlePrompt, "What is the title of the note you want to delete?");
    }
}
```

In this example, if the LUIS app detects the title of the note the user wants to delete, and finds it in the list of notes, it removes it from the list of notes. Otherwise it prompts the user for the name of the note to delete.

NOTE

The LUIS app you created is meant to be a starting point for training, and might not be able to detect the title of notes at first. A LUIS app learns from example, so teach it to better recognize the title entity by giving it more example utterances to learn from. You can retrain your LUIS app without any modification to your bot's code. See [Add example utterances](#) and [train and test your LUIS app](#).

The `TryFindNote` method that `DeleteNote` calls inspects the `result` parameter from LUIS to see if the LUIS app detected a title entity. It then searches for a note with that title.

```
public bool TryFindNote(LuisResult result, out Note note)
{
    note = null;

    string titleToFind;

    EntityRecommendation title;
    if (result.TryFindEntity(Entity_Note_Title, out title))
    {
        return this.noteByTitle.TryGetValue(title.Entity, out note); // TryGetValue returns false if no match is found.
    }
    else
    {
        return false;
    }
}
```

Notes dialog implementation

This code example shows the full dialog implementation for the Notes bot, which also includes methods for handling the Note.Create and Note.ReadAloud intents, as well as a default intent handler which can handle the None intent, or any other intents the LUIS app may detect.

```
using System;
using System.Threading.Tasks;
using Microsoft.Bot.Builder.Dialogs;
using Microsoft.Bot.Builder.Luis;
using System.Collections.Generic;
using System.Linq;
using Microsoft.Bot.Builder.Luis.Models;

namespace NotesBot.Dialogs
{

    [LuisModel("<YOUR_LUIS_APP_ID>", "YOUR_SUBSCRIPTION_KEY", domain: "westus.api.cognitive.microsoft.com")]
    [Serializable]
    public class SimpleNoteDialog : LuisDialog<object>
```

```

private readonly Dictionary<string, Note> noteByTitle = new Dictionary<string, Note>();

// Store notes in a dictionary that uses the title as a key
private readonly Dictionary<string, Note> noteByTitle = new Dictionary<string, Note>();

// Default note title
public const string DefaultNoteTitle = "default";

// Name of note title entity
public const string Entity_Note_Title = "Note.Title";

/// <summary>
/// This method overload inspects the result from LUIS to see if a title entity was detected, and finds the
note with that title, or the note with the default title, if no title entity was found.
/// </summary>
/// <param name="result">The result from LUIS that contains intents and entities that LUIS recognized.</param>
/// <param name="note">This parameter returns any note that is found in the list of notes that has a matching
title.</param>
/// <returns>true if a note was found, otherwise false</returns>
public bool TryFindNote(LuisResult result, out Note note)
{
    note = null;

    string titleToFind;

    EntityRecommendation title;
    if (result.TryFindEntity(Entity_Note_Title, out title))
    {
        titleToFind = title.Entity;
    }
    else
    {
        titleToFind = DefaultNoteTitle;
    }

    return this.noteByTitle.TryGetValue(titleToFind, out note); // TryGetValue returns false if no match is
found.
}

/// <summary>
/// This method overload takes a string and finds the note with that title.
/// </summary>
/// <param name="noteTitle">A string containing the title of the note to search for.</param>
/// <param name="note">This parameter returns any note that is found in the list of notes that has a matching
title.</param>
/// <returns>true if a note was found, otherwise false</returns>
public bool TryFindNote(string noteTitle, out Note note)
{
    bool foundNote = this.noteByTitle.TryGetValue(noteTitle, out note); // TryGetValue returns false if no
match is found.
    return foundNote;
}

/// <summary>
/// Send a generic help message if an intent without an intent handler is detected.
/// </summary>
/// <param name="context">Dialog context.</param>
/// <param name="result">The result from LUIS.</param>
[LuisIntent("")]
public async Task None(IDialogContext context, LuisResult result)
{

    string message = $"I'm the Notes bot. I can understand requests to create, delete, and read notes. \n\n
Detected intent: " + string.Join(", ", result.Intents.Select(i => i.Intent));
    await context.PostAsync(message);
    context.Wait(MessageReceived);
}

/// <summary>
/// Handle the Note.Delete intent. If a title isn't detected in the LUIS result, prompt the user for a title.
/// </summary>
/// <param name="context">Dialog context.</param>
/// <param name="result">The result from LUIS.</param>
[LuisIntent("Note.Delete")]

```

```

public async Task DeleteNote(IDialogContext context, LuisResult result)
{
    Note note;
    if (TryFindNote(result, out note))
    {
        this.noteByTitle.Remove(note.Title);
        await context.PostAsync($"Note {note.Title} deleted");
    }
    else
    {
        // Prompt the user for a note title
        PromptDialog.Text(context, After_DeleteTitlePrompt, "What is the title of the note you want to
delete?");
    }
}

// Try to delete note that the user specified in response to the prompt.
private async Task After_DeleteTitlePrompt(IDialogContext context, IAwaitable<string> result)
{
    Note note;
    string titleToDelete = await result;
    bool foundNote = this.noteByTitle.TryGetValue(titleToDelete, out note);

    if (foundNote)
    {
        this.noteByTitle.Remove(note.Title);
        await context.PostAsync($"Note {note.Title} deleted");
    }
    else
    {
        await context.PostAsync($"Did not find note named {titleToDelete}.");
    }

    context.Wait(MessageReceived);
}

/// <summary>
/// Handles the Note.ReadAloud intent by displaying a note or notes.
/// If a title of an existing note is found in the LuisResult, that note is displayed.
/// If no title is detected in the LuisResult, all of the notes are displayed.
/// </summary>
/// <param name="context">Dialog context.</param>
/// <param name="result">LUIS result.</param>
[LuisIntent("Note.ReadAloud")]
public async Task ReadNote(IDialogContext context, LuisResult result)
{
    Note note;
    if (TryFindNote(result, out note))
    {
        await context.PostAsync($"**{note.Title}**: {note.Text}.");
    }
    else
    {
        // Print out all the notes if no specific note name was detected
        string NoteList = "Here's the list of all notes: \n\n";
        foreach (KeyValuePair<string, Note> entry in noteByTitle)
        {
            Note noteInList = entry.Value;
            NoteList += $"**{noteInList.Title}**: {noteInList.Text}.\n\n";
        }
        await context.PostAsync(NoteList);
    }

    context.Wait(MessageReceived);
}

private Note noteToCreate;
private string currentTitle;

/// <summary>
/// Handles the Note.Create intent. Prompts the user for the note title if the title isn't detected in the
LuisResult.
/// </summary>

```

```

/// <param name="context">Dialog context.</param>
/// <param name="result">LUIS result.</param>
[LuisIntent("Note.Create")]
public Task CreateNote(IDialogContext context, LuisResult result)
{
    EntityRecommendation title;
    if (!result.TryFindEntity(Entity_Note_Title, out title))
    {
        // Prompt the user for a note title
        PromptDialog.Text(context, After_TitlePrompt, "What is the title of the note you want to create?");
    }
    else
    {
        var note = new Note() { Title = title.Entity };
        noteToCreate = this.noteByTitle[note.Title] = note;

        // Prompt the user for what they want to say in the note
        PromptDialog.Text(context, After_TextPrompt, "What do you want to say in your note?");
    }

    return Task.CompletedTask;
}

// Creates a new note using the user's response to the prompt for a title
private async Task After_TitlePrompt(IDialogContext context, IAwaitable<string> result)
{
    EntityRecommendation title;

    // Set the title to the user's response
    currentTitle = await result;
    if (currentTitle != null)
    {
        title = new EntityRecommendation(type: Entity_Note_Title) { Entity = currentTitle };
    }
    else
    {
        // Use the default note title
        title = new EntityRecommendation(type: Entity_Note_Title) { Entity = DefaultNoteTitle };
    }

    // Create a new note object
    var note = new Note() { Title = title.Entity };
    // Add the new note to the list of notes and also save it in order to add text to it later
    noteToCreate = this.noteByTitle[note.Title] = note;

    // Prompt the user for what they want to say in the note
    PromptDialog.Text(context, After_TextPrompt, "What do you want to say in your note?");
}

// Sets the text of a newly created note using the user's response to the prompt for the text of the note.
private async Task After_TextPrompt(IDialogContext context, IAwaitable<string> result)
{
    // Set the text of the note
    noteToCreate.Text = await result;

    await context.PostAsync($"Created note **{this.noteToCreate.Title}** that says \""
    $"{this.noteToCreate.Text}\\");

    context.Wait(MessageReceived);
}

public SimpleNoteDialog()
{
}

public SimpleNoteDialog(ILuisService service)
    : base(service)
{
}

[Serializable]

```

```

public sealed class Note : IEquatable<Note>
{
    public string Title { get; set; }
    public string Text { get; set; }

    public override string ToString()
    {
        return $"[{this.Title} : {this.Text}]";
    }

    public bool Equals(Note other)
    {
        return other != null
            && this.Text == other.Text
            && this.Title == other.Title;
    }

    public override bool Equals(object other)
    {
        return Equals(other as Note);
    }

    public override int GetHashCode()
    {
        return this.Title.GetHashCode();
    }
}
}

```

Update the Post method

To use the SimpleNoteDialog that you implemented, update the `Post` method in your `MessageController` class to reference it.

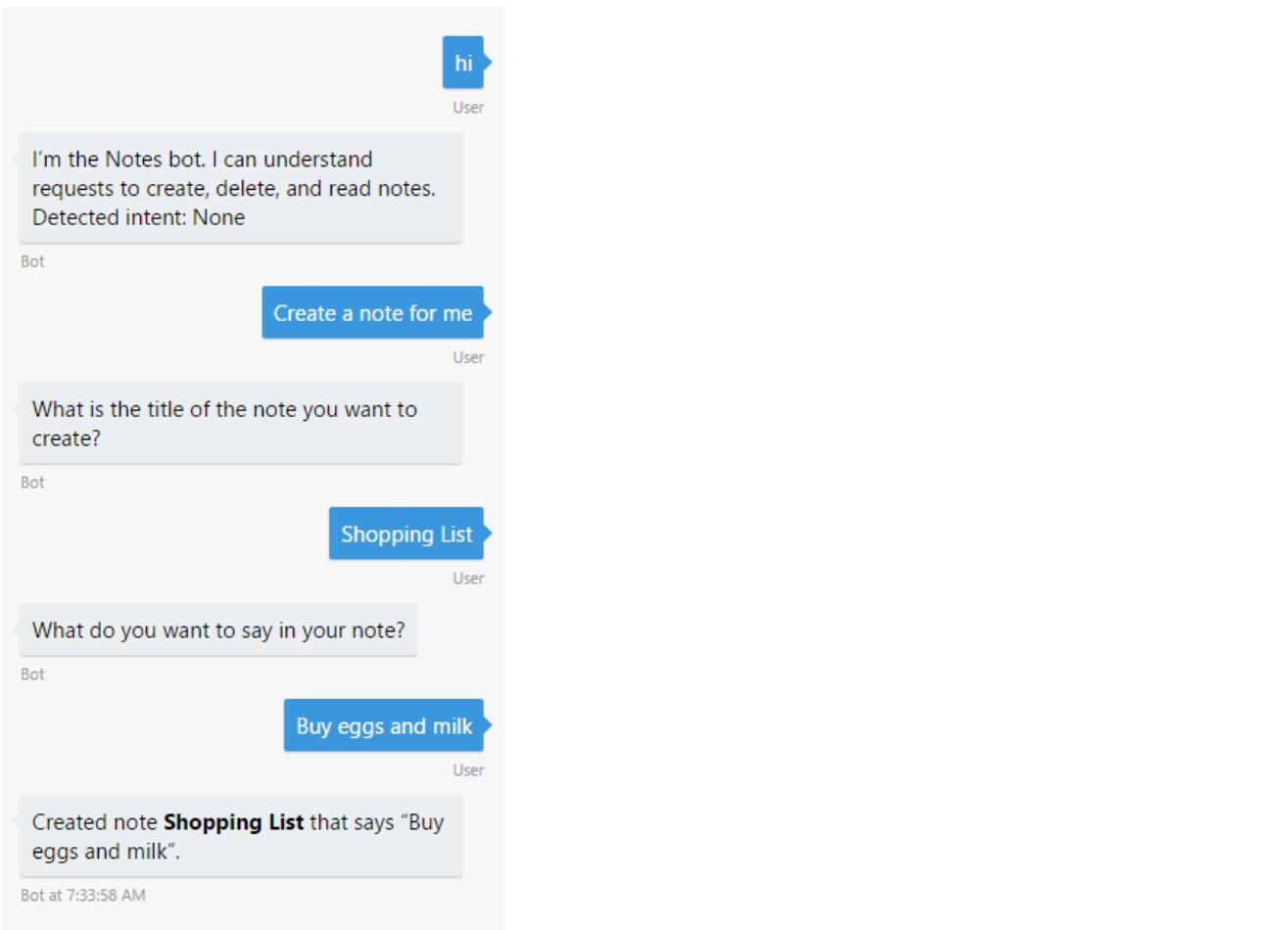
```

public async Task<HttpResponseMessage> Post([FromBody]Activity activity)
{
    if (activity.Type == ActivityTypes.Message)
    {
        await Conversation.SendAsync(activity, () => new Dialogs.SimpleNoteDialog());
    }
    else
    {
        HandleSystemMessage(activity);
    }
    var response = Request.CreateResponse(HttpStatusCode.OK);
    return response;
}

```

Try the bot

You can run the bot using the Bot Framework Emulator and tell it to create a note. <p align=center>



TIP

If you find that your bot doesn't always recognize the correct intent or entities, improve your LUIS app's performance by giving it more example utterances to train it. You can retrain your LUIS app without any modification to your bot's code. See [Add example utterances](#) and [train and test your LUIS app](#).

Regions and keys

The region to which you publish your LUIS app must correspond to the region or location you specify in the Azure portal when you create a key. To publish a LUIS app to more than one region, you need at least one key per region. LUIS apps created at <https://www.luis.ai> can be published to endpoints in the following regions:

AZURE REGION	ENDPOINT URL FORMAT
West US	<code>https://westus.api.cognitive.microsoft.com/luis/v2.0/apps/YOUR-APP-ID?subscription-key=YOUR-SUBSCRIPTION-KEY</code>
East US 2	<code>https://eastus2.api.cognitive.microsoft.com/luis/v2.0/apps/YOUR-APP-ID?subscription-key=YOUR-SUBSCRIPTION-KEY</code>
West Central US	<code>https://westcentralus.api.cognitive.microsoft.com/luis/v2.0/apps/YOUR-APP-ID?subscription-key=YOUR-SUBSCRIPTION-KEY</code>
Southeast Asia	<code>https://southeastasia.api.cognitive.microsoft.com/luis/v2.0/apps/YOUR-APP-ID?subscription-key=YOUR-SUBSCRIPTION-KEY</code>

To publish to the European regions, you can create LUIS apps at <https://eu.luis.ai>.

AZURE REGION	ENDPOINT URL FORMAT
--------------	---------------------

AZURE REGION	ENDPOINT URL FORMAT
West Europe	<code>https://westeurope.api.cognitive.microsoft.com/luis/v2.0/apps/YOUR_APP-ID?subscription-key=YOUR-SUBSCRIPTION-KEY</code>

Data privacy

To keep LUIS from saving the user's utterances, set the `log` parameter to false in the [LuisModel attribute](#).

```
[LuisModel("<YOUR_LUIS_APP_ID>", "YOUR_SUBSCRIPTION_KEY", Log = false)]
[Serializable]
public class SimpleNoteDialog : LuisDialog<object>
{
    // ...
}
```

Next steps

From trying the bot, you can see how tasks are invoked by a LUIS intent. However, this simple example doesn't allow for interruption of the currently active dialog. Allowing and handling interruptions like "help" or "cancel" is a flexible design that accounts for what users really do. Learn more about using scorable dialogs so that your dialogs can handle interruptions.

[Global message handlers using scorables](#)

Additional resources

- [Dialogs](#)
- [Manage conversation flow with dialogs](#)
- [Language understanding](#)
- [LUIS](#)
- [Bot Builder SDK for .NET Reference](#)

Request payment

11/2/2017 • 9 min to read • [Edit Online](#)

If your bot enables users to purchase items, it can request payment by including a special type of button within a [rich card](#). This article describes how to send a payment request using the Bot Builder SDK for .NET.

Prerequisites

Before you can send a payment request using the Bot Builder SDK for .NET, you must complete these prerequisite tasks.

Register and configure your bot

1. [Register](#) your bot with the Bot Framework.
2. Update your bot's **Web.config** file to set `MicrosoftAppId` and `MicrosoftAppPassword` to the app ID and password values that were generated for your bot during the registration process.

Create and configure merchant account

1. [Create and activate a Stripe account if you don't have one already.](#)
2. [Sign in to Seller Center with your Microsoft account.](#)
3. Within Seller Center, connect your account with Stripe.
4. Within Seller Center, navigate to the Dashboard and copy the value of **MerchantID**.
5. Update your bot's **Web.config** file to set `MerchantId` to the value that you copied from the Seller Center Dashboard.

Payment process overview

The payment process comprises three distinct parts:

1. The bot sends a payment request.
2. The user signs in with a Microsoft account to provide payment, shipping, and contact information. Callbacks are sent to the bot to indicate when the bot needs to perform certain operations (update shipping address, update shipping option, complete payment).
3. The bot processes the callbacks that it receives, including shipping address update, shipping option update, and payment complete.

Your bot must implement only step one and step three of this process; step two takes place outside the context of your bot.

Payment Bot sample

The [Payment Bot](#) sample provides an example of a bot that sends a payment request using .NET. To see this sample bot in action, you can [try it out in web chat](#), [add it as a Skype contact](#) or download the payment bot sample and run it locally using the Bot Framework Emulator.

NOTE

To complete the end-to-end payment process using the **Payment Bot** sample in web chat or Skype, you must specify a valid credit card or debit card within your Microsoft account (i.e., a valid card from a U.S. card issuer). Your card will not be charged and the card's CVV will not be verified, because the **Payment Bot** sample runs in test mode (i.e., `LiveMode` is set to `false` in `Web.config`).

The next few sections of this article describe the three parts of the payment process, in the context of the **Payment Bot** sample.

Requesting payment

Your bot can request payment from a user by sending a message that contains a [rich card attachment](#) with a button that specifies `CardAction.Type` of "payment". This code snippet from the **Payment Bot** sample creates a message that contains a Hero card with a **Buy** button that the user can click (or tap) to initiate the payment process.

```
var replyMessage = context.MakeMessage();

replyMessage.Attachments = new List<Attachment>();

var displayedItem = await new CatalogService().GetRandomItemAsync();

var cartId = displayedItem.Id.ToString();
context.ConversationData.SetValue(CART_KEY, cartId);
context.ConversationData.SetValue(cartId, context.Activity.From.Id);

var heroCard = new HeroCard
{
    Title = displayedItem.Title,
    Subtitle = $"{displayedItem.Currency} {displayedItem.Price.ToString("F")}",
    Text = displayedItem.Description,
    Images = new List<CardImage>
    {
        new CardImage
        {
            Url = displayedItem.ImageUrl
        }
    },
    Buttons = new List<CardAction>
    {
        new CardAction
        {
            Title = "Buy",
            Type = PaymentRequest.PaymentActionType,
            Value = BuildPaymentRequest(cartId, displayedItem, MethodData)
        }
    }
};

replyMessage.Attachments.Add(heroCard.ToAttachment());

await context.PostAsync(replyMessage);
```

In this example, the button's type is specified as `PaymentRequest.PaymentActionType`, which the Bot Builder library defines as "payment". The button's value is populated by the `BuildPaymentRequest` method, which returns a `PaymentRequest` object that contains information about supported payment methods, details, and options. For more information about implementation details, see [Dialogs/RootDialog.cs](#) within the **Payment Bot** sample.

This screenshot shows the Hero card (with **Buy** button) that's generated by the code snippet above.

paymentsample +

← → ⌂ | 🔒 webchat.botframework.com/embed/paymentsample?si= Chat



Scott Gu - Favorite Shirt
USD 1.99
Shiny red, ready to rock on Keynotes
Buy

paymentsample at 3:33:59 PM

Type your message... 

IMPORTANT

Any user that has access to the **Buy** button may use it to initiate the payment process. Within the context of a group conversation, it is not possible to designate a button for use by only a specific user.

User experience

When a user clicks the **Buy** button, he or she is directed to a payment web experience to provide all required payment, shipping, and contact information via their Microsoft account.

The screenshot shows a Microsoft Edge browser window with a 'Confirm and Pay' page. The page has several dropdown menus: 'Pay with' (empty), 'Ship to' (set to 'Select shipping address'), 'Shipping options' (empty), 'Email receipt to' (empty), and 'Phone number' (empty). Below these is a summary row: 'Total (USD) [Show details](#)' followed by '\$1.99*'. A small note below the total says '* - Indicates the cost isn't final'. At the bottom is a large blue button labeled 'Pay'.

HTTP callbacks

HTTP callbacks will be sent to your bot to indicate that it should perform certain operations. Each callback will be an [activity](#) that contains these property values:

PROPERTY	VALUE
<code>Activity.Type</code>	invoke
<code>Activity.Name</code>	Indicates the type of operation that the bot should perform (e.g., shipping address update, shipping option update, payment complete).
<code>Activity.Value</code>	The request payload in JSON format.
<code>Activity.RelatesTo</code>	Describes the channel and user that are associated with the payment request.

NOTE

`invoke` is a special activity type that is reserved for use by the Microsoft Bot Framework. The sender of an `invoke` activity will expect your bot to acknowledge the callback by sending an HTTP response.

Processing callbacks

When your bot receives a callback, it should verify that the information specified in the callback is valid and acknowledge the callback by sending an HTTP response.

Shipping Address Update and Shipping Option Update callbacks

When receiving a Shipping Address Update or a Shipping Option Update callback, your bot will be provided with the current state of the payment details from the client in the `Activity.Value` property. As a merchant, you should treat these callbacks as static, given input payment details you will calculate some output payment details and fail if the input state provided by the client is invalid for any reason. If the bot determines the given information is valid as-is, simply send HTTP status code `200 OK` along with the unmodified payment details. Alternatively, the bot may send HTTP status code `200 OK` along with an updated payment details that should be applied before the order can be processed. In some cases, your bot may determine that the updated information is invalid and the order cannot be processed as-is. For example, the user's shipping address may specify a country to which the product supplier does not ship. In that case, the bot may send HTTP status code `200 OK` and a message populating the `error` property of the payment details object. Sending any HTTP status code in the `400` or `500` range to will result in a generic error for the customer.

Payment Complete callbacks

When receiving a Payment Complete callback, your bot will be provided with a copy of the initial, unmodified payment request as well as the payment response objects in the `Activity.Value` property. The payment response object will contain the final selections made by the customer along with a payment token. Your bot should take the opportunity to recalculate the final payment request based on the initial payment request and the customer's final selections. Assuming the customer's selections are determined to be valid, the bot should verify the amount and currency in the payment token header to ensure that they match the final payment request. If the bot decides to charge the customer it should only charge the amount in the payment token header as this is the price the customer confirmed. If there is a mismatch between the values that the bot expects and the values that it received in the Payment Complete callback, it can fail the payment request by sending HTTP status code `200 OK` along with setting the `result` field to `failure`.

In addition to verifying payment details, the bot should also verify that the order can be fulfilled, before it initiates payment processing. For example, it may want to verify that the item(s) being purchased are still available in stock. If the values are correct and your payment processor has successfully charged the payment token, then the bot should respond with HTTP status code `200 OK` along with setting the `result` field to `success` in order for the payment web experience to display the payment confirmation. The payment token that the bot receives can only be used once, by the merchant that requested it, and must be submitted to Stripe (the only payment processor that the Bot Framework currently supports). Sending any HTTP status code in the `400` or `500` range to will result in a generic error for the customer.

The `OnInvoke` method in the **Payment Bot** sample processes the callbacks that the bot receives.

```

[MethodBind]
[ScorableGroup(1)]
private async Task OnInvoke(IInvokeActivity invoke, IConnectorClient connectorClient, IStateClient
stateClient, HttpResponseMessage response, CancellationToken token)
{
    MicrosoftAppCredentials.TrustServiceUrl(invoke.RelatesTo.ServiceUrl);

    var jobject = invoke.Value as JObject;
    if (jobject == null)
    {
        throw new ArgumentException("Request payload must be a valid json object.");
    }

    // This is a temporary workaround for the issue that the channelId for "webchat" is mapped to "directline"
    // in the incoming RelatesTo object
    invoke.RelatesTo.ChannelId = (invoke.RelatesTo.ChannelId == "directline") ? "webchat" :
    invoke.RelatesTo.ChannelId;

    if (invoke.RelatesTo.User == null)
    {
        // Bot keeps the userId in context.ConversationData[cartId]
        var conversationData = await stateClient.BotState.GetConversationDataAsync(invoke.RelatesTo.ChannelId,
        invoke.RelatesTo.Conversation.Id, token);
        var cartId = conversationData.GetProperty<string>(RootDialog.CARTKEY);

        if (!string.IsNullOrEmpty(cartId))
        {
            invoke.RelatesTo.User = new ChannelAccount
            {
                Id = conversationData.GetProperty<string>(cartId)
            };
        }
    }

    var updateResponse = default(object);

    switch (invoke.Name)
    {
        case PaymentOperations.UpdateShippingAddressOperationName:
            updateResponse = await ProcessShippingAddressUpdate(jobject.ToObject<PaymentRequestUpdate>(),
            token);
            break;

        case PaymentOperations.UpdateShippingOptionOperationName:
            updateResponse = await ProcessShippingOptionUpdate(jobject.ToObject<PaymentRequestUpdate>(),
            token);
            break;

        case PaymentOperations.PaymentCompleteOperationName:
            updateResponse = await ProcessPaymentComplete(invoke, jobject.ToObject<PaymentRequestComplete>(),
            token);
            break;

        default:
            throw new ArgumentException("Invoke activity name is not a supported request type.");
    }

    response.Content = new ObjectContent<object>(
        updateResponse,
        this.Configuration.Formatters.JsonFormatter,
        JsonMediaTypeFormatter.DefaultMediaType);

    response.StatusCode = HttpStatusCode.OK;
}

```

In this example, the bot examines the `Name` property of the incoming activity to identify the type of operation it

needs to perform, and then calls the appropriate method to process the callback. For more information about implementation details, see [Controllers/MessagesControllers.cs](#) within the [Payment Bot](#) sample.

Testing a payment bot

To fully test a bot that requests payment, you can [deploy](#) it to the cloud. After you have deployed your bot to the cloud, [configure](#) it to run on channels that support Bot Framework payments, like Web Chat and Skype.

Alternatively, you can test your bot locally using the [Bot Framework Emulator](#).

TIP

Callbacks are sent to your bot when a user changes data or clicks **Pay** during the payment web experience. Therefore, you can test your bot's ability to receive and process callbacks by interacting with the payment web experience yourself.

In the [Payment Bot](#) sample, the `LiveMode` configuration setting in [Web.config](#) determines whether Payment Complete callbacks will contain emulated payment tokens or real payment tokens. If `LiveMode` is set to `false`, a header is added to the bot's outbound payment request to indicate that the bot is in test mode, and the Payment Complete callback will contain an emulated payment token that cannot be charged. If `LiveMode` is set to `true`, the header which indicates that the bot is in test mode is omitted from the bot's outbound payment request, and the Payment Complete callback will contain a real payment token that the bot will submit to Stripe for payment processing. This will be a real transaction that results in charges to the specified payment instrument.

Additional resources

- [Payment Bot sample](#)
- [Activities overview](#)
- [Add rich cards to messages](#)
- [Web Payments at W3C](#)
- [Bot Builder SDK for .NET Reference](#)

Create data-driven experiences with Azure Search

8/7/2017 • 4 min to read • [Edit Online](#)

You can add [Azure Search](#) to a bot to help users navigate large amounts of content and create a data-driven exploration experience.

Azure Search is an Azure service that offers keyword search, built-in linguistics, custom scoring, faceted navigation, and more. Azure Search can also index content from various sources, including Azure SQL DB, DocumentDB, Blob Storage, and Table Storage. It supports "push" indexing for other sources of data, and it can open PDFs, Office documents, and other formats containing unstructured data. Once collected, the content goes into an Azure Search index, which the bot can then query.

Prerequisites

Install the [Microsoft.Azure.Search](#) Nuget package in your bot project.

The following three C# projects are required in your bot's solution. These projects provide additional functionality for bots and Azure Search. Fork the projects from [GitHub](#) or download the source code directly.

- [Search.Azure](#) defines the Azure Service call.
- [Search.Contracts](#) defines generic interfaces and data models to handle data.
- [Search.Dialogs](#) includes various generic Bot Builder dialogs used to query Azure Search.

Configure Azure Search settings

Configure the Azure Search settings in the **Web.config** file of the project using your own Azure Search credentials in the value fields. The constructor in the `AzureSearchClient` class will use these settings to register and bind the bot to the Azure Service.

```
<appSettings>
    <add key="SearchDialogsServiceName" value="Azure-Search-Service-Name" /> <!-- replace value field with
    Azure Service Name -->
    <add key="SearchDialogsServiceKey" value="Azure-Search-Service-Primary-Key" /> <!-- replace value field
    with Azure Service Key -->
    <add key="SearchDialogsIndexName" value="Azure-Search-Service-Index" /> <!-- replace value field with your
    Azure Search Index -->
</appSettings>
```

Create a search dialog

In your bot's project, create a new `AzureSearchDialog` class to call the Azure Service in your bot. This new class must inherit the `SearchDialog` class from the **Search.Dialogs** project, which handles most of the heavy lifting. The `GetTopRefiners()` override allows users to narrow/filter their search results without having to start the search over from the beginning, maintaining the search object's state. You can add your own custom refiners in the `TopRefiners` array to let your users filter or narrow down their search results.

```
[Serializable]
public class AzureSearchDialog : SearchDialog
{
    private static readonly string[] TopRefiners = { "refiner1", "refiner2", "refiner3" }; // define your own
    custom refiners

    public AzureSearchDialog(ISearchClient searchClient) : base(searchClient, multipleSelection: true)
    {
    }

    protected override string[] GetTopRefiners()
    {
        return TopRefiners;
    }
}
```

Define the response data model

The **SearchHit.cs** class within the `Search.Contracts` project defines the relevant data to be parsed from the Azure Search response. For your bot the only mandatory inclusions are the `PropertyBag` `IDictionary` declaration and creation in the constructor. You can define all other properties in this class relative to your bot's needs.

```
[Serializable]
public class SearchHit
{
    public SearchHit()
    {
        this.PropertyBag = new Dictionary<string, object>();
    }

    public IDictionary<string, object> PropertyBag { get; set; }

    // customize the fields below as needed
    public string Key { get; set; }

    public string Title { get; set; }

    public string PictureUrl { get; set; }

    public string Description { get; set; }
}
```

After Azure Search responds

Upon a successful query to the Azure Service, the search result will need to be parsed to retrieve the relevant data for the bot to display to the user. To enable this, you'll need to create a `SearchResultMapper` class. The `GenericSearchResult` object created in the constructor defines a list and dictionary to store results and facets respectively after each result is parsed respective to your bot's data models.

Synchronize the properties in the `ToSearchHit` method to match the data model in **SearchHit.cs**. The `ToSearchHit` method will be executed and generate a new `SearchHit` for every result found in the response.

```

public class SearchResultMapper : IMapper<DocumentSearchResult, GenericSearchResult>
{
    public GenericSearchResult Map(DocumentSearchResult documentSearchResult)
    {
        var searchResult = new GenericSearchResult();

        searchResult.Results = documentSearchResult.Results.Select(r => ToSearchHit(r)).ToList();
        searchResult.Facets = documentSearchResult.Facets?.ToDictionary(kv => kv.Key, kv => kv.Value.Select(f => ToFacet(f)));

        return searchResult;
    }

    private static GenericFacet ToFacet(FacetResult facetResult)
    {
        return new GenericFacet
        {
            Value = facetResult.Value,
            Count = facetResult.Count.Value
        };
    }

    private static SearchHit ToSearchHit(SearchResult hit)
    {
        return new SearchHit
        {
            // custom properties defined in SearchHit.cs
            Key = (string)hit.Document["id"],
            Title = (string)hit.Document["title"],
            PictureUrl = (string)hit.Document["thumbnail"],
            Description = (string)hit.Document["description"]
        };
    }
}

```

After the results are parsed and stored, the information still needs to be displayed to the user. The `SearchHitStyler` class will need to be managed to accommodate the your data model from the `SearchHit` class. For example, the `Title`, `PictureUrl`, and `Description` properties from the **SearchHit.cs** class are used in the sample code to create a new card attachments. The following code creates a new card attachment for every `SearchHit` object in the `GenericSearchResult` Results list to display to the user.

```
[Serializable]
public class SearchHitStyler : PromptStyler
{
    public override void Apply<T>(ref IMessageActivity message, string prompt, IReadOnlyList<T> options,
IReadOnlyList<string> descriptions = null)
    {
        var hits = options as IList<SearchHit>;
        if (hits != null)
        {
            var cards = hits.Select(h => new ThumbnailCard
            {
                Title = h.Title,
                Images = new[] { new CardImage(h.PictureUrl) },
                Buttons = new[] { new CardAction(ActionTypes.ImBack, "Pick this one", value: h.Key) },
                Text = h.Description
            });

            message.AttachmentLayout = AttachmentLayoutTypes.Carousel;
            message.Attachments = cards.Select(c => c.ToAttachment()).ToList();
            message.Text = prompt;
        }
        else
        {
            base.Apply<T>(ref message, prompt, options, descriptions);
        }
    }
}
```

The search results are displayed to the user and you've successfully added Azure Search to your bot.

Samples

For two complete samples that show how to support Azure Search with bots using the Bot Builder SDK for .NET, see the [Real Estate bot sample](#) or [Job Listing bot sample](#) in GitHub.

Additional resources

- [Azure Search](#)
- [Dialogs overview](#)
- [Azure Search bot samples](#)

Secure your bot

8/7/2017 • 1 min to read • [Edit Online](#)

Your bot can be connected to many different communication channels (Skype, SMS, email, and others) through the Bot Framework Connector service. This article describes how to secure your bot by using HTTPS and Bot Framework authentication.

Use HTTPS and Bot Framework authentication

To ensure that your bot's endpoint can only be accessed by the Bot Framework [Connector](#), configure your bot's endpoint to use only HTTPS and enable Bot Framework authentication by [registering](#) your bot to acquire its app Id and password.

Configure authentication for your bot

After you have registered your bot, specify its app Id and password in your bot's web.config file.

```
<appSettings>
  <add key="MicrosoftAppId" value="_appIdValue_" />
  <add key="MicrosoftAppPassword" value="_passwordValue_" />
</appSettings>
```

Then, use the `[BotAuthentication]` attribute to specify authentication credentials when using the Bot Builder SDK for .NET to create your bot.

To use the authentication credentials that are stored in the web.config file, specify the `[BotAuthentication]` with no parameters.

```
[BotAuthentication]
public class MessagesController : ApiController
{
}
```

To use other values for authentication credentials, specify the `[BotAuthentication]` attribute and pass in those values.

```
[BotAuthentication(MicrosoftAppId = "_appIdValue_", MicrosoftAppPassword=_passwordValue_")]
public class MessagesController : ApiController
{
}
```

Additional resources

- [Bot Builder SDK for .NET](#)
- [Key concepts in the bot Builder SDK for .NET](#)
- [Register a bot with the Bot Framework](#)

Release notes

11/2/2017 • 7 min to read • [Edit Online](#)

This article describes changes introduced by each version of the Bot Builder SDK for .NET.

IMPORTANT

As long as the Microsoft Bot Framework is in Preview mode, you should expect breaking changes in new versions of the Bot Builder SDK for .NET. See the [Bot Builder SDK GitHub repository](#) for a list of known issues.

v3.11.0

Changes

- Added support for speech-enabled channels (e.g., [Cortana](#)).
- Added support for [payment requests](#).
- Added prompt recognizers to improve the `PromptDialog` parser.
- Implemented general bug fixes and improvements.

v3.5.5

Changes

- Added `ConversationReference` (as a replacement to deprecated `ResumptionCookie`).
- Changed default LUIS host in LUIS service and deprecated LUIS v1 endpoint.
- Implemented general bug fixes.

v3.5.3

Breaking changes

- Changed the FormFlow prompter so that the state and current field are passed in. This only impacts bots that use a custom prompter instead of the default prompter in FormFlow.

Changes

- Improved exception propagation and messaging for bot authentication failures.

v3.5.2

Changes

- Fixed some FormFlow issues and added markdown support for FormFlow multi-line prompts.
- Updated `ActivityResolver` to support `IInvokeActivity`.
- Added `AlteredQuery` to `LuisResult`.

v3.5.1

Changes

- Deprecated `ITriggerActivity`. `IEventActivity` replaces `ITriggerActivity`.
- Added new activity types (`event` and `invoke`) to the Connector.
- Prepared the release of .NET core support for Microsoft Bot Connector.

- Implemented general bug fixes and code refactoring.

v3.5.0

Changes

- Updated to Bot Framework v3.1 `JwtToken`. For more information about v3.1 token changes, see [Authentication](#).
- Implemented general bug fixes and code refactoring.

v3.4.0

Breaking changes

- Renames `ScorableOrder` attribute to `ScorableGroup` in `DispatchDialog`. This only impacts bots that are using `ScorableOrder` in their `DispatchDialog`.

Changes

- Implemented improvements to `DispatchDialog`.
- Added support for [LUIS](#) API v2 and LUIS action dialog.
- Automatically remember last wait for each frame of stack.
- Implemented general bug fixes.

v3.3.3

Breaking changes

- `ICredentialProvider` should now implement `IsAuthenticatedAsync`. This only impacts bots that are using `ICredentialProvider` for their `BotAuthentication`.

Changes

- Added support for new `./media` card types (e.g. `AnimationCard`, `VideoCard`, and `AudioCard`).
- Added new `BotAuthenticator` utility class that can be used for bot authentication instead of using the `BotAuthentication` attribute.
- Implemented general bug fixes.

v3.3.1

Changes

- Added intent-based dispatch dialog.
- Added dialog task manager to enable multiple stacks per conversation.
- Added credential provider to enable multi-bot authentication.
- Added bot authenticator facility to enable non-attribute-based bot authentication scenarios.
- Implemented general bug fixes.

v3.3.0

Breaking changes

- Updated `IScorable` interface. This only impacts bots that use `ScoringDialogTask` and `IScorable` implementation to interrupt the conversation based on the score assigned to the incoming activity.

Changes

- Removed unnecessary dependencies from `bot.builder` `nuspec`.

- Added support for keyboard card and Facebook quick replies.
- Added scorable dispatch support.
- Implemented general bug fixes.

v3.2.1

Changes

- Factored out `Address` from `ResumptionCookie`.
- Serialized dialog execution pessimistically by conversation.
- Added `Then dialog` to enhance chaining of dialogs.
- Implemented general bug fixes.

v3.2.0

Breaking changes

- `IField<T>.FieldDescription` is now a `DescribeAttribute`. This only impacts bots that implement their own `Field<T>` class.

Changes

- Improved card support in Form dialog.
- Made Incoming activity available in `LuisDialog` intent handlers.
- Added a mechanism to serialize incoming requests for a conversation.
- Added LUIS resolution parser.
- Updated the Connector to depend on `Microsoft.Rest.ClientRuntime` v2.3.2.
- Implemented general bug fixes.

v3.1.0

Breaking changes

- `BotAuthentication` attribute now inherits from `ActionFilterAttribute` and not `AuthorizationFilterAttribute`.

Changes

- Made `ETag` consistency policy the default data consistency policy for `IBotDataStore`.
- Added better exception translation/propagation to bot builder internals.
- Made Bot authentication more reliable.
- Added trusted service urls to `MicrosoftAppCredentials`.
- Implemented general bug fixes.

v3.0.1

Changes

- Fixed an issue with `MicrosoftAppCredentials` not setting `AuthToken`.

v3.0.0

NOTE

This version of the SDK reflects changes that were implemented in version 3 of the Bot Connector API.

Breaking changes

- Updated schema to v3. `Message` is now [Activity](#) and there is a new addressing scheme.
- Changed reply model such that replies to the user will be sent asynchronously over a separately initiated HTTP request rather than inline with the HTTP POST for the incoming message to bot.
- Updated [Authentication model](#).
- Decoupled bot data storage [Bot State](#) from messaging API.
- Added [new card format](#) for attachments.

Changes

- Combined Bot Builder and Bot Connector into one [NuGet package](#).
- Made Bot Connector open source.
- Published [Bot.Builder.Calling](#) NuGet package that can be used to build [Skype Calling bots](#).

v1.2.5

Changes

- Moved `Microsoft.Bot.Builder.FormFlow.Json` into a separate NuGet package.
- Added `PromptDialog` Attachment.
- Added NuGet symbols to [NuGet.org](#).
- Implemented general bug fixes.

v1.2.4

Breaking changes

- Renamed `IFormBuilder.OnCompletionAsync` to `OnCompletion` (since the method itself is not async).
- Implemented significant changes to JSON FormFlow.
- Implemented change such that missing resources will cause an error to be thrown.

New features

- JSON FormFlow now supports specifying everything through the JSON schema file:
 1. There is a completely separate builder, `FormBuilderJson` which takes the schema.
 2. JSON Schema now supports message, confirmation, validation with regex and dynamically compiled code.
 3. You no longer need to specify the schema on every field.
 4. It is packaged as a separate dll (`Microsoft.Bot.Builder.FormFlow.Json`) that depends on `Microsoft.CodeAnalysis.CSharp.Scripting`, which brings in several dependencies.
- `LuisDialog` supports multiple `LuisModel` and `ILuisService` instances.
- New attribute `PatternAttribute` for field validation via regex.
- `IBotDataStore` now allows supplying your own storage implementation.

Bug fixes

- [#414](#) If an entity did not match a value, a blank string would be returned.
- [#434](#) Add calculator scorables/dialog as an example of `IDialogStack.Forward` from `IScorable.PostAsync`.
- [#465](#) Preserve exception stack trace when re-throwing.
- [#466](#) Add a typed `TooManyAttemptsException` for prompts.
- [#472](#) Test showing how to resolve dynamic form from container.
- [#416](#) Fix bug in enumeration where if a term included numbers it would result in no match.
- [#440](#) Fix syntax error in Japanese localization.
- [#446](#) Expose a `Confirm` method that was mistakenly marked internal.

- #449 `InitialUpper` for `Normalize` did not return value.
- Make it so validation feedback is surfaced if `FeedbackOptions.Always`.
- Buttons did not include no preference.
- JSON Forms were not handling optional correctly.

v1.2.3

Changes

- Updated FormFlow recognizers to take consistent set of args.
- Added `CancelScorable` as example of `IScorable`.
- Added sample using Azure Active Directory Authentication to access Microsoft graph.

Bug fixes

- #227 FormFlow now handles clarification and verification of [LUIS](#) entities. Initial messages are processed, then LUIS entities, and then remaining steps.
- #335 Made it so that n-gram generation would not include empty strings if there were multiple spaces.
- #400 Resolved issue with the way that ETag is set.

v1.2.2

Breaking changes

- There is no longer a dependency on `Newtonsoft.Json.Schema`, so you can use `JObject.Parse` to parse your JSON Schema and define forms.

Changes

- Made it possible to resolve root dialog from the container.
- Decoupled `IPostToBot` "middleware" from `IDialogTask`.
- Pushed lazy dialog instantiation into `DialogTask`.
- Factored out `ReactiveDialogTask` from `DialogTask`.
- Implemented forwarding of an item to child dialog.
- Added persistent dialog task.
- Added example of `AlwaysSendDirect_BotToUser`.
- Added **1** and **2** as possible responses to boolean recognizer (because buttons might send these values).

Bug fixes

- #227 Fix issue related to processing of [LUIS](#) entities.
- #230 Allow `ConnectorClientCredentials` to be injected from container.

v1.2.1

Changes

- Ensured that [LUIS](#) service queries are encoded with UTF8.
- Fixed a `Choice` prompt bug to rank complete matches higher than partial matches.

v1.2.0.1

Changes

- Fixed missing dependencies for `Microsoft.Bot.Builder` 1.2.0 NuGet package.

v1.2.0

Breaking changes

- Changed target framework to .NET 4.6. This change was necessary to reliably support using the thread culture for localization.
- Made it so that FormFlow `ValidateAsyncDelegate` must return the value to set in the field. This change was necessary to support programmatic value transformations.
- Changed the signature of `Conversation.Resume` to support a resumption cookie to maintain conversation state across dialog resumption.
- Moved to the latest NuGet packages, including for the Bot Framework Connector.

New features

- System dialogs and FormFlow will now generate buttons for channels that support them.
- FormFlow can now be driven by an extended JSON Schema that allows doing attributes in a similar way to C#. This allows forms to be generated at run-time from data rather than C# reflection.
- System dialogs and FormFlow are localized to nine languages. (Contributions for more languages would be welcome.) We also provide tools to help generate the resource files required to localize your FormFlow state classes.
- `DateTime` parsing in English now uses **Chronic**, which supports more natural Date/Time expressions such as "tomorrow at 4".
- `LuisDialog` now supports the full [LUIS](#) schema, including actions.

Bug fixes

- Implemented general bug fixes.

v1.1.0

Breaking changes

- Renamed some delegates and methods for consistency and to support dynamic field definition. This only impacts bots that were directly using `Field` or `FieldReflector`.

Changes

- Provided a way to dynamically define fields, confirmations and messages.
- Added `FormCanceledException`, which provides information on what steps were completed and where the user quit.
- Added more flexibility on how parenthesis are used when generating prompts.
- Fixed several bugs related to initial state and [LUIS](#) entities.
- Extended chain model to support branching (`Chain.Switch`).
- Added support for resumption of a conversation.
- Added example of Facebook OAuth.

v1.0.2

Changes

- Moved to `IDialog<T>` typed for result type.
- Added support for LINQ query syntax (e.g. `Select`, `SelectMany`).
- Multiple `IBotToUser.Post(Message)` calls.
- Moved to `Autofac` dependency injection container.
- Made it so that `IConnectorClient` is instantiated to point to emulator when emulating bot.
- Fixed an issue with `CommandDialog<T>`.

- Updated [LUIS](#) models.
- Added `ChoiceCase`, `ChoiceParens` to Form template attributes.

v1.0.1

Changes

- Fixed `LuisDialog` to handle **null** score returned by LUIS (due to a behavior change in Cortana pre-built apps by [LUIS](#)).
- Updated **nuspec** with better description.
- Added error-resilient context store.

Bot Builder SDK for Node.js

8/9/2017 • 1 min to read • [Edit Online](#)

Bot Builder SDK for Node.js is a powerful, easy-to-use framework that provides a familiar way for Node.js developers to write bots. You can use it to build a wide variety of conversational user interfaces, from simple prompts to free-form conversations.

The conversational logic for your bot is hosted as a web service. The Bot Builder SDK uses [restify](#), a popular framework for building web services, to create the bot's web server. The SDK is also compatible with [Express](#) and the use of other web app frameworks is possible with some adaption.

Using the SDK, you can take advantage of the following SDK features:

- Powerful system for building dialogs to encapsulate conversational logic.
- Built-in prompts for simple things such as Yes/No, strings, numbers, and enumerations, as well as support for messages containing images and attachments, and rich cards containing buttons.
- Built-in support for powerful AI frameworks such as [LUIS](#).
- Built-in recognizers and event handlers that guide the user through the conversation, providing help, navigation, clarification, and confirmation as needed.

Get started

If you are new to writing bots, [create your first bot with Node.js](#) with step-by-step instructions to help you set up your project, install the SDK, and run your first bot.

If you are new to the Bot Builder SDK for Node.js, you can start with key concepts that help you understand the major components of the Bot Builder SDK, see [Key concepts](#).

To ensure your bot addresses the top user scenarios, review the [design principles](#) and [explore patterns](#) for guidance.

Get samples

The [Bot Builder SDK for Node.js samples](#) demonstrate task-focused bots that show how to take advantage of features in the Bot Builder SDK for Node.js. You can use the samples to help you quickly get started with building great bots with rich capabilities.

Next steps

[Key concepts](#)

Additional resources

The following task-focused how-to guides demonstrate various features of the Bot Builder SDK for Node.js.

- [Respond to messages](#)
- [Handle user actions](#)
- [Recognize user intent](#)
- [Send a rich card](#)
- [Send attachments](#)
- [Saving user data](#)

If you encounter problems or have suggestions regarding the Bot Builder SDK for Node.js, see [Support](#) for a list of available resources.

Key concepts in the Bot Builder SDK for Node.js

9/18/2017 • 5 min to read • [Edit Online](#)

This article introduces key concepts in the Bot Builder SDK for Node.js. For an introduction to Bot Framework, see [Bot Framework overview](#).

Connector

The Bot Framework Connector is a service that connects your bot to multiple *channels*, which are clients like Skype, Facebook, Slack, and SMS. The Connector facilitates communication between bot and user by relaying messages from bot to channel and from channel to bot. Your bot's logic is hosted as a web service that receives messages from users through the Connector service, and your bot's replies are sent to the Connector using HTTPS POST.

The Bot Builder SDK for Node.js provides the [UniversalBot](#) and [ChatConnector](#) classes for configuring the bot to send and receive messages through the Bot Framework Connector. The `UniversalBot` class forms the brains of your bot. It's responsible for managing all the conversations your bot has with a user. The `ChatConnector` class connects your bot to the Bot Framework Connector Service. For an example that demonstrates using these classes, see [Create a bot with the Bot Builder SDK for Node.js](#).

The Connector also normalizes the messages that the bot sends to channels so that you can develop your bot in a platform-agnostic way. Normalizing a message involves converting it from the Bot Framework's schema into the channel's schema. In cases where the channel does not support all aspects of the framework's schema, the Connector will try to convert the message to a format that the channel supports. For example, if the bot sends a message that contains a card with action buttons to the SMS channel, the Connector may render the card as an image and include the actions as links in the message's text. The [Channel Inspector](#) is a web tool that shows you how the Connector renders messages on various channels.

The `chatConnector` requires an API endpoint to be setup within your bot. With the Node.js SDK, this is usually accomplished by installing the `restify` Node.js module. Bots can also be created for the console using the [ConsoleConnector](#), which does not require an API endpoint.

Messages

Messages can consist of text to be displayed, text to be spoken, attachments, rich cards, and suggested actions. You use the `session.send` method to send messages in response to a message from the user. Your bot may call `send` as many times as it likes in response to a message from the user. For an example that demonstrates this, see [Respond to user messages](#).

For an example that demonstrates how to send a rich graphical card containing interactive buttons that the user clicks to initiate an action, see [Add rich cards to messages](#). For an example that demonstrates how to send and receive attachments, see [Send attachments](#). For an example that demonstrates how to send a message that specifies text to be spoken by your bot on a speech-enabled channel, see [Add speech to messages](#). For an example that demonstrates how to send suggested actions, see [Send suggested actions](#).

Dialogs

Dialogs help you organize the conversational logic in your bot and are fundamental to [designing conversation flow](#). For an introduction to dialogs, see [Manage a conversation with dialogs](#).

Actions

You'll want to design your bot to be able to handle interruptions like requests for cancellation or help at any time during the conversation flow. The Bot Builder SDK for Node.js provides global message handlers that trigger actions like cancellation or invoking other dialogs. See [Handle user actions](#) for examples of how to use `triggerAction` handlers.

Recognizers

When users ask your bot for something, like "help" or "find news", your bot needs to understand what the user is asking for and then take the appropriate action. You can design your bot to recognize intents based on the user's input and associate that intent with actions.

You can use the built-in regular expression recognizer that the Bot Builder SDK provides, call an external service such as the LUIS API, or implement a custom recognizer to determine the user's intent. See [Recognize user intent](#) for examples that demonstrate how to add recognizers to your bot and use them to trigger actions.

Saving State

A key to good bot design is to track the context of a conversation, so that your bot remembers things like the last question the user asked. Bots built using Bot Builder SDK are designed to be stateless so that they can easily be scaled to run across multiple compute nodes. The Bot Framework provides a storage system that stores bot data, so that the bot web service can be scaled. Because of that you should generally avoid saving state using a global variable or function closure. Doing so will create issues when you want to scale out your bot. Instead, use the following properties of your bot's `session` object to save data relative to a user or conversation:

- **userData** stores information globally for the user across all conversations.
- **conversationData** stores information globally for a single conversation. This data is visible to everyone within the conversation so exercise with care when storing data to this property. It's enabled by default and you can disable it using the bot's `persistConversationData` setting.
- **privateConversationData** stores information globally for a single conversation but it is private data specific to the current user. This data spans all dialogs so it's useful for storing temporary state that you want cleaned up when the conversation ends.
- **dialogData** persists information for a single dialog instance. This is essential for storing temporary information in between the steps of a [waterfall](#) in a dialog.

For examples that demonstrate how to use these properties to store and retrieve data, see [Manage state data](#).

Natural language understanding

Bot Builder lets you use LUIS to add natural language understanding to your bot using the [LuisRecognizer](#) class. You can add an instance of a **LuisRecognizer** that references your published language model and then add handlers to take actions in response to the user's utterances. To see LUIS in action watch the following 10 minute tutorial:

- [Microsoft LUIS Tutorial](#) (video)

Next steps

[Dialogs overview](#)

Dialogs in the Bot Builder SDK for Node.js

9/6/2017 • 4 min to read • [Edit Online](#)

Dialogs in the Bot Builder SDK for Node.js allow you to model conversations and manage conversation flow. A bot communicates with a user via conversations. Conversations are organized into dialogs. Dialogs can contain waterfall steps, and prompts. As the user interacts with the bot, the bot will start, stop, and switch between various dialogs in response to user messages. Understanding how dialogs work is key to successfully designing and creating great bots.

This article introduces dialog concepts. After you read this article, then follow the links in the [Next steps](#) section to dive deeper into these concepts.

Conversations through dialogs

Bot Builder SDK for Node.js defines a conversation as the communication between a bot and a user through one or more dialogs. A dialog, at its most basic level, is a reusable module that performs an operation or collects information from a user. You can encapsulate the complex logic of your bot in reusable dialog code.

A conversation can be structured and changed in many ways:

- It can originate from your [default dialog](#).
- It can be redirected from one dialog to another.
- It can be resumed.
- It can follow a [waterfall](#) pattern, which guides the user through a series of steps or [prompts](#) the user with a series of questions.
- It can use [actions](#) that listen for words or phrases that trigger a different dialog.

You can think of a conversation like a parent to dialogs. As such, a conversation contains a *dialog stack* and maintain its own set of state data; namely, the `conversationData` and the `privateConversationData`. A dialog, on the other hand, maintains the `dialogData`. For more information on state data, see [Manage state data](#).

Dialog stack

A bot interacts with a user through a series of dialogs that are maintained on a dialog stack. Dialogs are pushed on and popped off the stack in the course of a conversation. The stack works like a normal LIFO stack; meaning, the last dialog added will be the first one to complete. Once a dialog completes then control is returned to the previous dialog on the stack.

When a bot conversation first starts or when a conversation ends, the dialog stack is empty. At this point, when the a user sends a message to the bot, the bot will respond with the *default dialog*.

Default dialog

Prior to Bot Framework version 3.5, a *root* dialog is defined by adding a dialog named `/`, which lead to naming conventions similar to that of URLs. This naming convention wasn't appropriate to naming dialogs.

NOTE

Starting with version 3.5 of the Bot Framework, the *default dialog* is registered as the second parameter in the constructor of `UniversalBot`.

The following code snippet shows how to define the default dialog when creating the `UniversalBot` object.

```
var bot = new builder.UniversalBot(connector, [
    //...Default dialog waterfall steps...
]);
```

The default dialog runs whenever the dialog stack is empty and no other dialog is triggered via LUIS or another recognizer. As the default dialog is the bot's first response to the user, the default dialog should provide some contextual information to the user, such as a list of available commands or an overview of what the bot can do.

Dialog handlers

The dialog handler manages the flow of a conversation. To progress through a conversation, the dialog handler directs the flow by starting and ending dialogs.

Starting and ending dialogs

To start a new dialog (and push it onto the stack), use `session.beginDialog()`. To end a dialog (and remove it from the stack, returning control to the calling dialog), use either `session.endDialog()` or `session.endDialogWithResult()`.

Using waterfalls and prompts

Waterfall is a simple way to model and manage conversation flow. A waterfall contains a sequence of steps. In each step, you can either complete an action on behalf of the user or **prompt** the user for information.

A waterfall is implemented using a dialog that's made up of a collection of functions. Each function defines a step in the waterfall. The following code sample shows a simple conversation that uses a two step waterfall to prompt the user for their name and greet them by name.

```
// Ask the user for their name and greet them by name.
bot.dialog('greetings', [
    function (session) {
        builder.Prompts.text(session, 'Hi! What is your name?');
    },
    function (session, results) {
        session.endDialog(`Hello ${results.response}!`);
    }
]);
```

When a bot reaches the end of the waterfall without ending the dialog, the next message from the user will restart that dialog at step one of the waterfall. This may lead to frustrations as the user may feel like they are trapped in a loop. To avoid this situation, when a conversation or dialog has come to an end, it is best practice to explicitly call `endDialog`, `endDialogWithResult`, or `endConversation`.

Next steps

To dive deeper into dialogs, it is important to understand how waterfall pattern works and how to use it to guide users through a process.

[Define conversation steps with waterfalls](#)

Define conversation steps with waterfalls

10/27/2017 • 5 min to read • [Edit Online](#)

A conversation is a series of messages exchanged between user and bot. When the bot's objective is to lead the user through a series of steps, you can use a waterfall to define the steps of the conversation.

A waterfall is a specific implementation of a [dialog](#) that is most commonly used to collect information from the user or guide the user through a series of tasks. The tasks are implemented as an array of functions where the results of the first function are passed as input into the next function, and so on. Each function typically represents one step in the overall process. At each step, the bot prompts the user for input, waits for a response, and then passes the result to the next step.

This article will help you understand how a waterfall works and how you can use it to [manage conversation flow](#).

Conversation steps

A conversation typically involves several prompt/response exchanges between the user and the bot. Each prompt/response exchange is a step forward in the conversation. You can create a conversation using a waterfall with as few as two steps.

For example, consider the following `greetings` dialog. The first step of the waterfall prompts for the user's name and the second step uses the response to greet the user by name.

```
bot.dialog('greetings', [
  // Step 1
  function (session) {
    builder.Prompts.text(session, 'Hi! What is your name?');
  },
  // Step 2
  function (session, results) {
    session.endDialog(`Hello ${results.response}`);
  }
]);
```

What makes this possible is the use of prompts. The Bot Builder SDK for Node.js provides several different types of built-in [prompts](#) that you can use to ask the user for various types of information.

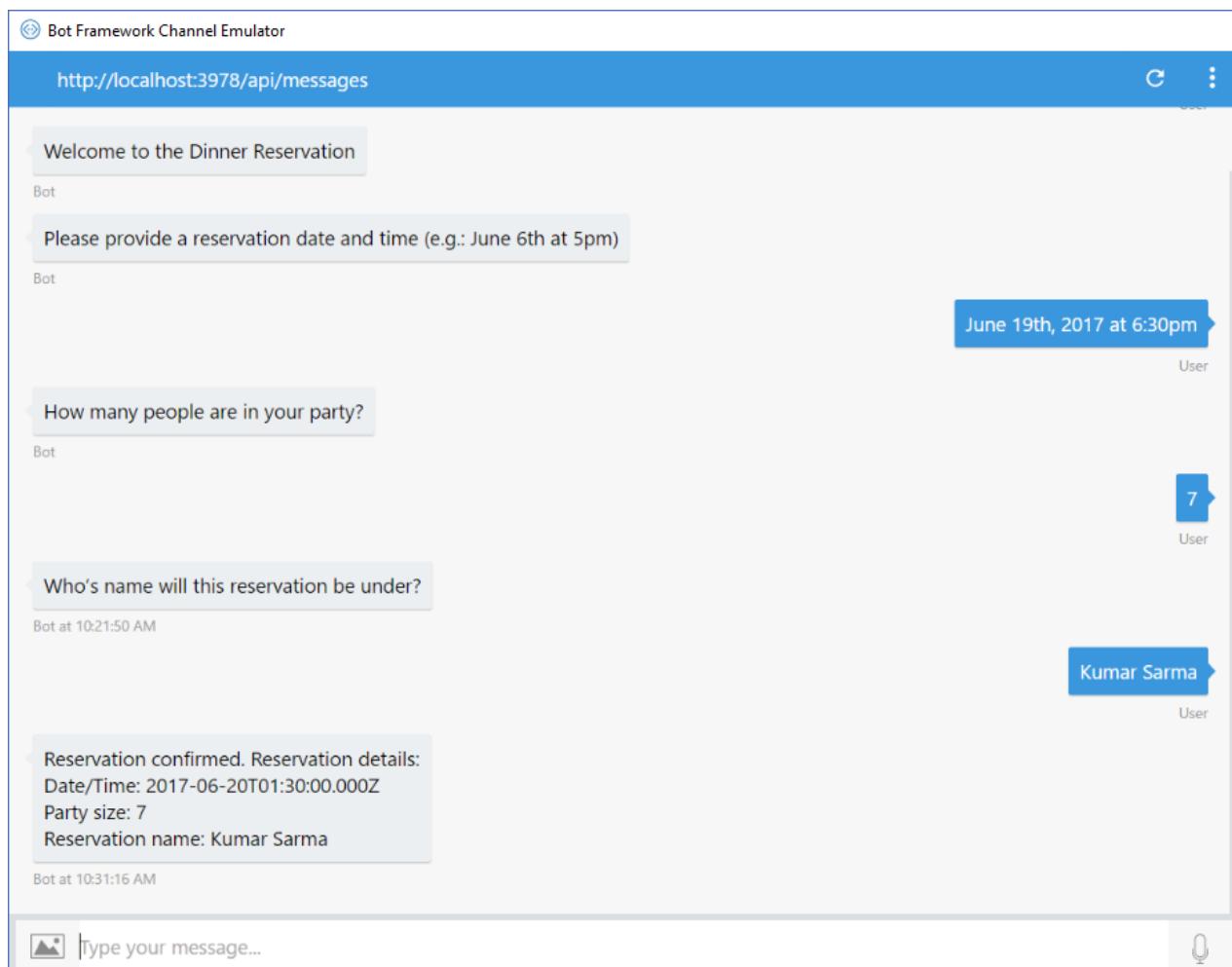
The following sample code shows a dialog that uses prompts to collect various pieces of information from the user throughout a 4-step waterfall.

```
// This is a dinner reservation bot that uses a waterfall technique to prompt users for input.
var bot = new builder.UniversalBot(connector, [
  function (session) {
    session.send("Welcome to the dinner reservation.");
    builder.Prompts.time(session, "Please provide a reservation date and time (e.g.: June 6th at 5pm)");
  },
  function (session, results) {
    session.dialogData.reservationDate = builder.EntityRecognizer.resolveTime([results.response]);
    builder.Prompts.text(session, "How many people are in your party?");
  },
  function (session, results) {
    session.dialogData.partySize = results.response;
    builder.Prompts.text(session, "Who's name will this reservation be under?");
  },
  function (session, results) {
    session.dialogData.reservationName = results.response;

    // Process request and display reservation details
    session.send(`Reservation confirmed. Reservation details: <br/>Date/Time:
${session.dialogData.reservationDate} <br/>Party size: ${session.dialogData.partySize} <br/>Reservation name:
${session.dialogData.reservationName}`);
    session.endDialog();
  }
]);
]);
```

In this example, the default dialog has four functions, each one representing a step in the waterfall. Each step prompts the user for input and sends the results to the next step to be processed. This process continues until the last step executes, thereby confirming the reservation and ending the dialog.

The following screenshot shows the results of this bot running in the [Bot Framework Emulator](#).



Create a conversation with multiple waterfalls

You can use multiple waterfalls within a conversation to define whatever conversation structure your bot requires. For example, you might use one waterfall to manage the conversation flow and use additional waterfalls to collect information from the user. Each waterfall is encapsulated in a dialog and can be invoked by calling the `session.beginDialog` function.

The following code sample shows how to use multiple waterfalls in a conversation. The waterfall within the `greetings` dialog manages the flow of the conversation, while the waterfall within the `askName` dialog collects information from the user.

```
bot.dialog('greetings', [
  function (session) {
    session.beginDialog('askName');
  },
  function (session, results) {
    session.endDialog('Hello %s!', results.response);
  }
]);
bot.dialog('askName', [
  function (session) {
    builder.Prompts.text(session, 'Hi! What is your name?');
  },
  function (session, results) {
    session.endDialogWithResult(results);
  }
]);
```

Advance the waterfall

A waterfall progresses from step to step in the sequence that the functions are defined in the array. The first function within a waterfall can receive the arguments that are passed to the dialog. Any function within a waterfall can use the `next` function to proceed to the next step without prompting user for input.

The following code sample shows how to use the `next` function within a dialog that walks the user through the process of providing information for their user profile. In each step of the waterfall, the bot prompts the user for a piece of information (if necessary), and the user's response (if any) is processed by the subsequent step of the waterfall. The final step of the waterfall in the `ensureProfile` dialog ends that dialog and returns the completed profile information to the calling dialog, which then sends a personalized greeting to the user.

```

// This bot ensures user's profile is up to date.
var bot = new builder.UniversalBot(connector, [
    function (session) {
        session.beginDialog('ensureProfile', session.userData.profile);
    },
    function (session, results) {
        session.userData.profile = results.response; // Save user profile.
        session.send(`Hello ${session.userData.profile.name}! I love ${session.userData.profile.company}!`);
    }
]);
bot.dialog('ensureProfile', [
    function (session, args, next) {
        session.dialogData.profile = args || {}; // Set the profile or create the object.
        if (!session.dialogData.profile.name) {
            builder.Prompts.text(session, "What's your name?");
        } else {
            next(); // Skip if we already have this info.
        }
    },
    function (session, results, next) {
        if (results.response) {
            // Save user's name if we asked for it.
            session.dialogData.profile.name = results.response;
        }
        if (!session.dialogData.profile.company) {
            builder.Prompts.text(session, "What company do you work for?");
        } else {
            next(); // Skip if we already have this info.
        }
    },
    function (session, results) {
        if (results.response) {
            // Save company name if we asked for it.
            session.dialogData.profile.company = results.response;
        }
        session.endDialogWithResult({ response: session.dialogData.profile });
    }
]);

```

NOTE

This example uses two different data bags to store data: `dialogData` and `userData`. If the bot is distributed across multiple compute nodes, each step of the waterfall could be processed by a different node. For more information about storing bot data, see [Manage state data](#).

End a waterfall

A dialog that is created using a waterfall must be explicitly ended, otherwise the bot will repeat the waterfall indefinitely. You can end a waterfall by using one of the following methods:

- `session.endDialog` : Use this method to end the waterfall if there is no data to return to the calling dialog.
- `session.endDialogWithResult` : Use this method to end the waterfall if there is data to return to the calling dialog. The `response` argument that is returned can be a JSON object or any JavaScript primitive data type. For example:

```

session.endDialogWithResult({
    response: { name: session.dialogData.name, company: session.dialogData.company }
});

```

- `session.endConversation` : Use this method to end the waterfall if the end of the waterfall represents the end of the conversation.

As an alternative to using one of these three methods to end a waterfall, you can attach the `endConversationAction` trigger to the dialog. For example:

```
bot.dialog('dinnerOrder', [
    //...waterfall steps...
    // Last step
    function(session, results){
        if(results.response){
            session.dialogData.room = results.response;
            var msg = `Thank you. Your order will be delivered to room ##${session.dialogData.room}`;
            session.endConversation(msg);
        }
    }
])
.endConversationAction(
    "endOrderDinner", "Ok. Goodbye.",
    {
        matches: /^cancel$|^goodbye$/i,
        confirmPrompt: "This will cancel your order. Are you sure?"
    }
);
```

Next steps

Using waterfall, you can collect information from the user with *prompts*. Let's dive into how you can prompt user for input.

[Prompt user for input](#)

Prompt for user input

10/12/2017 • 5 min to read • [Edit Online](#)

The Bot Builder SDK for Node.js provides a set of built-in prompts to simplify collecting inputs from a user.

A *prompt* is used whenever a bot needs input from the user. You can use prompts to ask a user for a series of inputs by chaining the prompts in a waterfall. You can use prompts in conjunction with [waterfall](#) to help you [manage conversation flow](#) in your bot.

This article will help you understand how prompts work and how you can use them to collect information from users.

Prompts and responses

Whenever you need input from a user, you can send a prompt, wait for the user to respond with input, and then process the input and send a response to the user.

The following code sample prompts the user for their name and responds with a greeting message.

```
bot.dialog('greetings', [
  // Step 1
  function (session) {
    builder.Prompts.text(session, 'Hi! What is your name?');
  },
  // Step 2
  function (session, results) {
    session.endDialog(`Hello ${results.response}`);
  }
]);
```

Using this basic construct, you can model your conversation flow by adding as many prompts and responses as your bot requires.

Prompt results

Built-in prompts are implemented as [dialogs](#) that return the user's response in the `results.response` field. For JSON objects, responses are returned in the `results.response.entity` field. Any type of [dialog handler](#) can receive the result of a prompt. Once the bot receives a response, it can consume it or pass it back to the calling dialog by calling the `session.endDialogWithResult` method.

The following code sample shows how to return a prompt result to the calling dialog by using the `session.endDialogWithResult` method. In this example, the `greetings` dialog uses the prompt result that the `askName` dialog returns to greet the user by name.

```
// Ask the user for their name and greet them by name.
bot.dialog('greetings', [
    function (session) {
        session.beginDialog('askName');
    },
    function (session, results) {
        session.endDialog(`Hello ${results.response}!`);
    }
]);
bot.dialog('askName', [
    function (session) {
        builder.Prompts.text(session, 'Hi! What is your name?');
    },
    function (session, results) {
        session.endDialogWithResult(results);
    }
]);

```

Prompt types

The Bot Builder SDK for Node.js includes several different types of built-in prompts.

PROMPT TYPE	DESCRIPTION
Prompts.text	Asks the user to enter a string of text.
Prompts.confirm	Asks the user to confirm an action.
Prompts.number	Asks the user to enter a number.
Prompts.time	Asks the user for a time or date/time.
Prompts.choice	Asks the user to choose from a list of options.
Prompts.attachment	Asks the user to upload a picture or video.

The following sections provide additional details about each type of prompt.

Prompts.text

Use the [Prompts.text\(\)](#) method to ask the user for a **string of text**. The prompt returns the user's response as an [IPromptTextResult](#).

```
builder.Prompts.text(session, "What is your name?");
```

Prompts.confirm

Use the [Prompts.confirm\(\)](#) method to ask the user to confirm an action with a **yes/no** response. The prompt returns the user's response as an [IPromptConfirmResult](#).

```
builder.Prompts.confirm(session, "Are you sure you wish to cancel your order?");
```

Prompts.number

Use the [Prompts.number\(\)](#) method to ask the user for a **number**. The prompt returns the user's response as an [IPromptNumberResult](#).

```
builder.Prompts.number(session, "How many would you like to order?");
```

Prompts.time

Use the [Prompts.time\(\)](#) method to ask the user for a **time** or **date/time**. The prompt returns the user's response as an [IPromptTimeResult](#). The framework uses the [Chrono](#) library to parse the user's response and supports both relative responses (e.g., "in 5 minutes") and non-relative responses (e.g., "June 6th at 2pm").

The [results.response](#) field, which represents the user's response, contains an [entity](#) object that specifies the date and time. To resolve the date and time into a JavaScript [Date](#) object, use the [EntityRecognizer.resolveTime\(\)](#) method.

TIP

The time that the user enters is converted to UTC time based upon the time zone of the server that hosts the bot. Since the server may be located in a different time zone than the user, be sure to take time zones into consideration. To convert date and time to the user's local time, consider asking the user what time zone they are in.

```
bot.dialog('createAlarm', [
    function (session) {
        session.dialogData.alarm = {};
        builder.Prompts.text(session, "What would you like to name this alarm?");
    },
    function (session, results, next) {
        if (results.response) {
            session.dialogData.name = results.response;
            builder.Prompts.time(session, "What time would you like to set an alarm for?");
        } else {
            next();
        }
    },
    function (session, results) {
        if (results.response) {
            session.dialogData.time = builder.EntityRecognizer.resolveTime([results.response]);
        }

        // Return alarm to caller
        if (session.dialogData.name && session.dialogData.time) {
            session.endDialogWithResult({
                response: { name: session.dialogData.name, time: session.dialogData.time }
            });
        } else {
            session.endDialogWithResult({
                resumed: builder.ResumeReason.notCompleted
            });
        }
    }
]);
```

Prompts.choice

Use the [Prompts.choice\(\)](#) method to ask the user to **choose from a list of options**. The user can convey their selection either by entering the number associated with the option that they choose or by entering the name of the option that they choose. Both full and partial matches of the option's name are supported. The prompt returns the user's response as an [IPromptChoiceResult](#).

To specify the style of the list that is presented to the user, set the [IPromptOptions.listStyle](#) property. The following table shows the [ListStyle](#) enumeration values for this property.

The [ListStyle](#) enum values are as follows:

INDEX	NAME	DESCRIPTION
0	none	No list is rendered. This is used when the list is included as part of the prompt.
1	inline	Choices are rendered as an inline list of the form "1. red, 2. green, or 3. blue".
2	list	Choices are rendered as a numbered list.
3	button	Choices are rendered as buttons for channels that support buttons. For other channels they will be rendered as text.
4	auto	The style is selected automatically based on the channel and number of options.

You can access this enum from the `builder` object or you can provide an index to choose a `ListStyle`. For example, both statements in the following code snippet accomplish the same thing.

```
// ListStyle passed in as Enum
builder.Prompts.choice(session, "Which color?", "red|green|blue", { listStyle: builder.ListStyle.button });

// ListStyle passed in as index
builder.Prompts.choice(session, "Which color?", "red|green|blue", { listStyle: 3 });
```

To specify the list of options, you can use a pipe-delimited (`|`) string, an array of strings, or an object map.

A pipe-delimited string:

```
builder.Prompts.choice(session, "Which color?", "red|green|blue");
```

An array of strings:

```
builder.Prompts.choice(session, "Which color?", ["red", "green", "blue"]);
```

An object map:

```

var salesData = {
    "west": {
        units: 200,
        total: "$6,000"
    },
    "central": {
        units: 100,
        total: "$3,000"
    },
    "east": {
        units: 300,
        total: "$9,000"
    }
};

bot.dialog('getSalesData', [
    function (session) {
        builder.Prompts.choice(session, "Which region would you like sales for?", salesData);
    },
    function (session, results) {
        if (results.response) {
            var region = salesData[results.response.entity];
            session.send(`We sold ${region.units} units for a total of ${region.total}.`);
        } else {
            session.send("OK");
        }
    }
]);

```

Prompts.attachment

Use the [Prompts.attachment\(\)](#) method to ask the user to upload a file such an image or video. The prompt returns the user's response as an [IPromptAttachmentResult](#).

```
builder.Prompts.attachment(session, "Upload a picture for me to transform.");
```

Next steps

Now that you know how to step users through a waterfall and prompt them for information, lets take a look at ways to help you better manage the conversation flow.

[Manage conversation flow](#)

Manage conversation flow with dialogs

9/12/2017 • 14 min to read • [Edit Online](#)

Managing conversation flow is an essential task in building bots. A bot needs to be able to perform core tasks elegantly and handle interruptions gracefully. With the Bot Builder SDK for Node.js, you can manage conversation flow using dialogs.

A dialog is like a function in a program. It is generally designed to perform a specific operation and it can be invoked as often as it is needed. You can chain multiple dialogs together to handle just about any conversation flow that you want your bot to handle. The Bot Builder SDK for Node.js includes built-in features such as [prompts](#) and [waterfalls](#) to help you manage conversation flow.

This article provides a series of examples to explain how to manage both simple conversation flows and complex conversation flows where your bot can handle interruptions and resume the flow gracefully using dialogs. The examples are based on the following scenarios:

1. Your bot will take a dinner reservation.
2. Your bot can process "Help" request at any time during the reservation.
3. Your bot can process context-sensitive "Help" for current step of the reservation.
4. Your bot can handle multiple topics of conversation.

Manage conversation flow with a waterfall

A [waterfall](#) is a dialog that allows the bot to easily walk a user through a series of tasks. In this example, the reservation bot asks the user a series of questions so that the bot can process the reservation request. The bot will prompt the user for the following information:

1. Reservation date and time
2. Number of people in the party
3. Name of the person making the reservation

The following code sample shows how to use a waterfall to guide the user through a series of prompts.

```
// This is a dinner reservation bot that uses a waterfall technique to prompt users for input.
var bot = new builder.UniversalBot(connector, [
  function (session) {
    session.send("Welcome to the dinner reservation.");
    builder.Prompts.time(session, "Please provide a reservation date and time (e.g.: June 6th at 5pm)");
  },
  function (session, results) {
    session.dialogData.reservationDate = builder.EntityRecognizer.resolveTime([results.response]);
    builder.Prompts.number(session, "How many people are in your party?");
  },
  function (session, results) {
    session.dialogData.partySize = results.response;
    builder.Prompts.text(session, "Whose name will this reservation be under?");
  },
  function (session, results) {
    session.dialogData.reservationName = results.response;

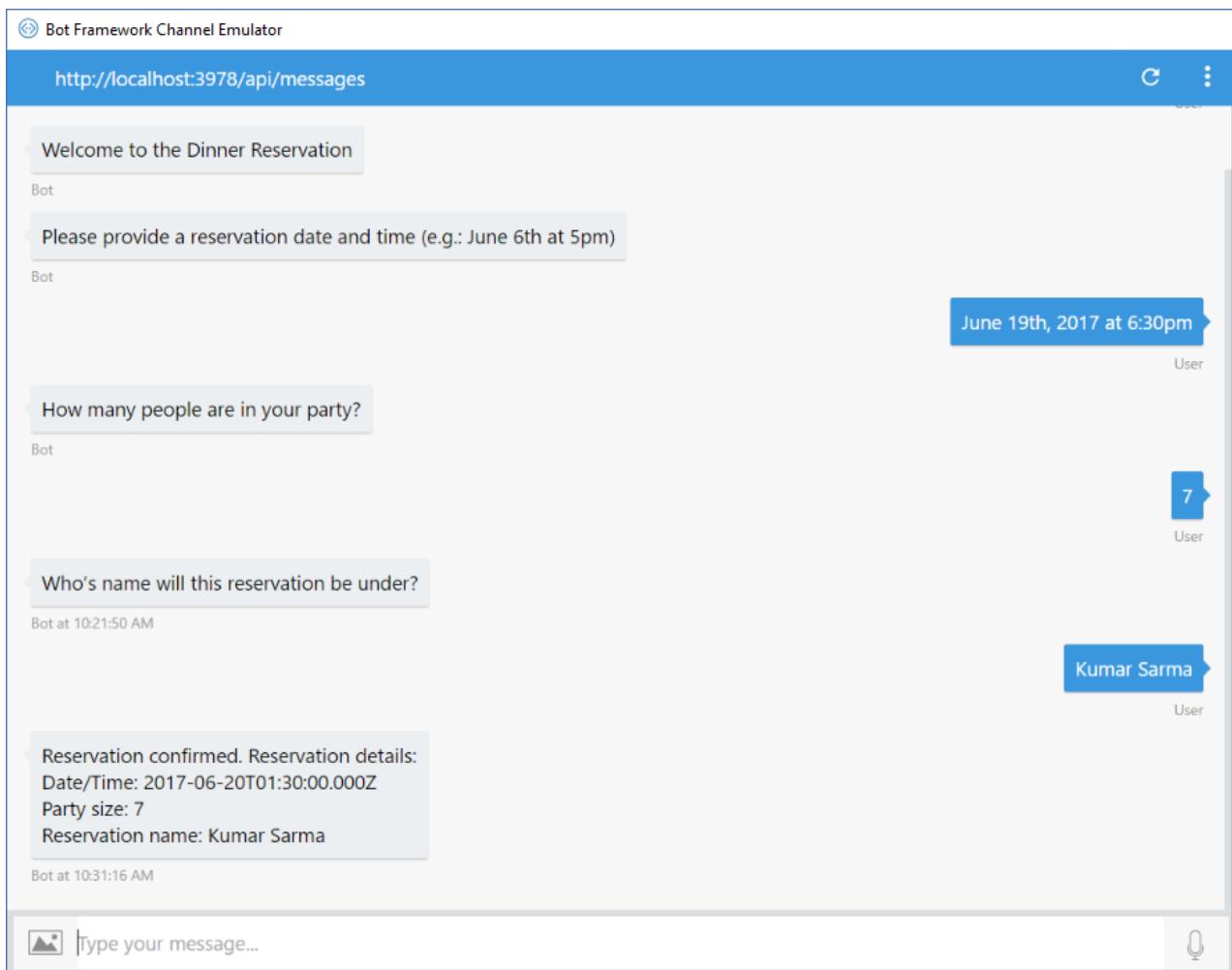
    // Process request and display reservation details
    session.send(`Reservation confirmed. Reservation details: <br/>Date/Time:
${session.dialogData.reservationDate} <br/>Party size: ${session.dialogData.partySize} <br/>Reservation name:
${session.dialogData.reservationName}`);
    session.endDialog();
  }
]);
]);
```

The core functionality of this bot occurs in the default dialog. The default dialog is defined when the bot is created:

```
var bot = new builder.UniversalBot(connector, [...waterfall steps...]);
```

The default dialog is created as an array of functions that define the steps of the waterfall. In the example, there are four functions so the waterfall has four steps. Each step performs a single task and the results are processed in the next step. The process continues until the last step, where the reservation is confirmed and the dialog ends.

The following screen shot shows the results of this bot running in the [Bot Framework Emulator](#):



Prompt user for input

Each step of this example uses a prompt to ask the user for input. A prompt is a special type of dialog that asks for user input, waits for a response, and returns the response to the next step in the waterfall. See [Prompt users for input](#) for information about the many different types of prompts you can use in your bot.

In this example, the bot uses `Prompts.text()` to solicit a freeform response from the user in text format. The user can respond with any text and the bot must decide how to handle the response. `Prompts.time()` uses the [Chrono](#) library to parse for date and time information from a string. This allows your bot to understand more natural language for specifying date and time. For example: "June 6th, 2017 at 9pm", "Today at 7:30pm", "next monday at 6pm", and so on.

TIP

The time that the user enters is converted to UTC time based upon the time zone of the server that hosts the bot. Since the server may be located in a different time zone than the user, be sure to take time zones into consideration. To convert date and time to the user's local time, consider asking the user what time zone they are in.

Manage a conversation flow with multiple dialogs

Another technique for managing conversation flow is to use a combination of waterfall and multiple dialogs. The waterfall allows you to chain functions together in a dialog, while dialogs enable you to break a conversation into smaller pieces of functionality that can be reused at any time.

For example, consider the dinner reservation bot. The following code sample shows the previous example rewritten to use waterfall and multiple dialogs.

```

// This is a dinner reservation bot that uses multiple dialogs to prompt users for input.
var bot = new builder.UniversalBot(connector, [
    function (session) {
        session.send("Welcome to the dinner reservation.");
        session.beginDialog('askForDateTime');
    },
    function (session, results) {
        session.dialogData.reservationDate = builder.EntityRecognizer.resolveTime([results.response]);
        session.beginDialog('askForPartySize');
    },
    function (session, results) {
        session.dialogData.partySize = results.response;
        session.beginDialog('askForReserverName');
    },
    function (session, results) {
        session.dialogData.reservationName = results.response;

        // Process request and display reservation details
        session.send(`Reservation confirmed. Reservation details: <br/>Date/Time:
${session.dialogData.reservationDate} <br/>Party size: ${session.dialogData.partySize} <br/>Reservation name:
${session.dialogData.reservationName}`);
        session.endDialog();
    }
]);

// Dialog to ask for a date and time
bot.dialog('askForDateTime', [
    function (session) {
        builder.Prompts.time(session, "Please provide a reservation date and time (e.g.: June 6th at 5pm)");
    },
    function (session, results) {
        session.endDialogWithResult(results);
    }
]);

// Dialog to ask for number of people in the party
bot.dialog('askForPartySize', [
    function (session) {
        builder.Prompts.text(session, "How many people are in your party?");
    },
    function (session, results) {
        session.endDialogWithResult(results);
    }
]);

// Dialog to ask for the reservation name.
bot.dialog('askForReserverName', [
    function (session) {
        builder.Prompts.text(session, "Who's name will this reservation be under?");
    },
    function (session, results) {
        session.endDialogWithResult(results);
    }
]);

```

The results from executing this bot are exactly the same as the previous bot where only waterfall is used. However, programmatically, there are two primary differences:

1. The default dialog is dedicated to managing the flow of the conversation.
2. The task for each step of the conversation is being managed by a separate dialog. In this case, the bot needed three pieces of information so it prompts the user three times. Each prompt is now contained within its own dialog.

With this technique, you can separate the conversation flow from the task logic. This allows the dialogs to be

reused by different conversation flows if necessary.

Respond to user input

In the process of guiding the user through a series of tasks, if the user has questions or wants to request additional information before answering, how would you handle those requests? For example, regardless of where the user is in the conversation, how would the bot respond if the user enters "Help", "Support" or "Cancel"? What if the user wants additional information about a step? What happens if the user changes their mind and wants to abandon the current task to start a completely different task?

The Bot Builder SDK for Node.js allows a bot to listen for certain input within a global context or within a local context in the scope of the current dialog. These inputs are called [actions](#), which allow the bot to listen for user input based on a `matches` clause. It's up to the bot to decide how to react to specific user inputs.

Handle global action

If you want your bot to be able to handle actions at any point in a conversation, use `triggerAction`. Triggers allow the bot to invoke a specific dialog when the input matches the specified term. For example, if you want to support a global "Help" option, you can create a help dialog and attach a `triggerAction` that listens for input matching "Help".

The following code sample shows how to attach a `triggerAction` to a dialog to specify that the dialog should be invoked when the user enters "help".

```
// The dialog stack is cleared and this dialog is invoked when the user enters 'help'.
bot.dialog('help', function (session, args, next) {
    session.endDialog("This is a bot that can help you make a dinner reservation. <br/>Please say 'next' to
    continue");
})
.triggerAction({
    matches: /^help$/i,
});
```

By default, when a `triggerAction` executes, the dialog stack is cleared and the triggered dialog becomes the new default dialog. In this example, when the `triggerAction` executes, the dialog stack is cleared and the `help` dialog is then added to the stack as the new default dialog. If this is not the desired behavior, you can add the `onSelectAction` option to the `triggerAction`. The `onSelectAction` option allows the bot to start a new dialog without clearing the dialog stack, which enables the conversation to be temporarily redirected and later resume where it left off.

The following code sample shows how to use the `onSelectAction` option with `triggerAction` so that the `help` dialog is added to the existing dialog stack (and the dialog stack is not cleared).

```
bot.dialog('help', function (session, args, next) {
    session.endDialog("This is a bot that can help you make a dinner reservation. <br/>Please say 'next' to
    continue");
})
.triggerAction({
    matches: /^help$/i,
    onSelectAction: (session, args, next) => {
        // Add the help dialog to the dialog stack
        // (override the default behavior of replacing the stack)
        session.beginDialog(args.action, args);
    }
});
```

In this example, the `help` dialog takes control of the conversation when the user enters "help". Because the `triggerAction` contains the `onSelectAction` option, the `help` dialog is pushed onto the existing dialog stack and

the stack is not cleared. When the `help` dialog ends, it is removed from the dialog stack and the conversation resumes from the point at which it was interrupted with the `help` command.

Handle contextual action

In the previous example, the `help` dialog is invoked if the user enters "help" at any point in the conversation. As such, the dialog can only offer general help guidance, as there is no specific context for the user's help request. But, what if the user wants to request help regarding a specific point in the conversation? In that case, the `help` dialog must be triggered within the context of the current dialog.

For example, consider the dinner reservation bot. What if a user wants to know the maximum party size when they are asked for the number of members in their party? To handle this scenario, you can attach a `beginDialogAction` to the `askForPartySize` dialog, listening for the user input "help".

The following code sample shows how to attach context-sensitive help to a dialog using `beginDialogAction`.

```
// Dialog to ask for number of people in the party
bot.dialog('askForPartySize', [
  function (session) {
    builder.Prompts.text(session, "How many people are in your party?");
  },
  function (session, results) {
    session.endDialogWithResult(results);
  }
])
.beginDialogAction('partySizeHelpAction', 'partySizeHelp', { matches: /^help$/i });

// Context Help dialog for party size
bot.dialog('partySizeHelp', function(session, args, next) {
  var msg = "Party size help: Our restaurant can support party sizes up to 150 members.";
  session.endDialog(msg);
})
```

In this example, whenever the user enters "help", the bot will push the `partySizeHelp` dialog onto the stack. That dialog sends a help message to the user and then ends the dialog, returning control back to the `askForPartySize` dialog which reprompts the user for a party size.

It is important to note that this context-sensitive help is only executed when the user is in the `askForPartySize` dialog. Otherwise, the general help message from the `triggerAction` will execute instead. In other words, the local `matches` clause always takes precedence over the global `matches` clause. For example, if a `beginDialogAction` matches for `help`, then the matches for `help` in the `triggerAction` will not be executed. For more information, see [Action precedence](#).

Change the topic of conversation

By default, executing a `triggerAction` clears the dialog stack and resets the conversation, starting with the specified dialog. This behavior is often preferable when your bot needs to switch from one topic of conversation to another, such as if a user in the midst of booking a dinner reservation instead decides to order dinner to be delivered to their hotel room.

The following example builds upon the previous one such that the bot allows the user to either make a dinner reservation or order dinner to be delivered. In this bot, the default dialog is a greeting dialog that presents two options to the user: `Dinner Reservation` and `Order Dinner`.

```
// This bot enables users to either make a dinner reservation or order dinner.
var bot = new builder.UniversalBot(connector, function(session){
  var msg = "Welcome to the reservation bot. Please say `Dinner Reservation` or `Order Dinner`";
  session.send(msg);
});
```

The dinner reservation logic that was in the default dialog of the previous example is now in its own dialog called `dinnerReservation`. The flow of the `dinnerReservation` remains the same as the multiple dialog version discussed earlier. The only difference is that the dialog has a `triggerAction` attached to it. Notice that in this version, the `confirmPrompt` asks the user to confirm that they want to change the topic of conversation, before invoking the new dialog. It is good practice to include a `confirmPrompt` in scenarios like this because once the dialog stack is cleared, the user will be directed to the new topic of conversation, thereby abandoning the topic of conversation that was underway.

```
// This dialog helps the user make a dinner reservation.
bot.dialog('dinnerReservation', [
    function (session) {
        session.send("Welcome to the dinner reservation.");
        session.beginDialog('askForDateTime');
    },
    function (session, results) {
        session.dialogData.reservationDate = builder.EntityRecognizer.resolveTime([results.response]);
        session.beginDialog('askForPartySize');
    },
    function (session, results) {
        session.dialogData.partySize = results.response;
        session.beginDialog('askForReserverName');
    },
    function (session, results) {
        session.dialogData.reservationName = results.response;

        // Process request and display reservation details
        session.send(`Reservation confirmed. Reservation details: <br/>Date/Time:
${session.dialogData.reservationDate} <br/>Party size: ${session.dialogData.partySize} <br/>Reservation name:
${session.dialogData.reservationName}`);
        session.endDialog();
    }
])
.triggerAction({
    matches: /^dinner reservation$/i,
    confirmPrompt: "This will cancel your current request. Are you sure?"
});
```

The second topic of conversation is defined in the `orderDinner` dialog using a waterfall. This dialog simply displays the dinner menu and prompts the user for a room number after the order is specified. A `triggerAction` is attached to the dialog to specify that it should be invoked when the user enters "order dinner" and to ensure that the user is prompted to confirm their selection, should they indicate a desire to change the topic of conversation.

```

// This dialog help the user order dinner to be delivered to their hotel room.
var dinnerMenu = {
    "Potato Salad - $5.99": {
        Description: "Potato Salad",
        Price: 5.99
    },
    "Tuna Sandwich - $6.89": {
        Description: "Tuna Sandwich",
        Price: 6.89
    },
    "Clam Chowder - $4.50": {
        Description: "Clam Chowder",
        Price: 4.50
    }
};

bot.dialog('orderDinner', [
    function(session){
        session.send("Lets order some dinner!");
        builder.Prompts.choice(session, "Dinner menu:", dinnerMenu);
    },
    function (session, results) {
        if (results.response) {
            var order = dinnerMenu[results.response.entity];
            var msg = `You ordered: ${order.Description} for a total of $$${order.Price}.`;
            session.dialogData.order = order;
            session.send(msg);
            builder.Prompts.text(session, "What is your room number?");
        }
    },
    function(session, results){
        if(results.response){
            session.dialogData.room = results.response;
            var msg = `Thank you. Your order will be delivered to room ##${session.dialogData.room}`;
            session.endDialog(msg);
        }
    }
])
.triggerAction({
    matches: /^order dinner$/i,
    confirmPrompt: "This will cancel your order. Are you sure?"
});

```

After the user starts a conversation and selects either `Dinner Reservation` or `Order Dinner`, they may change their mind at any time. For example, if the user is in the middle of making a dinner reservation and enters "order dinner," the bot will confirm by saying, "This will cancel your current request. Are you sure?". If the user types "no," then the request is canceled and the user can continue with the dinner reservation process. If the user types "yes," the bot will clear the dialog stack and transfer control of the conversation to the `orderDinner` dialog.

End conversation

In the examples above, dialogs are closed by either using `session.endDialog` or `session.endDialogWithResult`, both of which end the dialog, remove it from the stack, and return control to the calling dialog. In situations where the user has reached the end of the conversation, you should use `session.endConversation` to indicate that the conversation is finished.

The `session.endConversation` method ends a conversation and optionally sends a message to the user. For example, the `orderDinner` dialog in the previous example could end the conversation by using `session.endConversation`, as shown in the following code sample.

```

bot.dialog('orderDinner', [
    //...waterfall steps...
    // Last step
    function(session, results){
        if(results.response){
            session.dialogData.room = results.response;
            var msg = `Thank you. Your order will be delivered to room #${session.dialogData.room}`;
            session.endConversation(msg);
        }
    }
]);

```

Calling `session.endConversation` will end the conversation by clearing the dialog stack and resetting the `session.conversationData` storage. For more information on data storage, see [Manage state data](#).

Calling `session.endConversation` is a logical thing to do when the user completes the conversation flow for which the bot is designed. You may also use `session.endConversation` to end the conversation in situations where the user enters "cancel" or "goodbye" in the midst of a conversation. To do so, simply attach an `endConversationAction` to the dialog and have this trigger listen for input matching "cancel" or "goodbye".

The following code sample shows how to attach an `endConversationAction` to a dialog to end the conversation if the user enters "cancel" or "goodbye".

```

bot.dialog('dinnerOrder', [
    //...waterfall steps...
])
.endConversationAction(
    "endOrderDinner", "Ok. Goodbye.",
    {
        matches: /^cancel$|^goodbye$/i,
        confirmPrompt: "This will cancel your order. Are you sure?"
    }
);

```

Since ending a conversation with `session.endConversation` or `endConversationAction` will clear the dialog stack and force the user to start over, you should include a `confirmPrompt` to ensure that the user really wants to do so.

Next steps

In this article, you explore ways to manage conversations that are sequential in nature. What if you want to repeat a dialog or use looping pattern in your conversation? Let's see how you can do that by replacing dialogs on the stack.

[Replace dialogs](#)

Replace dialogs

9/6/2017 • 10 min to read • [Edit Online](#)

The ability to replace a dialog can be useful when you need to validate user input or repeat an action during the course of a conversation. With the Bot Builder SDK for Node.js, you can replace a dialog by using the `session.replaceDialog` method. This method enables you to end the current dialog and replace it with a new dialog without returning to the caller.

Create custom prompts to validate input

The Bot Builder SDK for Node.js includes input validation for some types of `prompts` such as `Prompts.time` and `Prompts.choice`. To validate text input that you receive in response to `Prompts.text`, you must create your own validation logic and custom prompts.

You may want to validate an input if the input must comply with a certain value, pattern, range, or criteria that you define. If an input fails validation, the bot can prompt the user for that information again by using the `session.replaceDialog` method.

The following code sample shows how to create a custom prompt to validate user input for a phone number.

```
// This dialog prompts the user for a phone number.  
// It will re-prompt the user if the input does not match a pattern for phone number.  
bot.dialog('phonePrompt', [  
    function (session, args) {  
        if (args && args.reprompt) {  
            builder.Prompts.text(session, "Enter the number using a format of either: '(555) 123-4567' or '555-  
123-4567' or '5551234567'")  
        } else {  
            builder.Prompts.text(session, "What's your phone number?");  
        }  
    },  
    function (session, results) {  
        var matched = results.response.match(/\d+/g);  
        var number = matched ? matched.join('') : '';  
        if (number.length == 10 || number.length == 11) {  
            session.userData.phoneNumber = number; // Save the number.  
            session.endDialogWithResult({ response: number });  
        } else {  
            // Repeat the dialog  
            session.replaceDialog('phonePrompt', { reprompt: true });  
        }  
    }  
]);
```

In this example, the user is initially prompted to provide their phone number. The validation logic uses a regular expression that matches a range of digits from the user input. If the input contains 10 or 11 digits, then that number is returned in the response. Otherwise, the `session.replaceDialog` method is executed to repeat the `phonePrompt` dialog, which prompts the user for input again, this time providing more specific guidance regarding the format of input that is expected.

When you call the `session.replaceDialog` method, you specify the name of the dialog to repeat and an arguments list. In this example, the arguments list contains `{ reprompt: true }`, which enables the bot to provide different prompt messages depending on whether it is an initial prompt or a reprompt, but you can specify whatever arguments your bot may require.

Repeat an action

There may be times in the course of a conversation where you want to repeat a dialog to enable the user to complete a certain action multiple times. For example, if your bot offers a variety of services, you might initially display the menu of services, walk the user through the process of requesting a service, and then display the menu of services again, thereby enabling the user to request another service. To achieve this, you can use the `session.replaceDialog` method to display the menu of services again, rather than ending the conversation with the `'session.endConversation'` method.

The following example shows how to use the `session.replaceDialog` method to implement this type of scenario.

First, the menu of services is defined:

```
// Main menu
var menuItems = {
    "Order dinner": {
        item: "orderDinner"
    },
    "Dinner reservation": {
        item: "dinnerReservation"
    },
    "Schedule shuttle": {
        item: "scheduleShuttle"
    },
    "Request wake-up call": {
        item: "wakeUpCall"
    }
}
```

The `mainMenu` dialog is invoked by the default dialog, so the menu will be presented to the user at the beginning of the conversation. Additionally, a `triggerAction` is attached to the `mainMenu` dialog so that the menu will also be presented any time the user input is "main menu". When the user is presented with the menu and selects an option, the dialog that corresponds to the user's selection is invoked by using the `session.beginDialog` method.

```
// This is a reservation bot that has a menu of offerings.
var bot = new builder.UniversalBot(connector, [
    function(session){
        session.send("Welcome to Contoso Hotel and Resort.");
        session.beginDialog("mainMenu");
    }
]);

// Display the main menu and start a new request depending on user input.
bot.dialog("mainMenu", [
    function(session){
        builder.Prompts.choice(session, "Main Menu:", menuItems);
    },
    function(session, results){
        if(results.response){
            session.beginDialog(menuItems[results.response.entity].item);
        }
    }
])
.triggerAction({
    // The user can request this at any time.
    // Once triggered, it clears the stack and prompts the main menu again.
    matches: /^main menu$/i,
    confirmPrompt: "This will cancel your request. Are you sure?"
});
```

In this example, if the user chooses option 1 to order dinner to be delivered to their room, the `orderDinner` dialog will be invoked and will walk the user through the process of ordering dinner. At the end of the process, the bot will

confirm the order and display the main menu again by using the `session.replaceDialog` method.

```
// Menu: "Order dinner"
// This dialog allows user to order dinner to be delivered to their hotel room.
bot.dialog('orderDinner', [
    function(session){
        session.send("Lets order some dinner!");
        builder.Prompts.choice(session, "Dinner menu:", dinnerMenu);
    },
    function (session, results) {
        if (results.response) {
            var order = dinnerMenu[results.response.entity];
            var msg = `You ordered: %(Description)s for a total of $$\{order.Price}\.`;
            session.dialogData.order = order;
            session.send(msg);
            builder.Prompts.text(session, "What is your room number?");
        }
    },
    function(session, results){
        if(results.response){
            session.dialogData.room = results.response;
            var msg = `Thank you. Your order will be delivered to room #\$\{results.response}\.`;
            session.send(msg);
            session.replaceDialog("mainMenu"); // Display the menu again.
        }
    }
])
.reloadAction(
    "restartOrderDinner", "Ok. Let's start over.",
    {
        matches: /^start over$/i,
        confirmPrompt: "This wil cancel your order. Are you sure?"
    }
)
.cancelAction(
    "cancelOrder", "Type 'Main Menu' to continue.",
    {
        matches: /^cancel$/i,
        confirmPrompt: "This will cancel your order. Are you sure?"
    }
);
});
```

Two triggers are attached to the `orderDinner` dialog to enable the user to "start over" or "cancel" the order at any time during the ordering process.

The first trigger is `reloadAction`, which allows the user to start the order process over again by sending the input "start over". When the trigger matches the utterance "start over", the `reloadAction` restarts the dialog from the beginning.

The second trigger is `cancelAction`, which allows the user to abort the order process completely by sending the input "cancel". This trigger does not automatically display the main menu again, but instead sends a message that tells the user what to do next by saying "Type 'Main Menu' to continue."

Dialog loops

In the example above, the user can only select one item per order. That is, if the user wanted to order two items from the menu, they would have to complete the entire ordering process for the first item and then repeat the entire ordering process again for the second item.

The following example shows how to improve upon the previous bot by refactoring the dinner menu into a separate dialog. Doing so enables the bot to repeat the dinner menu in a loop and therefore allows the user to select multiple items within a single order.

First, add a "Check out" option to the menu. This option will allow the user to exit the item selection process and continue with the check out process.

```
// The dinner menu
var dinnerMenu = {
    //...other menu items...
    "Check out": {
        Description: "Check out",
        Price: 0 // Order total. Updated as items are added to order.
    }
};
```

Next, refactor the order prompt into its own dialog so that the bot can repeat the menu and allow user to add multiple items to their order.

```
// Add dinner items to the list by repeating this dialog until the user says `check out`.
bot.dialog("addDinnerItem", [
    function(session, args){
        if(args && args.reprompt){
            session.send("What else would you like to have for dinner tonight?");
        }
        else{
            // New order
            // Using the conversationData to store the orders
            session.conversationData.orders = new Array();
            session.conversationData.orders.push({
                Description: "Check out",
                Price: 0
            })
        }
        builder.Prompts.choice(session, "Dinner menu:", dinnerMenu);
    },
    function(session, results){
        if(results.response){
            if(results.response.entity.match(/^check out$/i)){
                session.endDialog("Checking out...");
            }
            else {
                var order = dinnerMenu[results.response.entity];
                session.conversationData.orders[0].Price += order.Price; // Add to total.
                var msg = `You ordered: ${order.Description} for a total of ${order.Price}.`;
                session.send(msg);
                session.conversationData.orders.push(order);
                session.replaceDialog("addDinnerItem", { reprompt: true }); // Repeat dinner menu
            }
        }
    }
])
.reloadAction(
    "restartOrderDinner", "Ok. Let's start over.",
{
    matches: /^start over$/i,
    confirmPrompt: "This will cancel your order. Are you sure?"
}
);
```

In this example, orders are stored in a bot data store that is scoped to the current conversation, `session.conversationData.orders`. For every new order, the variable is re-initialized with a new array and for every item the user chooses, the bot adds that item to the `orders` array and adds the price to the total, which is stored in the checkout's `Price` variable. When the user finishes selecting items for their order, they can say "check out" and then continue with the remainder of the order process.

NOTE

For more information about bot data storage, see [Manage state data](#).

Finally, update the second step of the waterfall within the `orderDinner` dialog to process the order with confirmation.

```
// Menu: "Order dinner"
// This dialog allows user to order dinner and have it delivered to their room.
bot.dialog('orderDinner', [
    function(session){
        session.send("Lets order some dinner!");
        session.beginDialog("addDinnerItem");
    },
    function (session, results) {
        if (results.response) {
            // Display itemize order with price total.
            for(var i = 1; i < session.conversationData.orders.length; i++){
                session.send(`You ordered: ${session.conversationData.orders[i].Description} for a total of
${session.conversationData.orders[i].Price}.`);
            }
            session.send(`Your total is: ${session.conversationData.orders[0].Price}`);
        }

        // Continue with the check out process.
        builder.Prompts.text(session, "What is your room number?");
    }
}, 
function(session, results){
    if(results.response){
        session.dialogData.room = results.response;
        var msg = `Thank you. Your order will be delivered to room #${results.response}`;
        session.send(msg);
        session.replaceDialog("mainMenu");
    }
}
])
//...attached triggers...
;
```

Cancel a dialog

While the `session.replaceDialog` method can be used to replace the *current* dialog with a new one, it cannot be used to replace a dialog that is located further down the dialog stack. To replace a dialog within the dialog stack that is not the *current* dialog, use the `session.cancelDialog` method instead.

The `session.cancelDialog` method can be used to end a dialog regardless of where it exists in the dialog stack and optionally invoke a new dialog in its place. To call the `session.cancelDialog` method, specify the ID of the dialog to cancel and optionally, specify the ID of the dialog to invoke in its place. For example, this code snippet cancels the `orderDinner` dialog and replaces it with the `mainMenu` dialog:

```
session.cancelDialog('orderDinner', 'mainMenu');
```

When the `session.cancelDialog` method is called, the dialog stack will be searched backwards and the first occurrence of that dialog will be canceled, causing that dialog and its child dialogs (if any) to be removed from the dialog stack. Control will then be returned to the calling dialog, which can check for a `results.resumed` code equal to `ResumeReason.notCompleted` to detect the cancellation.

As an alternative to specifying the ID of the dialog to cancel when you call the `session.cancelDialog` method, you

can instead specify the zero-based index of the dialog to cancel, where the index represents the dialog's position in the dialog stack. For example, the following code snippet terminates the currently active dialog (index = 0) and starts the `mainMenu` dialog in its place. The `mainMenu` dialog is invoked at position 0 of the dialog stack, thereby becoming the new default dialog.

```
session.cancelDialog(0, 'mainMenu');
```

Consider the sample that is discussed in [dialog loops](#) above. When the user reaches the item selection menu, that dialog (`addDinnerItem`) is the fourth dialog in the dialog stack:

[`default dialog, mainMenu, orderDinner, addDinnerItem`]. How can you enable the user to cancel their order from within the `addDinnerItem` dialog? If you attach a `cancelAction` trigger to the `addDinnerItem` dialog, it will only return the user back to the previous dialog (`orderDinner`), which will send the user right back into the `addDinnerItem` dialog.

This is where the `session.cancelDialog` method is useful. Starting with the [dialog loops example](#), add "Cancel order" as an explicit option within the dinner menu.

```
// The dinner menu
var dinnerMenu = {
    //...other menu items...
    "Check out": {
        Description: "Check out",
        Price: 0      // Order total. Updated as items are added to order.
    },
    "Cancel order": { // Cancel the order and back to Main Menu
        Description: "Cancel order",
        Price: 0
    }
};
```

Then, update the `addDinnerItem` dialog to check for a "cancel order" request. If "cancel" is detected, use the `session.cancelDialog` method to cancel the default dialog (i.e., the dialog at index 0 of the stack) and invoke the `mainMenu` dialog in its place.

```
// Add dinner items to the list by repeating this dialog until the user says `check out`.
bot.dialog("addDinnerItem", [
    //...waterfall steps...
    // Last step
    function(session, results){
        if(results.response){
            if(results.response.entity.match(/^check out$/i)){
                session.endDialog("Checking out...");
            }
            else if(results.response.entity.match(/^cancel/i)){
                // Cancel the order and start "mainMenu" dialog.
                session.cancelDialog(0, "mainMenu");
            }
            else {
                //...add item to list and prompt again...
                session.replaceDialog("addDinnerItem", { reprompt: true }); // Repeat dinner menu.
            }
        }
    }
])
//...attached triggers...
;
```

By using the `session.cancelDialog` method in this way, you can implement whatever conversation flow your bot

requires.

Next steps

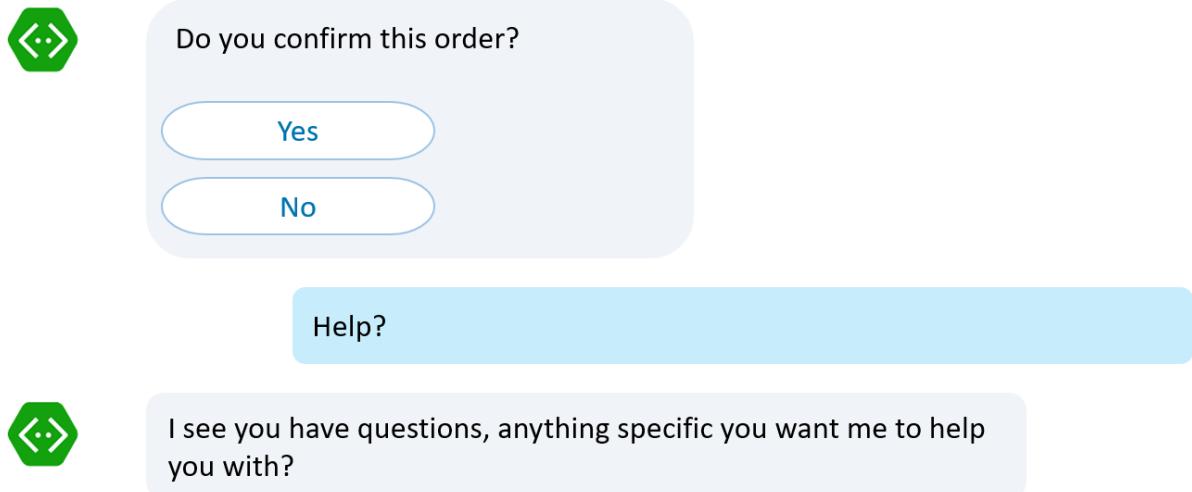
As you can see, to replace dialogs on the stack, you use various types of **actions** to accomplish that task. Actions gives you great flexibilities in managing conversation flow. Let's take a closer look at **actions** and how to better handle user actions.

[Handle user actions](#)

Handle user actions

9/6/2017 • 9 min to read • [Edit Online](#)

Users commonly attempt to access certain functionality within a bot by using keywords like "help", "cancel", or "start over." Users do this in the middle of a conversation, when the bot is expecting a different response. By implementing **actions**, you can design your bot to handle such requests more gracefully. The handlers will examine user input for the keywords that you specify, such as "help", "cancel", or "start over," and respond appropriately.



Action types

The action types you can attach to a dialog are listed in the table below. The link for each action name will take you to a section that provide more details about that action.

ACTION	SCOPE	DESCRIPTION
triggerAction	Global	Binds an action to the dialog that will clear the dialog stack and push itself onto the bottom of stack. Use <code>onSelectAction</code> option to override this default behavior.
customAction	Global	Binds a custom action to the bot that can process information or take action without affecting the dialog stack. Use <code>onSelectAction</code> option to customize the functionality of this action.
beginDialogAction	Contextual	Binds an action to the dialog that starts another dialog when it is triggered. The starting dialog will be pushed onto the stack and popped off once it ends.

ACTION	SCOPE	DESCRIPTION
reloadAction	Contextual	Binds an action to the dialog that causes the dialog to reload when it is triggered. You can use <code>reloadAction</code> to handle user utterances like "start over."
cancelAction	Contextual	Binds an action to the dialog that cancels the dialog when it is triggered. You can use <code>cancelAction</code> to handle user utterances like "cancel" or "nevermind."
endConversationAction	Contextual	Binds an action to the dialog that ends the conversation with the user when triggered. You can use <code>endConversationAction</code> to handle user utterances like "goodbye."

Action precedence

When a bot receives an utterance from a user, it checks the utterance against all the registered actions that are on the dialog stack. The matching starts from the top of the stack down to the bottom of the stack. If nothing matches, then it checks the utterance against the `matches` option of all global actions.

The action precedence is important in cases where you use the same command in different contexts. For example, you can use the "Help" command for the bot as a general help. You can also use "Help" for each of the tasks but these help commands are contextual to each task. For a working sample that elaborates on this point, see [Respond to user input](#).

Bind actions to dialog

Either user utterances or button clicks can *trigger* an action, which is associated with a [dialog](#). If `matches` is specified, the action will listen for the user to say a word or a phrase that triggers the action. The `matches` option can take a regular expression or the name of a [recognizer](#). To bind the action to a button click, use [`CardAction.dialogAction\(\)`](#) to trigger the action.

Actions are *chainable*, which allows you to bind as many actions to a dialog as you want.

Bind a triggerAction

To bind a [triggerAction](#) to a dialog, do the following:

```
// Order dinner.
bot.dialog('orderDinner', [
  //...waterfall steps...
])
// Once triggered, will clear the dialog stack and pushes
// the 'orderDinner' dialog onto the bottom of stack.
.triggerAction({
  matches: /^order dinner$/i
});
```

Binding a `triggerAction` to a dialog registers it to the bot. Once triggered, the `triggerAction` will clear the dialog stack and push the triggered dialog onto the stack. This action is best used to switch between [topic of conversation](#) or to allow users to request arbitrary stand-alone tasks. If you want to override the behavior where

this action clears the dialog stack, add an `onSelectAction` option to the `triggerAction`.

The code snippet below shows how to provide general help from a global context without clearing the dialog stack.

```
bot.dialog('help', function (session, args, next) {
    //Send a help message
    session.endDialog("Global help menu.");
})
// Once triggered, will start a new dialog as specified by
// the 'onSelectAction' option.
.triggerAction({
    matches: /^help$/i,
    onSelectAction: (session, args, next) => {
        // Add the help dialog to the top of the dialog stack
        // (override the default behavior of replacing the stack)
        session.beginDialog(args.action, args);
    }
});
```

In this case, the `triggerAction` is attached to the `help` dialog itself (as opposed to the `orderDinner` dialog). The `onSelectAction` option allows you to start this dialog without clearing the dialog stack. This allows you to handle global requests such as "help", "about", "support", etc. Notice that the `onSelectAction` option explicitly calls the `session.beginDialog` method to start the triggered dialog. The ID of the triggered dialog is provided via the `args.action`. Do not hard code the dialog ID (e.g.: 'help') into this method otherwise you may get runtime errors. If you wish to trigger a contextual help message for the `orderDinner` task itself then consider attaching a `beginDialogAction` to the `orderDinner` dialog instead.

Bind a customAction

Unlike other action types, `customAction` does not have any default action defined. It's up to you to define what this action does. The benefit of using `customAction` is that you have the option to process user requests without manipulating the dialog stack. When a `customAction` is triggered, the `onSelectAction` option can process the request without pushing new dialogs onto the stack. Once the action is completed, control is passed back to the dialog that is at the top of the stack and the bot can continue.

You can use a `customAction` to provide general and quick action requests such as "What is the temperature outside right now?", "What time is it right now in Paris?", "Remind me to buy milk at 5pm today.", etc. These are general actions that the bot can perform without manipulating the stack.

Another key difference with `customAction` is that it binds to the bot as opposed to a dialog.

The following code sample shows how to bind a `customAction` to the `bot` listening for requests to set a reminder.

```
bot.customAction({
    matches: /remind|reminder/gi,
    onSelectAction: (session, args, next) => {
        // Set reminder...
        session.send("Reminder is set.");
    }
})
```

Bind a beginDialogAction

Binding a `beginDialogAction` to a dialog will register the action to the dialog. This method will start another dialog when it is triggered. The behavior of this action is similar to calling the `beginDialog` method. The new dialog is pushed onto the top of the dialog stack so it does not automatically end the current task. The current task is continued once the new dialog ends.

The following code snippet shows how to bind a [beginDialogAction](#) to a dialog.

```
// Order dinner.  
bot.dialog('orderDinner', [  
    //...waterfall steps...  
])  
// Once triggered, will start the 'showDinnerCart' dialog.  
// Then, the waterfall will resume from the step that was interrupted.  
.beginDialogAction('showCartAction', 'showDinnerCart', {  
    matches: /^show cart$/i  
});  
  
// Show dinner items in cart  
bot.dialog('showDinnerCart', function(session){  
    for(var i = 1; i < session.conversationData.orders.length; i++){  
        session.send(`You ordered: ${session.conversationData.orders[i].Description} for a total of  
${session.conversationData.orders[i].Price}.`);  
    }  
  
    // End this dialog  
    session.endDialog(`Your total is: ${session.conversationData.orders[0].Price}`);  
});
```

In cases where you need to pass additional arguments into the new dialog, you can add a [dialogArgs](#) option to the action.

Using the sample above, you can modify it to accept arguments passed in via the [dialogArgs](#).

```
// Order dinner.  
bot.dialog('orderDinner', [  
    //...waterfall steps...  
])  
// Once triggered, will start the 'showDinnerCart' dialog.  
// Then, the waterfall will resume from the step that was interrupted.  
.beginDialogAction('showCartAction', 'showDinnerCart', {  
    matches: /^show cart$/i,  
    dialogArgs: {  
        showTotal: true;  
    }  
});  
  
// Show dinner items in cart with the option to show total or not.  
bot.dialog('showDinnerCart', function(session, args){  
    for(var i = 1; i < session.conversationData.orders.length; i++){  
        session.send(`You ordered: ${session.conversationData.orders[i].Description} for a total of  
${session.conversationData.orders[i].Price}.`);  
    }  
  
    if(args && args.showTotal){  
        // End this dialog with total.  
        session.endDialog(`Your total is: ${session.conversationData.orders[0].Price}`);  
    }  
    else{  
        session.endDialog(); // Ends without a message.  
    }  
});
```

Bind a reloadAction

Binding a [reloadAction](#) to a dialog will register it to the dialog. Binding this action to a dialog causes the dialog to restart when the action is triggered. Triggering this action is similar to calling the [replaceDialog](#) method. This is useful for implementing logic to handle user utterances like "start over" or to create [loops](#).

The following code snippet shows how to bind a [reloadAction](#) to a dialog.

```
// Order dinner.
bot.dialog('orderDinner', [
    //...waterfall steps...
])
// Once triggered, will restart the dialog.
.reloadAction('startOver', 'Ok, starting over.', {
    matches: /^start over$/i
});
```

In cases where you need to pass additional arguments into the reloaded dialog, you can add a `dialogArgs` option to the action. This option is passed into the `args` parameter. Rewriting the sample code above to receive an argument on a reload action will look something like this:

```
// Order dinner.
bot.dialog('orderDinner', [
    function(session, args, next){
        if(args && args.isReloaded){
            // Reload action was triggered.
        }

        session.send("Lets order some dinner!");
        builder.Prompts.choice(session, "Dinner menu:", dinnerMenu);
    }
    //...other waterfall steps...
])
// Once triggered, will restart the dialog.
.reloadAction('startOver', 'Ok, starting over.', {
    matches: /^start over$/i,
    dialogArgs: {
        isReloaded: true;
    }
});
```

Bind a cancelAction

Binding a `cancelAction` will register it to the dialog. Once triggered, this action will abruptly end the dialog. Once the dialog ends, the parent dialog will resume with a resumed code indicating that it was `canceled`. This action allows you to handle utterances such as "nevermind" or "cancel." If you need to programmatically cancel a dialog instead, see [Cancel a dialog](#). For more information on *resumed code*, see [Prompt results](#).

The following code snippet shows how to bind a `cancelAction` to a dialog.

```
// Order dinner.
bot.dialog('orderDinner', [
    //...waterfall steps...
])
//Once triggered, will end the dialog.
.cancelAction('cancelAction', 'Ok, cancel order.', {
    matches: /^nevermind$|^cancel$|^cancel.*order/i
});
```

Bind an endConversationAction

Binding an `endConversationAction` will register it to the dialog. Once triggered, this action ends the conversation with the user. Triggering this action is similar to calling the `endConversation` method. Once a conversation ends, the Bot Builder SDK for Node.js will clear the dialog stack and persisted state data. For more information on persisted state data, see [Manage state data](#).

The following code snippet shows how to bind an `endConversationAction` to a dialog.

```
// Order dinner.  
bot.dialog('orderDinner', [  
    //...waterfall steps...  
])  
// Once triggered, will end the conversation.  
.endConversationAction('endConversationAction', 'Ok, goodbye!', {  
    matches: /^goodbye$/i  
});
```

Confirm interruptions

Most, if not, all of these actions interrupt the normal flow of a conversation. Many are disruptive and should be handled with care. For example, the `triggerAction`, `cancelAction`, or the `endConversationAction` will clear the dialog stack. If the user made the mistake of triggering either of these actions, they will have to start the task over again. To make sure the user really intended to trigger these actions, you can add a `confirmPrompt` option to these actions. The `confirmPrompt` will ask if the user is sure about canceling or ending the current task. It allows the user to change their minds and continue the process.

The code snippet below shows a `cancelAction` with a `confirmPrompt` to make sure the user really wants to cancel the order process.

```
// Order dinner.  
bot.dialog('orderDinner', [  
    //...waterfall steps...  
])  
// Confirm before triggering the action.  
// Once triggered, will end the dialog.  
.cancelAction('cancelAction', 'Ok, cancel order.', {  
    matches: /^nevermind$|^cancel$|^cancel.*order/i  
    confirmPrompt: "Are you sure?"  
});
```

Once this action is triggered, it will ask the user "Are you sure?" The user will have to answer "Yes" to go through with the action or "No" to cancel the action and continue where they were.

Next steps

Actions allow you to anticipate user requests and allow the bot to handle those requests gracefully. Many of these actions are disruptive to the current conversation flow. If you want to allow users the ability to switch out and resume a conversation flow, you need to save the user state before switching out. Let's take a closer look at how to save user state and manage state data.

[Manage state data](#)

Create messages

10/13/2017 • 3 min to read • [Edit Online](#)

Communication between the bot and the user is through messages. Your bot will send message activities to communicate information to users, and likewise, will also receive message activities from users. Some messages may simply consist of plain text, while others may contain richer content such as text to be spoken, suggested actions, media attachments, rich cards, and channel-specific data.

This article describes some of the commonly-used message methods you can use to enhance your user experience.

Default message handler

The Bot Builder SDK for Node.js comes with a default message handler built into the `session` object. This message handler allows you to send and receive text messages between the bot and the user.

Send a text message

Sending a text message using the default message handler is simple, just call `session.send` and pass in a **string**.

This sample shows how you can send a text message to greet the user.

```
session.send("Good morning.");
```

This sample shows how you can send a text message using JavaScript string template.

```
var msg = `You ordered: ${order.Description} for a total of $$ {order.Price}.`;
session.send(msg); //msg: "You ordered: Potato Salad for a total of $5.99."
```

Receive a text message

When a user sends the bot a message, the bot receives the message through the `session.message` property.

This sample shows how to access the user's message.

```
var userMessage = session.message.text;
```

Customizing a message

To have more control over the text formatting of your messages, you can create a custom `message` object and set the properties necessary before sending it to the user.

This sample shows how to create a custom `message` object and set the `text`, `textFormat`, and `textLocale` properties.

```
var customMessage = new builder.Message(session)
    .text("Hello!")
    .textFormat("plain")
    .textLocale("en-us");
session.send(customMessage);
```

In cases where you do not have the `session` object in scope, you can use `bot.send` method to send a formatted message to the user.

The `textFormat` property of a message can be used to specify the format of the text. The `textFormat` property can be set to **plain**, **markdown**, or **xml**. The default value for `textFormat` is **markdown**.

For a list of commonly supported text formatting, see [Text formatting](#). To ensure that the feature(s) you want to use is supported by the target channel, preview the feature(s) using the [Channel Inspector](#).

Message property

The `Message` object has an internal **data** property that it uses to manage the message being sent. Other properties you set are through the different methods this object expose to you.

Message methods

Message properties are set and retrieved through the object's methods. The table below provide a list of the methods you can call to set/get the different **Message** properties.

METHOD	DESCRIPTION
<code>addAttachment(attachment:AttachmentType)</code>	Adds an attachment to a message
<code>addEntity(obj:Object)</code>	Adds an entity to the message.
<code>address(addr:IAddress)</code>	Address routing information for the message. To send user a proactive message, save the message's address in the userData bag.
<code>attachmentLayout(style:string)</code>	Hint for how clients should layout multiple attachments. The default value is 'list'.
<code>attachments(list:AttachmentType)</code>	A list of cards or images to send to the user.
<code>compose(prompts:string[], ...args:any[])</code>	Composes a complex and randomized reply to the user.
<code>entities(list:Object[])</code>	Structured objects passed to the bot or user.
<code>inputHint(hint:string)</code>	Hint sent to user letting them know if the bot is expecting further input or not. The built-in prompts will automatically populate this value for outgoing messages.
<code>localTimeStamp((optional)time:string)</code>	Local time when message was sent (set by client or bot, Ex: 2016-09-23T13:07:49.4714686-07:00.)
<code>originalEvent(event:any)</code>	Message in original/native format of the channel for incoming messages.
<code>sourceEvent(map:ISourceEventMap)</code>	For outgoing messages can be used to pass source specific event data like custom attachments.
<code>speak(ssml:TextType, ...args:any[])</code>	Sets the speak field of the message as <i>Speech Synthesis Markup Language (SSML)</i> . This will be spoken to the user on supported devices.

METHOD	DESCRIPTION
<code>suggestedActions(suggestions:ISuggestedActions IIssuggestedActions)</code>	Optional suggested actions to send to the user. Suggested actions will be displayed only on the channels that support suggested actions.
<code>summary(text:TextType, ...args:any[])</code>	Text to be displayed as fall-back and as short description of the message content in (e.g.: List of recent conversations.)
<code>text(text:TextType, ...args:any[])</code>	Sets the message text.
<code>textFormat(style:string)</code>	Set the text format. Default format is markdown .
<code>textLocale(locale:string)</code>	Set the target language of the message.
<code>toMessage()</code>	Gets the JSON for the message.
<code>composePrompt(session:Session, prompts:string[], args?:any[])</code>	Combines an array of prompts into a single localized prompt and then optionally fills the prompts template slots with the passed in arguments.
<code>randomPrompt(prompts:TextType)</code>	Gets a random prompt from the array of *prompts that is passed in.

Next step

[Send and receive attachments](#)

Send and receive attachments

6/13/2017 • 1 min to read • [Edit Online](#)

A message exchange between user and bot can contain media attachments, such as images, video, audio, and files. The types of attachments that can be sent varies by channel, but these are the basic types:

- **Media and Files:** You can send files like images, audio and video by setting **contentType** to the MIME type of the [IAttachment object](#) and then passing a link to the file in **contentUrl**.
- **Cards:** You can send a rich set of visual cards by setting the **contentType** to the desired card's type and then pass the JSON for the card. If you use one of the rich card builder classes like **HeroCard**, the attachment is automatically filled in for you. See [send a rich card](#) for an example of this.

Add a media attachment

The message object is expected to be an instance of an [IMessage](#) and it's most useful to send the user a message as an object when you'd like to include an attachment like an image. Use the [session.send\(\)](#) method to send messages in the form of a JSON object.

Example

The following example checks to see if the user has sent an attachment, and if they have, it will echo back any image contained in the attachment. You can test this with the Bot Framework Emulator by sending your bot an image.

```
// Create your bot with a function to receive messages from the user
var bot = new builder.UniversalBot(connector, function (session) {
  var msg = session.message;
  if (msg.attachments && msg.attachments.length > 0) {
    // Echo back attachment
    var attachment = msg.attachments[0];
    session.send({
      text: "You sent:",
      attachments: [
        {
          contentType: attachment.contentType,
          contentUrl: attachment.contentUrl,
          name: attachment.name
        }
      ]
    });
  } else {
    // Echo back users text
    session.send("You said: %s", session.message.text);
  }
});
```

Additional resources

- [Preview features with the Channel Inspector](#)
- [IMessage](#)
- [Send a rich card](#)
- [session.send](#)

Send proactive messages

9/6/2017 • 6 min to read • [Edit Online](#)

Typically, each message that a bot sends to the user directly relates to the user's prior input. In some cases, a bot may need to send the user a message that is not directly related to the current topic of conversation. These types of messages are called **proactive messages**.

Proactive messages can be useful in a variety of scenarios. If a bot sets a timer or reminder, it will need to notify the user when the time arrives. Or, if a bot receives a notification from an external system, it may need to communicate that information to the user immediately. For example, if the user has previously asked the bot to monitor the price of a product, the bot will alert the user if it receives notification that the price of the product has dropped by 20%. Or, if a bot requires some time to compile a response to the user's question, it may inform the user of the delay and allow the conversation to continue in the meantime. When the bot finishes compiling the response to the question, it will share that information with the user.

When implementing proactive messages in your bot:

Don't send several proactive messages within a short amount of time. Some channels enforce restrictions on how frequently a bot can send messages to the user, and will disable the bot if it violates those restrictions.

Don't send proactive messages to users who have not previously interacted with the bot or solicited contact with the bot through another means such as e-mail or SMS.

Consider the following scenario:



What city are you travelling to?

London



What is your planned date for the trip?



Good news! Your preferred hotel in Las Vegas is offering a discount! Want to book a trip?

Wait, what?



Sorry, this isn't a valid date for your London trip...

Bot you're drunk...

In this example, the user has previously asked the bot to monitor prices of a hotel in Las Vegas. The bot launched a background monitoring task, which has been running continuously for the past several days. In the current conversation, the user is booking a trip to London when the background task triggers a notification message about a discount for the Las Vegas hotel. The bot interjects

this information into the current conversation, making for a confusing user experience.

How should the bot have handled this situation?

- Wait for the current travel booking to finish, then deliver the notification. This approach would be minimally disruptive, but the delay in communicating the information might cause the user to miss out on the low-price opportunity for the Las Vegas hotel.
- Cancel the current travel booking flow and deliver the notification immediately. This approach delivers the information in a timely fashion but would likely frustrate the user by forcing them start over with their travel booking.
- Interrupt the current booking, clearly change the topic of conversation to the hotel in Las Vegas until the user responds, and then switch back to the in-progress travel booking and continue from where it was interrupted. This approach may seem like the best choice, but it introduces complexity both for the bot developer and the user.

Most commonly, your bot will use some combination of **ad hoc proactive messages** and **dialog-based proactive messages** to handle situations like this.

Types of proactive messages

An **ad hoc proactive message** is the simplest type of proactive message. The bot simply interjects the message into the conversation whenever it is triggered, without any regard for whether the user is currently engaged in a separate topic of conversation with the bot and will not attempt to change the conversation in any way.

A **dialog-based proactive message** is more complex than an ad hoc proactive message. Before it can inject this type of proactive message into the conversation, the bot must identify the context of the existing conversation and decide how (or if) it will resume that conversation after the message interrupts.

For example, consider a bot that needs to initiate a survey at a given point in time. When that time arrives, the bot stops the existing conversation with the user and redirects the user to a `SurveyDialog`. The `SurveyDialog` is added to the top of the dialog stack and takes control of the conversation. When the user finishes all required tasks at the `SurveyDialog`, the `SurveyDialog` closes, returning control to the previous dialog, where the user can continue with the prior topic of conversation.

A dialog-based proactive message is more than just simple notification. In sending the notification, the bot changes the topic of the existing conversation. It then must decide whether to resume that conversation later, or to abandon that conversation altogether by resetting the dialog stack.

Send an ad hoc proactive message

The following code samples show how to send an ad hoc proactive message by using the Bot Builder SDK for Node.js.

To be able to send an ad hoc message to a user, the bot must first collect and save information about the user from the current conversation. The **address** property of the message includes all of the information that the bot will need to send an ad hoc message to the user later.

```
bot.dialog('adhocDialog', function(session, args) {
  var savedAddress = session.message.address;

  // (Save this information somewhere that it can be accessed later, such as in a database, or
  session.userData)
  session.userData.savedAddress = savedAddress;

  var message = 'Hello user, good to meet you! I now know your address and can send you notifications in the
future.';
  session.send(message);
})
```

NOTE

The bot can store the user data in any manner as long as the bot can access it later.

After the bot has collected information about the user, it can send an ad hoc proactive message to the user at any time. To do so, it simply retrieves the user data that it stored previously, constructs the message, and sends it.

```
var bot = new builder.UniversalBot(connector);

function sendProactiveMessage(address) {
  var msg = new builder.Message().address(address);
  msg.text('Hello, this is a notification');
  msg.textLocale('en-US');
  bot.send(msg);
}
```

TIP

An ad hoc proactive message can be initiated like from asynchronous triggers such as http requests, timers, queues or from anywhere else that the developer chooses.

Send a dialog-based proactive message

The following code samples show how to send a dialog-based proactive message by using the Bot Builder SDK for Node.js. You can find the complete working example in the [Microsoft/BotBuilder-Samples/Node/core-proactiveMessages/startNewDialog](#) folder.

To be able to send a dialog-based message to a user, the bot must first collect (and save) information from the current conversation. The `session.message.address` object includes all of the information that the bot will need to send a dialog-based proactive message to the user.

```
// proactiveDialog dialog
bot.dialog('proactiveDialog', function (session, args) {

    savedAddress = session.message.address;

    var message = 'Hey there, I\'m going to interrupt our conversation and start a survey in five seconds...';
    session.send(message);

    message = `You can also make me send a message by accessing:  
http://localhost:${server.address().port}/api/CustomWebApi`;
    session.send(message);

    setTimeout(() => {
        startProactiveDialog(savedAddress);
    }, 5000);
});
```

When it is time to send the message, the bot creates a new dialog and adds it to the top of the dialog stack. The new dialog takes control of the conversation, delivers the proactive message, closes, and then returns control to the previous dialog in the stack.

```
// initiate a dialog proactively
function startProactiveDialog(address) {
    bot.beginDialog(address, "*:survey");
}
```

NOTE

The code sample above requires a custom file, **botadapter.js**, which you can [download from GitHub](#).

The survey dialog controls the conversation until it finishes. Then, it closes (by calling `session.endDialog()`), thereby returning control back to the previous dialog.

```
// handle the proactive initiated dialog
bot.dialog('survey', function (session, args, next) {
    if (session.message.text === "done") {
        session.send("Great, back to the original conversation");
        session.endDialog();
    } else {
        session.send('Hello, I\'m the survey dialog. I\'m interrupting your conversation to ask you a question.  
Type "done" to resume');
    }
});
```

Sample code

For a complete sample that shows how to send proactive messages using the Bot Builder SDK for Node.js, see the [Proactive Messages sample](#) in GitHub. Within the Proactive Messages sample, [simpleSendMessage](#) shows how to send an ad-hoc proactive message and [startNewDialog](#) shows how to send a dialog-based proactive message.

Additional resources

- [Designing conversation flow](#)

Add rich card attachments to messages

9/14/2017 • 6 min to read • [Edit Online](#)

Several channels, like Skype & Facebook, support sending rich graphical cards to users with interactive buttons that the user clicks to initiate an action. The SDK provides several message and card builder classes which can be used to create and send cards. The Bot Framework Connector Service will render these cards using schema native to the channel, supporting cross-platform communication. If the channel does not support cards, such as SMS, the Bot Framework will do its best to render a reasonable experience to users.

Types of rich cards

The Bot Framework currently supports eight types of rich cards:

CARD TYPE	DESCRIPTION
Adaptive Card	A customizable card that can contain any combination of text, speech, images, buttons, and input fields. See per-channel support .
Animation Card	A card that can play animated GIFs or short videos.
Audio Card	A card that can play an audio file.
Hero Card	A card that typically contains a single large image, one or more buttons, and text.
Thumbnail Card	A card that typically contains a single thumbnail image, one or more buttons, and text.
Receipt Card	A card that enables a bot to provide a receipt to the user. It typically contains the list of items to include on the receipt, tax and total information, and other 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.
Video Card	A card that can play videos.

Send a carousel of Hero cards

The following example shows a bot for a fictional t-shirt company and shows how to send a carousel of cards in response to the user saying "show shirts".

```

// Create your bot with a function to receive messages from the user
// Create bot and default message handler
var bot = new builder.UniversalBot(connector, function (session) {
    session.send("Hi... We sell shirts. Say 'show shirts' to see our products.");
});

// Add dialog to return list of shirts available
bot.dialog('showShirts', function (session) {
    var msg = new builder.Message(session);
    msg.attachmentLayout(builder.AttachmentLayout.carousel)
    msg.attachments([
        new builder.HeroCard(session)
            .title("Classic White T-Shirt")
            .subtitle("100% Soft and Luxurious Cotton")
            .text("Price is $25 and carried in sizes (S, M, L, and XL)")
            .images([builder.CardImage.create(session,
                'http://petersapparel.parseapp.com/img/whiteshirt.png')])
        .buttons([
            builder.CardAction.imBack(session, "buy classic white t-shirt", "Buy")
        ]),
        new builder.HeroCard(session)
            .title("Classic Gray T-Shirt")
            .subtitle("100% Soft and Luxurious Cotton")
            .text("Price is $25 and carried in sizes (S, M, L, and XL)")
            .images([builder.CardImage.create(session,
                'http://petersapparel.parseapp.com/img/grayshirt.png')])
        .buttons([
            builder.CardAction.imBack(session, "buy classic gray t-shirt", "Buy")
        ])
    ]);
    session.send(msg).endDialog();
}).triggerAction({ matches: /^(show|list)/i });

```

This example uses the [Message](#) class to build a carousel.

The carousel is comprised of a list of [HeroCard](#) classes that contain an image, text, and a single button that triggers buying the item.

Clicking the **Buy** button triggers sending a message so we need to add a second dialog to catch the button click.

Handle button input

The `buyButtonClick` dialog will be triggered any time a message is received that starts with "buy" or "add" and is followed by something containing the word "shirt". The dialog starts by using a couple of regular expressions to look for the color and optional size shirt that the user asked for. This added flexibility lets you support both button clicks and natural language messages from the user like "please add a large gray shirt to my cart". If the color is valid but the size is unknown, the bot prompts the user to pick a size from a list before adding the item to the cart. Once the bot has all the information it needs, it puts the item onto a cart that's persisted using `session.userData` and then sends the user a confirmation message.

```

// Add dialog to handle 'Buy' button click
bot.dialog('buyButtonClick', [
  function (session, args, next) {
    // Get color and optional size from users utterance
    var utterance = args.intent.matched[0];
    var color = /(white|gray)/i.exec(utterance);
    var size = /\b(Extra Large|Large|Medium|Small)\b/i.exec(utterance);
    if (color) {
      // Initialize cart item
      var item = session.dialogData.item = {
        product: "classic " + color[0].toLowerCase() + " t-shirt",
        size: size ? size[0].toLowerCase() : null,
        price: 25.0,
        qty: 1
      };
      if (!item.size) {
        // Prompt for size
        builder.Prompts.choice(session, "What size would you like?", "Small|Medium|Large|Extra Large");
      } else {
        // Skip to next waterfall step
        next();
      }
    } else {
      // Invalid product
      session.send("I'm sorry... That product wasn't found.").endDialog();
    }
  },
  function (session, results) {
    // Save size if prompted
    var item = session.dialogData.item;
    if (results.response) {
      item.size = results.response.entity.toLowerCase();
    }

    // Add to cart
    if (!session.userData.cart) {
      session.userData.cart = [];
    }
    session.userData.cart.push(item);

    // Send confirmation to users
    session.send("A '%(size)s %(product)s' has been added to your cart.", item).endDialog();
  }
]).triggerAction({ matches: /(buy|add)\s.*shirt/i });

```

Add a message delay for image downloads

Some channels tend to download images before displaying a message to the user so that if you send a message containing an image followed immediately by a message without images you'll sometimes see the messages flipped in the user's feed. To minimize the chance of this you can try to insure that your images are coming from content deliver networks (CDNs) and avoid the use of overly large images. In extreme cases you may even need to insert a 1-2 second delay between the message with the image and the one that follows it. You can make this delay feel a bit more natural to the user by calling **session.sendTyping()** to send a typing indicator before starting your delay.

The Bot Framework implements a batching to try to prevent multiple messages from the bot from being displayed out of order. When your bot sends multiple replies to the user, the individual messages will be automatically grouped into a batch and delivered to the user as a set in an effort to preserve the original order of the messages. This automatic batching waits a default of 250ms after every call to **session.send()** before initiating the next call to **send()**.

The message batching delay is configurable. To disable the SDK's auto-batching logic, set the default delay to a large number and then manually call **sendBatch()** with a callback to invoke after the batch is delivered.

Send an Adaptive card

The Adaptive Card can contain any combination of text, speech, images, buttons, and input fields. Adaptive Cards are created using the JSON format specified in [Adaptive Cards](#), which gives you full control over card content and format.

To create an Adaptive Card using Node.js, leverage the information within the [Adaptive Cards](#) site to understand Adaptive Card schema, explore Adaptive Card elements, and see JSON samples that can be used to create cards of varying composition and complexity. Additionally, you can use the Interactive Visualizer to design Adaptive Card payloads and preview card output.

This code example shows how to create a message that contains an Adaptive Card for a calendar reminder:

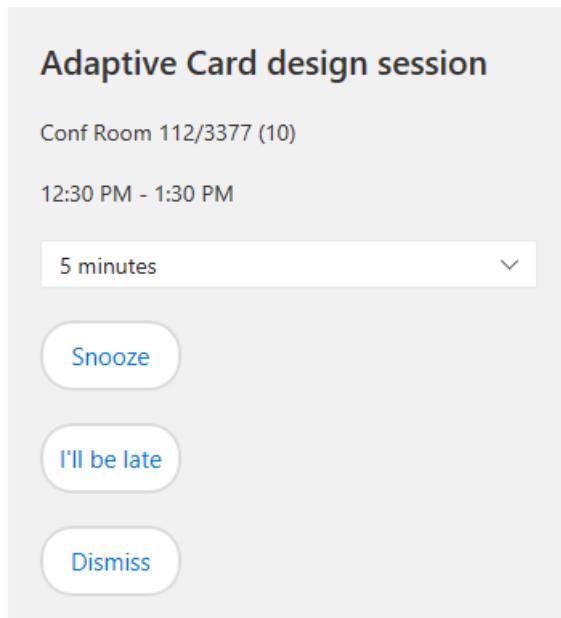
```
var msg = new builder.Message(session)
    .addAttachment({
        contentType: "application/vnd.microsoft.card.adaptive",
        content: {
            type: "AdaptiveCard",
            speak: "<s>Your meeting about \"Adaptive Card design session\"<br/> is starting at 12:30pm</s><s>Do you want to snooze <br/> or do you want to send a late notification to the attendees?</s>",
            body: [
                {
                    type: "TextBlock",
                    text: "Adaptive Card design session",
                    size: "large",
                    weight: "bolder"
                },
                {
                    type: "TextBlock",
                    text: "Conf Room 112/3377 (10)"
                },
                {
                    type: "TextBlock",
                    text: "12:30 PM - 1:30 PM"
                },
                {
                    type: "TextBlock",
                    text: "Snooze for"
                },
                {
                    type: "Input.ChoiceSet",
                    id: "snooze",
                    style: "compact",
                    choices: [
                        {
                            title: "5 minutes",
                            value: "5",
                            isSelected: true
                        },
                        {
                            title: "15 minutes",
                            value: "15"
                        },
                        {
                            title: "30 minutes",
                            value: "30"
                        }
                    ]
                }
            ],
            actions: [
                {
                    type: "Action.Submit",
                    title: "Snooze"
                }
            ]
        }
    })
    .actions([
        {
            type: "Action.OpenUrl",
            url: "https://adaptivecards.io/designer"
        }
    ])
    .send();
```

```

    },
    "type": "Action.Http",
    "method": "POST",
    "url": "http://foo.com",
    "title": "Snooze"
},
{
    "type": "Action.Http",
    "method": "POST",
    "url": "http://foo.com",
    "title": "I'll be late"
},
{
    "type": "Action.Http",
    "method": "POST",
    "url": "http://foo.com",
    "title": "Dismiss"
}
]
}
);

```

The resulting card contains three blocks of text, an input field (choice list), and three buttons:



Additional resources

- [Preview features with the Channel Inspector](#)
- [Adaptive Cards](#)
- [AnimationCard](#)
- [AudioCard](#)
- [HeroCard](#)
- [ThumbnailCard](#)
- [ReceiptCard](#)
- [SigninCard](#)
- [VideoCard](#)
- [Message](#)
- [How to send attachments](#)

Add speech to messages

7/26/2017 • 3 min to read • [Edit Online](#)

If you are building a bot for a speech-enabled channel such as Cortana, you can construct messages that specify the text to be spoken by your bot. You can also attempt to influence the state of the client's microphone by specifying an [input hint](#) to indicate whether your bot is accepting, expecting, or ignoring user input.

Specify text to be spoken by your bot

Using the Bot Builder SDK for Node.js, there are multiple ways to specify the text to be spoken by your bot on a speech-enabled channel. You can set the `IMessage.speak` property and send the message using the `session.send()` method, send the message using the `session.say()` method (passing parameters that specify display text, speech text, and options), or send the message using a built-in prompt (specifying options `speak` and `retrySpeak`).

`IMessage.speak`

If you are creating a message that will be sent using the `session.send()` method, set the `speak` property to specify the text to be spoken by your bot. The following code example creates a message that specifies text to be spoken and indicates that the bot is [accepting user input](#).

```
var msg = new builder.Message(session)
  .speak('This is the text that will be spoken.')
  .inputHint(builder.InputHint.acceptingInput);
session.send(msg).endDialog();
```

`session.say()`

As an alternative to using `session.send()`, you can call the `session.say()` method to create and send a message that specifies the text to be spoken, in addition to the text to be displayed and other options. The method is defined as follows:

```
session.say(displayText: string, speechText: string, options?: object)
```

PARAMETER	DESCRIPTION
<code>displayText</code>	The text to be displayed.
<code>speechText</code>	The text (in plain text or SSML format) to be spoken.
<code>options</code>	An IMessage object that can contain an attachment or input hint .

The following code example sends a message that specifies text to be displayed and text to be spoken and indicates that the bot is [ignoring user input](#).

```
session.say('Please hold while I calculate a response.',
  'Please hold while I calculate a response.',
  { inputHint: builder.InputHint.ignoringInput })
);
```

Prompt options

Using any of the built-in prompts, you can set the options `speak` and `retrySpeak` to specify the text to be spoken by your bot. The following code example creates a prompt that specifies text to be displayed, text to be spoken initially, and text to be spoken after waiting a while for user input. It indicates that the bot is [expecting user input](#) and uses [SSML](#) formatting to specify that the word "sure" should be spoken with a moderate amount of emphasis.

```
builder.Prompts.text(session, 'Are you sure that you want to cancel this transaction?', {
    speak: 'Are you <emphasis level=\"moderate\">sure</emphasis> that you want to cancel this transaction?',
    retrySpeak: 'Are you <emphasis level=\"moderate\">sure</emphasis> that you want to cancel this transaction?',
    inputHint: builder.InputHint.expectingInput
});
```

Speech Synthesis Markup Language (SSML)

To specify text to be spoken by your bot, you can use either a plain text string or a string that is formatted as Speech Synthesis Markup Language (SSML), an XML-based markup language that enables you to control various characteristics of your bot's speech such as voice, rate, volume, pronunciation, pitch, and more. For details about SSML, see [Speech Synthesis Markup Language Reference](#).

TIP

Use an [SSML library](#) to create well-formatted SSML.

Input hints

When you send a message on a speech-enabled channel, you can attempt to influence the state of the client's microphone by also including an input hint to indicate whether your bot is accepting, expecting, or ignoring user input. For more information, see [Add input hints to messages](#).

Sample code

For a complete sample that shows how to create a speech-enabled bot using the Bot Builder SDK for .NET, see the [Roller sample](#) in GitHub.

Additional resources

- [Speech Synthesis Markup Language \(SSML\)](#)
- [Roller sample \(GitHub\)](#)
- [Bot Builder SDK for Node.js Reference](#)

Add input hints to messages

7/26/2017 • 2 min to read • [Edit Online](#)

By specifying an input hint for a message, you can indicate whether your bot is accepting, expecting, or ignoring user input after the message is delivered to the client. For many channels, this enables clients to set the state of user input controls accordingly. For example, if a message's input hint indicates that the bot is ignoring user input, the client may close the microphone and disable the input box to prevent the user from providing input.

Accepting input

To indicate that your bot is passively ready for input but is not awaiting a response from the user, set the message's input hint to `builder.InputHint.acceptingInput`. On many channels, this will cause the client's input box to be enabled and microphone to be closed, but still accessible to the user. For example, Cortana will open the microphone to accept input from the user if the user holds down the microphone button. The following code example creates a message that indicates the bot is accepting user input.

```
var msg = new builder.Message(session)
    .speak('This is the text that will be spoken.')
    .inputHint(builder.InputHint.acceptingInput);
session.send(msg).endDialog();
```

Expecting input

To indicate that your bot is awaiting a response from the user, set the message's input hint to `builder.InputHint.expectingInput`. On many channels, this will cause the client's input box to be enabled and microphone to be open. The following code example creates a prompt that indicates the bot is expecting user input.

```
builder.Prompts.text(session, 'This is the text that will be displayed.', {
    speak: 'This is the text that will be spoken initially.',
    retrySpeak: 'This is the text that is spoken after waiting a while for user input.',
    inputHint: builder.InputHint.expectingInput
});
```

Ignoring input

To indicate that your bot is not ready to receive input from the user, set the message's input hint to `builder.InputHint.ignoringInput`. On many channels, this will cause the client's input box to be disabled and microphone to be closed. The following code example uses the `session.say()` method to send a message that indicates the bot is ignoring user input.

```
session.say('Please hold while I calculate a response. Thanks!',
    'Please hold while I calculate a response. Thanks!',
    { inputHint: builder.InputHint.ignoringInput }
);
```

Default values for input hint

If you do not set the input hint for a message, the Bot Builder SDK will automatically set it for you by using this

logic:

- If your bot sends a prompt, the input hint for the message will specify that your bot is **expecting input**.
- If your bot sends single message, the input hint for the message will specify that your bot is **accepting input**.
- If your bot sends a series of consecutive messages, the input hint for all but the final message in the series will specify that your bot is **ignoring input**, and the input hint for the final message in the series will specify that your bot is **accepting input**.

Additional resources

- [Add speech to messages](#)
- [Bot Builder SDK for Node.js Reference](#)

Add suggested actions to messages

9/18/2017 • 1 min to read • [Edit Online](#)

Suggested actions enable your bot to present buttons that the user can tap to provide input. Suggested actions appear close to the composer and enhance user experience by enabling the user to answer a question or make a selection with a simple tap of a button, rather than having to type a response with a keyboard. Unlike buttons that appear within rich cards (which remain visible and accessible to the user even after being tapped), buttons that appear within the suggested actions pane will disappear after the user makes a selection. This prevents the user from tapping stale buttons within a conversation and simplifies bot development (since you will not need to account for that scenario).

TIP

Use the [Channel Inspector](#) to see how suggested actions look and work on various channels.

Suggested actions example

To add suggested actions to a message, set the `suggestedActions` property of the message to a list of [card actions](#) that represent the buttons to be presented to the user.

This code example shows how to send a message that presents three suggested actions to the user:

```
var msg = new builder.Message(session)
  .text("Thank you for expressing interest in our premium golf shirt! What color of shirt would you like?")
  .suggestedActions(
    builder.SuggestedActions.create(
      session, [
        builder.CardAction.imBack(session, "productId=1&color=green", "Green"),
        builder.CardAction.imBack(session, "productId=1&color=blue", "Blue"),
        builder.CardAction.imBack(session, "productId=1&color=red", "Red")
      ]
    ));
session.send(msg);
```

When the user taps one of the suggested actions, the bot will receive a message from the user that contains the `value` of the corresponding action.

Be aware that the `imBack` method will post the `value` to the chat window of the channel you are using. If this is not the desired effect, you can use the `postBack` method, which will still post the selection back to your bot, but will not display the selection in the chat window. Some channels do not support `postBack`, however, and in those instances the method will behave like `imBack`.

Additional resources

- [Preview features with the Channel Inspector](#)
- [IMessage](#)
- [ICardAction](#)
- [session.send](#)

Send a typing indicator

8/7/2017 • 1 min to read • [Edit Online](#)

Users expect a timely response to their messages. If your bot performs some long-running task like calling a server or executing a query without giving the user some indication that the bot heard them, the user could get impatient and send additional messages or just assume the bot is broken. Many channels support the sending of a typing indication to show the user that the message was received and is being processed.

Typing indicator example

The following example demonstrates how to send a typing indication using [session.sendTyping\(\)](#). You can test this with the Bot Framework Emulator.

```
// Create bot and default message handler
var bot = new builder.UniversalBot(connector, function (session) {
    session.sendTyping();
    setTimeout(function () {
        session.send("Hello there...");
    }, 3000);
});
```

Typing indicators are also useful when inserting a message delay to prevent messages that contain images from being sent out of order.

To learn more, see [How to send a rich card](#).

Additional resources

- [sendTyping](#)

Intercept messages

8/7/2017 • 1 min to read • [Edit Online](#)

The **middleware** functionality in the Bot Builder SDK enables your bot to intercept all messages that are exchanged between user and bot. For each message that is intercepted, you may choose to do things such as save the message to a data store that you specify, which creates a conversation log, or inspect the message in some way and take whatever action your code specifies.

NOTE

The Bot Framework does not automatically save conversation details, as doing so could potentially capture private information that bots and users do not wish to share with outside parties. If your bot saves conversation details, it should communicate that to the user and describe what will be done with the data.

Example

The following code sample shows how to intercept messages that are exchanged between user and bot by using the concept of **middleware** in the Bot Builder SDK for Node.js.

First, configure the handler for incoming messages (`botbuilder`) and for outgoing messages (`send`).

```
server.post('/api/messages', connector.listen());
var bot = new builder.UniversalBot(connector);
bot.use({
  botbuilder: function (session, next) {
    myMiddleware.logIncomingMessage(session, next);
  },
  send: function (event, next) {
    myMiddleware.logOutgoingMessage(event, next);
  }
})
```

Then, implement each of the handlers to define the action to take for each message that is intercepted.

```
module.exports = {
  logIncomingMessage: function (session, next) {
    console.log(session.message.text);
    next();
  },
  logOutgoingMessage: function (event, next) {
    console.log(event.text);
    next();
  }
}
```

Now, every inbound message (from user to bot) will trigger `logIncomingMessage`, and every outbound message (from bot to user) will trigger `logOutgoingMessage`. In this example, the bot simply prints some information about each message, but you can update `logIncomingMessage` and `logOutgoingMessage` as necessary to define the actions that you want to take for each message.

Sample code

For a complete sample that shows how to intercept and log messages using the Bot Builder SDK for Node.js, see the [Middleware and Logging sample](#) in GitHub.

Build a speech-enabled bot with Cortana skills

8/9/2017 • 10 min to read • [Edit Online](#)

The Bot Builder SDK for Node.js enables you to build a speech-enabled bot by connecting it to the Cortana channel as a Cortana skill. Cortana skills let you provide functionality through Cortana in response to spoken input from a user.

TIP

For more information on what a skill is, and what they can do, see [The Cortana Skills Kit](#).

Creating a Cortana skill using Bot Framework requires very little Cortana-specific knowledge and primarily consists of building a bot. One of the key differences from other bots that you may have created is that Cortana has both visual and audio components. For the visual component, Cortana provides an area of the canvas for rendering content such as cards. For the audio component, you provide text or SSML in your bot's messages, which Cortana reads to the user, giving your bot a voice.

NOTE

Cortana is available on many different devices. Some have a screen while others, like a standalone speaker, might not. You should make sure that your bot is capable of handling both scenarios. See [Cortana-specific entities](#) to learn how to check device information.

Adding speech to your bot

Spoken messages from your bot are represented as Speech Synthesis Markup Language (SSML). The Bot Builder SDK lets you include SSML in your bot's responses to control what the bot says, in addition to what it shows.

`session.say`

Your bot uses the `session.say` method to speak to the user, in place of `session.send`. It includes optional parameters for sending SSML output, as well as attachments like cards.

The method has this format:

```
session.say(displayText: string, speechText: string, options?: object)
```

PARAMETER	DESCRIPTION
<code>displayText</code>	A textual message to display in Cortana's UI.
<code>speechText</code>	The text or SSML that Cortana reads to the user.
<code>options</code>	An IMessage object that can contain an attachment or input hint. Input hints indicate whether the bot is accepting, expecting, or ignoring input. Card attachments are displayed in Cortana's canvas below the <code>displayText</code> information.

The `inputHint` property helps indicate to Cortana whether your bot is expecting input. If you're using a built-in prompt, this value is automatically set to the default of `expectingInput`.

VALUE	DESCRIPTION
acceptingInput	Your bot is passively ready for input but is not waiting on a response. Cortana accepts input from the user if the user holds down the microphone button.
expectingInput	Indicates that the bot is actively expecting a response from the user. Cortana listens for the user to speak into the microphone.
ignoringInput	Cortana is ignoring input. Your bot may send this hint if it is actively processing a request and will ignore input from users until the request is complete.

The following example shows how Cortana reads plain text or SSML:

```
// Have Cortana read plain text
session.say('This is the text that Cortana displays', 'This is the text that is spoken by Cortana.');

// Have Cortana read SSML
session.say('This is the text that Cortana displays', '<speak version="1.0"
xmlns="http://www.w3.org/2001/10/synthesis" xml:lang="en-US">This is the text that is spoken by Cortana.
</speak>');
```

This example shows how to let Cortana know that user input is expected. The microphone will be left open.

```
// Add an InputHint to let Cortana know to expect user input
session.say('Hi there', 'Hi, what's your name?', {
    inputHint: builder.InputHint.expectingInput
});
```

Prompts

In addition to using the **session.say()** method you can also pass text or SSML to built-in prompts using the **speak** and **retrySpeak** options.

```
builder.Prompts.text(session, 'text based prompt', {
    speak: 'Cortana reads this out initially',
    retrySpeak: 'This message is repeated by Cortana after waiting a while for user input',
    inputHint: builder.InputHint.expectingInput
});
```

To present the user with a list of choices, use **Prompts.choice**. The **synonyms** option allows for more flexibility in recognizing user utterances. The **value** option is returned in **results.response.entity**. The **action** option specifies the label that your bot displays for the choice.

Prompts.choice supports ordinal choices. This means that the user can say "the first", "the second" or "the third" to choose an item in a list. For example, given the following prompt, if the user asked Cortana for "the second option", the prompt will return the value of 8.

```

var choices = [
    { value: '4', action: { title: '4 Sides' }, synonyms: 'four|for|4 sided|4 sides' },
    { value: '8', action: { title: '8 Sides' }, synonyms: 'eight|ate|8 sided|8 sides' },
    { value: '12', action: { title: '12 Sides' }, synonyms: 'twelve|12 sided|12 sides' },
    { value: '20', action: { title: '20 Sides' }, synonyms: 'twenty|20 sided|20 sides' },
];
builder.Prompts.choice(session, 'choose_sides', choices, {
    speak: speak(session, 'choose_sides_ssml') // use helper function to format SSML
});

```

In the previous example, the SSML for the prompt's **speak** property is formatted by using strings stored in a localized prompts file with the following format.

```
{
    "choose_sides": "__Number of Sides__",
    "choose_sides_ssml": [
        "How many sides on the dice?",
        "Pick your poison.",
        "All the standard sizes are supported."
    ]
}
```

A helper function then builds the required root element of a Speech Synthesis Markup Language (SSML) document.

```

module.exports.speak = function (template, params, options) {
    options = options || {};
    var output = '<speak xmlns="http://www.w3.org/2001/10/synthesis" ' +
        'version="' + (options.version || '1.0') + '" ' +
        'xml:lang="' + (options.lang || 'en-US') + '"';
    output += module.exports.vsprintf(template, params);
    output += '</speak>';
    return output;
}

```

TIP

You can find a small utility module (`ssml.js`) for building your bot's SSML-based responses in the [Roller sample skill](#). There are also several useful SSML libraries available through [npm](#) which make it easy to create well formatted SSML.

Display cards in Cortana

In addition to spoken responses, Cortana can also display card attachments. Cortana supports the following rich cards:

- [HeroCard](#)
- [ReceiptCard](#)
- [ThumbnailCard](#)

See [Card design best practices](#) to see what these cards look like inside Cortana. For an example of how to add a rich card to a bot, see [Send rich cards](#).

The following code demonstrates how to add the **speak** and **inputHint** properties to a message containing a Hero card.

```

bot.dialog('HelpDialog', function (session) {
  var card = new builder.HeroCard(session)
    .title('help_title')
    .buttons([
      builder.CardAction.imBack(session, 'roll some dice', 'Roll Dice'),
      builder.CardAction.imBack(session, 'play yahtzee', 'Play Yahtzee')
    ]);
  var msg = new builder.Message(session)
    .speak(speak(session, 'I\'m roller, the dice rolling bot. You can say \'roll some dice\''))
    .addAttachment(card)
    .inputHint(builder.InputHint.acceptingInput); // Tell Cortana to accept input
  session.send(msg).endDialog();
}).triggerAction({ matches: '/help/i });

/** This helper function builds the required root element of a Speech Synthesis Markup Language (SSML)
document. */
module.exports.speak = function (template, params, options) {
  options = options || {};
  var output = '<speak xmlns="http://www.w3.org/2001/10/synthesis" ' +
    'version="' + (options.version || '1.0') + '" ' +
    'xml:lang="' + (options.lang || 'en-US') + '"';
  output += module.exports.vsprintf(template, params);
  output += '</speak>';
  return output;
}

```

Sample: RollerSkill

The code in the following sections comes from a sample Cortana skill for rolling dice. Download the full code for the bot from the [BotBuilder-Samples repository](#).

You invoke the skill by saying its [invocation name](#) to Cortana. For the roller skill, after you [add the bot to the Cortana channel](#) and register it as a Cortana skill, you can invoke it by telling Cortana "Ask Roller" or "Ask Roller to roll dice".

Explore the code

The RollerSkill sample starts by opening a card with some buttons to tell the user which options are available to them.

```

/**
 * Create your bot with a default message handler that receive messages from the user.
 * - This function is be called anytime the user's utterance isn't
 *   recognized by the other handlers in the bot.
 */
var bot = new builder.UniversalBot(connector, function (session) {
    // Just redirect to our 'HelpDialog'.
    session.replaceDialog('HelpDialog');
});

//...

bot.dialog('HelpDialog', function (session) {
    var card = new builder.HeroCard(session)
        .title('help_title')
        .buttons([
            builder.CardAction.imBack(session, 'roll some dice', 'Roll Dice'),
            builder.CardAction.imBack(session, 'play craps', 'Play Craps')
        ]);
    var msg = new builder.Message(session)
        .speak(speak(session, 'help_ssml'))
        .addAttachment(card)
        .inputHint(builder.InputHint.acceptingInput);
    session.send(msg).endDialog();
}).triggerAction({ matches: '/help/i '});

```

Prompt the user for input

The following dialog sets up a custom game for the bot to play. It asks the user how many sides they want the dice to have and then how many should be rolled. Once it has built the game structure it will pass it to a separate 'PlayGameDialog'.

To start the dialog, the **triggerAction()** handler on this dialog allows a user to say something like "I'd like to roll some dice". It uses a regular expression to match the user's input but you could just as easily use a [LUIS intent](#).

```

bot.dialog('CreateGameDialog', [
  function (session) {
    // Initialize game structure.
    // - dialogData gives us temporary storage of this data in between
    //   turns with the user.
    var game = session.dialogData.game = {
      type: 'custom',
      sides: null,
      count: null,
      turns: 0
    };

    var choices = [
      { value: '4', action: { title: '4 Sides' }, synonyms: 'four|for|4 sided|4 sides' },
      { value: '6', action: { title: '6 Sides' }, synonyms: 'six|sex|6 sided|6 sides' },
      { value: '8', action: { title: '8 Sides' }, synonyms: 'eight|8 sided|8 sides' },
      { value: '10', action: { title: '10 Sides' }, synonyms: 'ten|10 sided|10 sides' },
      { value: '12', action: { title: '12 Sides' }, synonyms: 'twelve|12 sided|12 sides' },
      { value: '20', action: { title: '20 Sides' }, synonyms: 'twenty|20 sided|20 sides' },
    ];
    builder.Prompts.choice(session, 'choose_sides', choices, {
      speak: speak(session, 'choose_sides_ssml')
    });
  },
  function (session, results) {
    // Store users input
    // - The response comes back as a find result with index & entity value matched.
    var game = session.dialogData.game;
    game.sides = Number(results.response.entity);

    /**
     * Ask for number of dice.
     */
    var prompt = session.gettext('choose_count', game.sides);
    builder.Prompts.number(session, prompt, {
      speak: speak(session, 'choose_count_ssml'),
      minValue: 1,
      maxValue: 100,
      integerOnly: true
    });
  },
  function (session, results) {
    // Store users input
    // - The response is already a number.
    var game = session.dialogData.game;
    game.count = results.response;

    /**
     * Play the game we just created.
     *
     * replaceDialog() ends the current dialog and start a new
     * one in its place. We can pass arguments to dialogs so we'll pass the
     * 'PlayGameDialog' the game we created.
     */
    session.replaceDialog('PlayGameDialog', { game: game });
  }
]).triggerAction({ matches: [
  /(roll|role|throw|shoot).*(dice|die|dye|bones)/i,
  /new game/i
]}));

```

Render results

This dialog is our main game loop. The bot stores the game structure in **session.conversationData** so that should

the user say "roll again" we can just re-roll the same set of dice again.

```
bot.dialog('PlayGameDialog', function (session, args) {
    // Get current or new game structure.
    var game = args.game || session.conversationData.game;
    if (game) {
        // Generate rolls
        var total = 0;
        var rolls = [];
        for (var i = 0; i < game.count; i++) {
            var roll = Math.floor(Math.random() * game.sides) + 1;
            if (roll > game.sides) {
                // Accounts for 1 in a million chance random() generated a 1.0
                roll = game.sides;
            }
            total += roll;
            rolls.push(roll);
        }

        // Format roll results
        var results = '';
        var multiLine = rolls.length > 5;
        for (var i = 0; i < rolls.length; i++) {
            if (i > 0) {
                results += ' . ';
            }
            results += rolls[i];
        }

        // Render results using a card
        var card = new builder.HeroCard(session)
            .subtitle(game.count > 1 ? 'card_subtitle_plural' : 'card_subtitle_singular', game)
            .buttons([
                builder.CardAction.imBack(session, 'roll again', 'Roll Again'),
                builder.CardAction.imBack(session, 'new game', 'New Game')
            ]);
        if (multiLine) {
            //card.title('card_title').text('\n\n' + results + '\n\n');
            card.text(results);
        } else {
            card.title(results);
        }
        var msg = new builder.Message(session).addAttachment(card);

        // Determine bots reaction for speech purposes
        var reaction = 'normal';
        var min = game.count;
        var max = game.count * game.sides;
        var score = total/max;
        if (score == 1.0) {
            reaction = 'best';
        } else if (score == 0) {
            reaction = 'worst';
        } else if (score <= 0.3) {
            reaction = 'bad';
        } else if (score >= 0.8) {
            reaction = 'good';
        }

        // Check for special craps rolls
        if (game.type == 'craps') {
            switch (total) {
                case 2:
                case 3:
                case 12:
                    reaction = 'craps_lose';
                    break;
                case 7:
                    reaction = 'craps_win';
                    break;
            }
        }
    }
});
```

```

        case 7:
            reaction = 'craps_seven';
            break;
        case 11:
            reaction = 'craps_eleven';
            break;
        default:
            reaction = 'craps_retry';
            break;
    }
}

// Build up spoken response
var spoken = '';
if (game.turn == 0) {
    spoken += sessiongettext('start_' + game.type + '_game_ssml') + ' ';
}
spoken += sessiongettext(reaction + '_roll_reaction_ssml');
msg.speak(ssml.speak(spoken));

// Increment number of turns and store game to roll again
game.turn++;
session.conversationData.game = game;

/**
 * Send card and bot's reaction to user.
 */
msg.inputHint(builder.InputHint.acceptingInput);
session.send(msg).endDialog();
} else {
    // User started session with "roll again" so let's just send them to
    // the 'CreateGameDialog'
    session.replaceDialog('CreateGameDialog');
}
}).triggerAction({ matches: /(roll|role|throw|shoot) again/i });

```

Next steps

If you have a bot running locally or deployed in the cloud, you can invoke it from Cortana. See [Test a Cortana skill](#) for the steps required to try out your Cortana Skill.

Additional resources

- [The Cortana Skills Kit](#)
- [Add speech to messages](#)
- [SSML Reference](#)
- [Voice design best practices for Cortana](#)
- [Card design best practices for Cortana](#)
- [Cortana Dev Center](#)
- [Testing and debugging best practices for Cortana](#)

Support audio calls with Skype

9/6/2017 • 4 min to read • [Edit Online](#)

Skype supports a rich feature called Calling Bots. When enabled, users can place a voice call to your bot and interact with it using Interactive Voice Response (IVR). The Bot Builder for Node.js SDK includes a special [Calling SDK](#) which developers can use to add calling features to their chat bot.

The Calling SDK is very similar to the [Chat SDK](#). They have similar classes, share common constructs and you can even use the Chat SDK to send a message to the user you're on a call with. The two SDKs are designed to run side-by-side but while they are similar, there are some significant differences and you should generally avoid mixing classes from the two libraries.

Create a calling bot

The following example code shows how the "Hello World" for a calling bot looks very similar to a regular chat bot.

```
var restify = require('restify');
var calling = require('botbuilder-calling');

// Setup Restify Server
var server = restify.createServer();
server.listen(process.env.port || process.env.PORT || 3978, function () {
    console.log(`[${server.name}] listening to ${server.url}`);
});

// Create calling bot
var connector = new calling.CallConnector({
    callbackUrl: 'https://<your host>/api/calls',
    appId: '<your bots app id>',
    appPassword: '<your bots app password>'
});
var bot = new calling.UniversalCallBot(connector);
server.post('/api/calls', connector.listen());

// Add root dialog
bot.dialog('/', function (session) {
    session.send('Watson... come here!');
});
```

The emulator doesn't currently support testing calling bots. To test your bot, you'll need to go through most of the steps required to publish your bot. You will also need to use a Skype client to interact with the bot.

Enable the Skype channel

[Register your bot](#) and enable the Skype channel. You will need provide a messaging endpoint when you register your bot in the developer portal. It is recommended that you pair your calling bot with a chat bot so the chat bot's endpoint is what you would normally put in that field. If you're only registering a calling bot you can simply paste your calling endpoint into that field.

To enable the actual calling feature you'll need to go into the Skype channel for your bot and turn on the calling feature. You'll then be provided with a field to copy your calling endpoint into. Make sure you use the https ngrok link for the host portion of your calling endpoint.

During the registration of your bot you'll be assigned an app ID & password which you should paste into the connector settings for your hello world bot. You'll also need to take your full calling link and paste that in for the callbackUrl setting.

Add bot to contacts

On your bot's registration page in the developer portal you'll see an **add to Skype** button next to your bots Skype channel. Click the button to add your bot to your contact list in Skype. Once you do that you (and anyone you give the join link to) will be able to communicate with the bot.

Test your bot

You can test your bot using a Skype client. You should notice the call icon light up when you click on your bots contact entry (you may have to search for the bot to see it.) It can take a few minutes for the call icon to light up if you've added calling to an existing bot.

If you press the call button it should dial your bot and you should hear it speak "Watson... come here!" and then hang up.

Calling basics

The [UniversalCallBot](#) and [CallConnector](#) classes let you author a calling bot in much the same way you would a chat bot. You add dialogs to your bot that are essentially identical to [chat dialogs](#). You can add [waterfalls](#) to your bot. The a session object, the [CallSession](#) class, which contains added [answer\(\)](#), [hangup\(\)](#), and [reject\(\)](#) methods for managing the current call. In general, you don't need to worry about these call control methods though as the CallSession has logic to automatically manage the call for you. The session will automatically answer the call if you take an action like sending a message or calling a built-in prompt. It will also automatically hangup/reject the call if you call [endConversation\(\)](#) or it detects that you've stopped asking the caller questions (you didn't call a built-in prompt.)

Another difference between calling and chat bots is that while chat bots typically send messages, cards, and keyboards to a user a calling bot deals in Actions and Outcomes. Skype calling bots are required to create [workflows](#) that are comprised of one or more [actions](#). This is another thing that in practice you don't have to worry too much about as the Bot Builder calling SDK will manage most of this for you. The [CallSession.send\(\)](#) method lets you pass either actions or strings which it will turn into [PlayPromptActions](#). The session contains auto batching logic to combine multiple actions into a single workflow that's submitted to the calling service so you can safely call [send\(\)](#) multiple times. And you should rely on the SDK's built-in [prompts](#) to collect input from the user as they process all of the outcomes.

Manage state data

9/6/2017 • 5 min to read • [Edit Online](#)

The Bot Framework State service enables your bot to store and retrieve state data that is associated with a user, a conversation, or a specific user within the context of a specific conversation. State data can be used for many purposes, such as determining where the prior conversation left off or simply greeting a returning user by name. If you store a user's preferences, you can use that information to customize the conversation the next time you chat. For example, you might alert the user to a news article about a topic that interests her, or alert a user when an appointment becomes available.

In the Builder SDK for Node.js, the [ChatConnector](#) class provides an implementation of this storage system and you can use the `session` object to store, retrieve, and delete state data.

Storage containers

In the Bot Builder SDK for Node.js, the `session` object exposes the following properties for storing state data.

PROPERTY	SCOPED TO	DESCRIPTION
<code>userData</code>	User	Contains data that is saved for the user on the specified channel. This data will persist across multiple conversations.
<code>privateConversationData</code>	Conversation	Contains data that is saved for the user within the context of a particular conversation on the specified channel. This data is private to the current user and will persist for the current conversation only. The property is cleared when the conversation ends or when <code>endConversation</code> is called explicitly.
<code>conversationData</code>	Conversation	Contains data that is saved in the context of a particular conversation on the specified channel. This data is shared with all users participating in the conversation and will persist for the current conversation only. The property is cleared when the conversation ends or when <code>endConversation</code> is called explicitly.
<code>dialogData</code>	Dialog	Contains data that is saved for the current dialog only. Each dialog maintains its own copy of this property. The property is cleared when the dialog is removed from the dialog stack.

These four properties correspond to the four data storage containers that can be used to store data. Which properties you use to store data will depend upon the appropriate scope for the data you are storing, the nature of the data that you are storing, and how long you want the data to persist. For example, if you need to store user data that will be available across multiple conversations, consider using the `userData` property. If you need to

temporarily store local variable values within the scope of a dialog, consider using the `dialogData` property. If you need to temporarily store data that must be accessible across multiple dialogs, consider using the `conversationData` property.

Data Persistence

By default, data that is stored using the `userData`, `privateConversationData`, and `conversationData` properties is set to persist after the conversation ends. If you do not want the data to persist in the `userData` container, set the `persistUserData` flag to **false**. If you do not want the data to persist in the `conversationData` container, set the `persistConversationData` flag to **false**.

```
// Do not persist userData  
bot.set(`persistUserData`, false);  
  
// Do not persist conversationData  
bot.set(`persistConversationData`, false);
```

NOTE

You cannot disable data persistence for the `privateConversationData` container; it is always persisted.

Set data

You can store simple JavaScript objects by saving them directly to a storage container. For a complex object like `Date`, consider converting it to `string`. This is because state data is serialized and stored as JSON. The following code samples show how to store primitive data, an array, an object map, and a complex `Date` object.

Store primitive data

```
session.userData.userName = "Kumar Sarma";  
session.userData.userAge = 37;  
session.userData.hasChildren = true;
```

Store an array

```
session.userData.profile = ["Kumar Sarma", "37", "true"];
```

Store an object map

```
session.userData.about = {  
    "Profile": {  
        "Name": "Kumar Sarma",  
        "Age": 37,  
        "hasChildren": true  
    },  
    "Job": {  
        "Company": "Contoso",  
        "StartDate": "June 8th, 2010",  
        "Title": "Developer"  
    }  
}
```

Store Date and Time

For a complex JavaScript object, convert it to a string before saving to storage container.

```
var startDate = builder.EntityRecognizer.resolveTime([results.response]);  
  
// Date as string: "2017-08-23T05:00:00.000Z"  
session.userData.start = startDate.toISOString();
```

Saving data

Data that is created in each storage container will remain in memory until the container is saved. The Bot Builder SDK for Node.js sends data to the `chatConnector` service in batches to be saved when there are messages to be sent. To save the data that exists in the storage containers without sending any messages, you can manually call the `save` method. If you do not call the `save` method, the data that exists in the storage containers will be persisted as part of the batch processing.

```
session.userData.favoriteColor = "Red";  
session.userData.about.job.Title = "Senior Developer";  
session.save();
```

Get data

To access the data that is saved in a particular storage container, simply reference the corresponding property. The following code samples show how to access data that was previously stored as primitive data, an array, an object map, and a complex Date object.

Access primitive data

```
var userName = session.userData.userName;  
var userAge = session.userData.userAge;  
var hasChildren = session.userData.hasChildren;
```

Access an array

```
var userProfile = session.userData.userProfile;  
  
session.send("User Profile:");  
for(int i = 0; i < userProfile.length, i++){  
    session.send(userProfile[i]);  
}
```

Access an object map

```
var about = session.userData.about;  
  
session.send("User %s works at %s.", about.Profile.Name, about.Job.Company);
```

Access a Date object

Retrieve date data as string then convert it into a JavaScript's Date object.

```
// startDate as a JavaScript Date object.  
var startDate = new Date(session.userData.start);
```

Delete data

By default, data that is stored in the `dialogData` container is cleared when a dialog is removed from the dialog stack. Likewise, data that is stored in the `conversationData` container and `privateConversationData` container is cleared when the `endConversation` method is called. However, to delete data stored in the `userData` container, you have to explicitly clear it.

To explicitly clear the data that is stored in any of the storage containers, simply reset the container as shown in the following code sample.

```
// Clears data stored in container.  
session.userData = {};  
session.privateConversationData = {};  
session.conversationData = {};  
session.dialogData = {};
```

Never set a data container `null` or remove it from the `session` object, as doing so will cause errors the next time you try to access the container. Also, you may want to manually call `session.save();` after you manually clear a container in memory, to clear any corresponding data that has previously been persisted.

Manage data storage

Under the hood, the Bot Builder SDK for Node.js stores state data using the Bot Connector State service, which is intended for prototyping only and is not designed for use by bots in a production environment. For performance and security reasons in the production environment, consider implementing one of the following data storage options:

1. [Manage state data with Cosmos DB](#)
2. [Manage state data with Table storage](#)

With either of these [Azure Extensions](#) options, the mechanism for setting and persisting data via the Bot Framework SDK for Node.js remains the same as described previously in this article.

Next steps

Now that you understand how to manage user state data, let's take a look at how you can use it to better manage conversation flow.

[Manage conversation flow](#)

Additional resources

- [Prompt user for input](#)

Manage custom state data with Azure Cosmos DB for Node.js

9/6/2017 • 3 min to read • [Edit Online](#)

In this article, you'll implement Cosmos DB storage to store and manage your bot's state data. The default Connector State Service used by bots is not intended for the production environment. You should either use [Azure Extensions](#) available on GitHub or implement a custom state client using data storage platform of your choice. Here are some of the reasons to use custom state storage:

- higher state API throughput (more control over performance)
- lower-latency for geo-distribution
- control over where the data is stored (e.g.: West US vs East US)
- access to the actual state data
- state data db not shared with other bots
- store more than 32kb

Prerequisites

- [Node.js](#).
- [Bot Framework Emulator](#)
- Must have a Node.js bot. If you do not have one, go [create a bot](#).

Create Azure account

If you don't have an Azure account, click [here](#) to sign up for a free trial.

Set up the Azure Cosmos DB database

1. After you've logged into the Azure portal, create a new *Azure Cosmos DB* database by clicking **New**.
2. Click **Databases**.
3. Find **Azure Cosmos DB** and click **Create**.
4. Fill in the fields. For the **API** field, select **SQL (DocumentDB)**. When done filling in all the fields, click the **Create** button at the bottom of the screen to deploy the new database.
5. After the new database is deployed, navigate to your new database. Click **Access keys** to find keys and connection strings. Your bot will use this information to call the storage service to save state data.

Install botbuilder-azure module

To install the `botbuilder-azure` module from a command prompt, navigate to the bot's directory and run the following npm command:

```
npm install --save botbuilder-azure
```

Modify your bot code

To use your **Azure Cosmos DB** database, add the following lines of code to your bot's **app.js** file.

1. Require the newly installed module.

```
var azure = require('botbuilder-azure');
```

2. Configure the connection settings to connect to Azure.

```
var documentDbOptions = {
  host: 'Your-Azure-DocumentDB-URI',
  masterKey: 'Your-Azure-DocumentDB-Key',
  database: 'botdocs',
  collection: 'botdata'
};
```

The `host` and `masterKey` values can be found in the **Keys** menu of your database. If the `database` and `collection` entries do not exist in the Azure database, they will be created for you.

3. Using the `botbuilder-azure` module, create two new objects to connect to the Azure database. First, create an instance of `DocumentDbClient` passing in the connection configuration settings (defined as `documentDbOptions` from above). Next, create an instance of `AzureBotStorage` passing in the `DocumentDbClient` object. For example:

```
var docDbClient = new azure.DocumentDbClient(documentDbOptions);

var cosmosStorage = new azure.AzureBotStorage({ gzipData: false }, docDbClient);
```

4. Specify that you want to use your custom database instead of the default database. For example:

```
var bot = new builder.UniversalBot(connector, function (session) {
  // ... Bot code ...
})
.set('storage', cosmosStorage);
```

Now you are ready to test the bot with the emulator.

Run your bot app

From a command prompt, navigate to your bot's directory and run your bot with the following command:

```
node app.js
```

Connect your bot to the emulator

At this point, your bot is running locally. Start the emulator and then connect to your bot from the emulator:

1. Type `http://localhost:port-number/api/messages` into the address bar, where port-number matches the port number shown in the browser where your application is running. You can leave **Microsoft App ID** and **Microsoft App Password** fields blank for now. You'll get this information later when you register your bot.
2. Click **Connect**.
3. Test your bot by sending your bot a message. Interact with your bot as you normally would. When you are done, go to **Storage Explorer** and view your saved state data.

View state data on Azure Portal

To view the state data, sign into your Azure portal and navigate to your database. Click [Data Explorer \(preview\)](#) to verify that the state information from your bot is being saved.

Next step

Now that you have full control over your bot's state data, let's take a look at how you can use it to better manage conversation flow.

[Manage conversation flow](#)

Manage custom state data with Azure Table storage for Node.js

9/6/2017 • 3 min to read • [Edit Online](#)

In this article, you'll implement Azure Table storage to store and manage your bot's state data. The default Connector State Service used by bots is not intended for the production environment. You should either use [Azure Extensions](#) available on GitHub or implement a custom state client using data storage platform of your choice. Here are some of the reasons to use custom state storage:

- higher state API throughput (more control over performance)
- lower-latency for geo-distribution
- control over where the data is stored (e.g.: West US vs East US)
- access to the actual state data
- state data db not shared with other bots
- store more than 32kb

Prerequisites

- [Node.js](#).
- [Bot Framework Emulator](#).
- Must have a Node.js bot. If you do not have one, go [create a bot](#).
- [Storage Explorer](#).

Create Azure account

If you don't have an Azure account, click [here](#) to sign up for a free trial.

Set up the Azure Table storage service

1. After you've logged into the Azure portal, create a new Azure Table storage service by clicking on **New**.
2. Search for **Storage account** that implements the Azure Table. Click **Create** to start creating the storage account.
3. Fill in the fields, click the **Create** button at the bottom of the screen to deploy the new storage service.
4. After the new storage service is deployed, navigate to the storage account you just created. You can find it listed in the **Storage Accounts** blade.
5. Select **Access keys**, and copy the key for later use. Your bot will use **Storage account name** and **Key** to call the storage service to save state data.

Install botbuilder-azure module

To install the `botbuilder-azure` module from a command prompt, navigate to the bot's directory and run the following npm command:

```
npm install --save botbuilder-azure
```

Modify your bot code

To use your **Azure Table** storage, add the following lines of code to your bot's **app.js** file.

1. Require the newly installed module.

```
var azure = require('botbuilder-azure');
```

2. Configure the connection settings to connect to Azure.

```
// Table storage
var tableName = "Table-Name"; // You define
var storageName = "Table-Storage-Name"; // Obtain from Azure Portal
var storageKey = "Azure-Table-Key"; // Obtain from Azure Portal
```

The `storageName` and `storageKey` values can be found in the **Access keys** menu of your Azure Table. If the `tableName` does not exist in the Azure Table, it will be created for you.

3. Using the `botbuilder-azure` module, create two new objects to connect to the Azure Table. First, create an instance of `AzureTableClient` passing in the connection configuration settings. Next, create an instance of `AzureBotStorage` passing in the `AzureTableClient` object. For example:

```
var azureTableClient = new azure.AzureTableClient(tableName, storageName, storageKey);

var tableStorage = new azure.AzureBotStorage({gzipData: false}, azureTableClient);
```

4. Specify that you want to use your custom database instead of the default database. For example:

```
var bot = new builder.UniversalBot(connector, function (session) {
    // ... Bot code ...
})
.set('storage', tableStorage);
```

Now you are ready to test the bot with the emulator.

Run your bot app

From a command prompt, navigate to your bot's directory and run your bot with the following command:

```
node app.js
```

Connect your bot to the emulator

At this point, your bot is running locally. Start the emulator and then connect to your bot from the emulator:

1. Type `http://localhost:port-number/api/messages` into the address bar, where port-number matches the port number shown in the browser where your application is running. You can leave **Microsoft App ID** and **Microsoft App Password** fields blank for now. You'll get this information later when you register your bot.
2. Click **Connect**.
3. Test your bot by sending your bot a message. Interact with your bot as you normally would. When you are done, go to **Storage Explorer** and view your saved state data.

View data in Storage Explorer

To view the state data, open **Storage Explorer** and connect to Azure using your Azure Portal credential or connect

directly to the Table using the `storageName` and `storageKey` and navigate to your `tableName`.

Next step

Now that you have full control over your bot's state data, let's take a look at how you can use it to better manage conversation flow.

[Manage conversation flow](#)

Recognize user intent from message content

8/9/2017 • 3 min to read • [Edit Online](#)

When your bot receives a message from a user, your bot can use a **recognizer** to examine the message and determine intent. The intent provides a mapping from messages to dialogs to invoke. This article explains how to recognize intent using regular expressions or by inspecting the content of a message. For example, a bot can use regular expressions to check if a message contains the word "help" and invoke a help dialog. A bot can also check the properties of the user message, for example, to see if the user sent an image instead of text and invoke an image processing dialog.

NOTE

For information on recognizing intent using LUIS, see [Recognize intents and entities with LUIS](#)

Use the built-in regular expression recognizer

Use [RegExpRecognizer](#) to detect the user's intent using a regular expression. You can pass multiple expressions to the recognizer to support multiple languages.

TIP

See [Support localization](#) for more information on localizing your bot.

The following code creates a regular expression recognizer named `CancelIntent` and adds it to your bot. The recognizer in this example provides regular expressions for both the `en_us` and `ja_jp` locales.

```
bot.recognizer(new builder.RegExpRecognizer( "CancelIntent", { en_us: /^(cancel|nevermind)/i, ja_jp: /^(キャンセル)/ }));
```

Once the recognizer is added to your bot, attach a [triggerAction](#) to the dialog that you want the bot to invoke when the recognizer detects the intent. Use the `matches` option to specify the intent name, as shown in the following code:

```
bot.dialog('CancelDialog', function (session) {
    session.endConversation("Ok, I'm canceling your order.");
}).triggerAction({ matches: 'CancelIntent' });
```

Intent recognizers are global, which means that the recognizer will run for every message received from the user. If a recognizer detects an intent that is bound to a dialog using a `triggerAction`, it can trigger interruption of the currently active dialog. Allowing and handling interruptions is a flexible design that accounts for what users really do.

TIP

To learn how a `triggerAction` works with dialogs, see [Managing conversation flow](#). To learn more about the various actions you can associate with a recognized intent, [Handle user actions](#).

Register a custom intent recognizer

You can also implement a custom recognizer. This example adds a simple recognizer that looks for the user to say 'help' or 'goodbye' but you could easily add a recognizer that does more complex processing, such as one that recognizes when the user sends an image.

```
var builder = require('..../core/');

// Create bot and default message handler
var connector = new builder.ConsoleConnector().listen();
var bot = new builder.UniversalBot(connector, function (session) {
    session.send("You said: '%s'. Try asking for 'help' or say 'goodbye' to quit", session.message.text);
});

// Install a custom recognizer to look for user saying 'help' or 'goodbye'.
bot.recognizer({
    recognize: function (context, done) {
        var intent = { score: 0.0 };

        if (context.message.text) {
            switch (context.message.text.toLowerCase()) {
                case 'help':
                    intent = { score: 1.0, intent: 'Help' };
                    break;
                case 'goodbye':
                    intent = { score: 1.0, intent: 'Goodbye' };
                    break;
            }
        }
        done(null, intent);
    }
});
```

Once you've registered a recognizer, you can associate the recognizer with an action using a `matches` clause.

```
// Add a help dialog with a trigger action that is bound to the 'Help' intent
bot.dialog('helpDialog', function (session) {
    session.endDialog("This bot will echo back anything you say. Say 'goodbye' to quit.");
}).triggerAction({ matches: 'Help' });

// Add a global endConversation() action that is bound to the 'Goodbye' intent
bot.endConversationAction('goodbyeAction', "Ok... See you later.", { matches: 'Goodbye' });
```

Disambiguate between multiple intents

Your bot can register more than one recognizer. Notice that the custom recognizer example involves assigning a numerical score to each intent. This is done since your bot may have more than one recognizer, and the Bot Builder SDK provides built-in logic to disambiguate between intents returned by multiple recognizers. The score assigned to an intent is typically between 0.0 and 1.0, but a custom recognizer may define an intent greater than 1.1 to ensure that that intent will always be chosen by the Bot Builder SDK's disambiguation logic.

By default, recognizers run in parallel, but you can set `recognizeOrder` in [IIntentRecognizerSetOptions](#) so the process quits as soon as your bot finds one that gives a score of 1.0.

The Bot Builder SDK includes a [sample](#) that demonstrates how to provide custom disambiguation logic in your bot by implementing [IDisambiguateRouteHandler](#).

Next steps

The logic for using regular expressions and inspecting message contents can become complex, especially if your bot's conversational flow is open-ended. To help your bot handle a wider variety of textual and spoken input from users, you can use an intent recognition service like [LUIS](#) to add natural language understanding to your bot.

[Recognize intents and entities with LUIS](#)

Recognize intents and entities with LUIS

9/28/2017 • 8 min to read • [Edit Online](#)

This article uses the example of a bot for taking notes, to demonstrate how Language Understanding Intelligent Service ([LUIS](#)) helps your bot respond appropriately to natural language input. A bot detects what a user wants to do by identifying their **intent**. This intent is determined from spoken or textual input, or **utterances**. The intent maps utterances to actions that the bot takes, such as invoking a dialog. A bot may also need to extract **entities**, which are important words in utterances. Sometimes entities are required to fulfill an intent. In the example of a note-taking bot, the `Notes.Title` entity identifies the title of each note.

Create your LUIS app

To create the LUIS app, which is the web service you configure at [www.luis.ai](#) to provide the intents and entities to the bot, do the following steps:

1. Log in to [www.luis.ai](#) using your Cognitive Services API account. If you don't have an account, you can create a free account in the [Azure portal](#).
2. In the **My Apps** page, click **New App**, enter a name like Notes in the **Name** field, and choose **Bootstrap Key** in the **Key to use** field.
3. In the **Intents** page, click **Add prebuilt domain intents** and select **Notes.Create**, **Notes.Delete** and **Notes.ReadAloud**.
4. In the **Intents** page, click on the **None** intent. This intent is meant for utterances that don't correspond to any other intents. Enter an example of an utterance unrelated to weather, like "Turn off the lights"
5. In the **Entities** page, click **Add prebuilt domain entities** and select **Notes.Title**.
6. In the **Train & Test** page, train your app.
7. In the **Publish** page, click **Publish**. After successful publish, copy the **Endpoint URL** from the **Publish App** page, to use later in your bot's code. The URL has a format similar to this example:

```
https://westus.api.cognitive.microsoft.com/luis/v2.0/apps/3889f7d0-9501-45c8-be5f-8635975eea8b?subscription-key=67073e45132a459db515ca04cea325d3&timezoneOffset=0&verbose=true&q=
```

Create a note-taking bot integrated with the LUIS app

To create a bot that uses the LUIS app, you can first start with the sample code in [Create a bot with Node.js](#), and add code according to the instructions in the following sections.

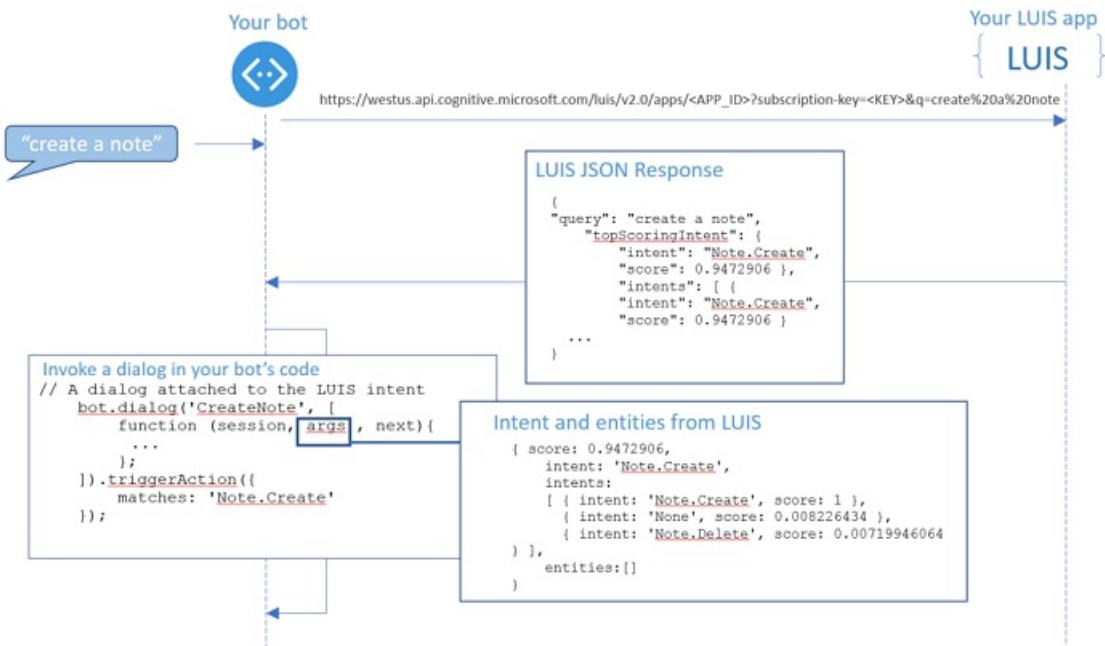
TIP

You can also find the sample code described in this article in the [Notes bot sample](#).

How LUIS passes intents and entities to your bot

The following diagram shows the sequence of events that happen after the bot receives an utterance from the user. First, the bot passes the utterance to the LUIS app and gets a JSON result from LUIS that contains intents and entities. Next, your bot automatically invokes any matching [Dialog](#) that your bot associates with the high-scoring intent in the LUIS result. The full details of the match, including the list of intents and entities that LUIS detected, are passed to the `args` parameter of the matching dialog.

```
<p align=center>
```



Create the bot

Create the bot that communicates with the Bot Framework Connector service by instantiating a [UniversalBot](#) object. The constructor takes a second parameter for a default message handler. This message handler sends a generic help message about the functionality that the note-taking bot provides, and initializes the `session.userData.notes` object for storing notes. You use `session.userData` so the notes are persisted for the user. Edit the code that creates the bot, so that the constructor looks like the following code:

```
// Create your bot with a function to receive messages from the user.
// This default message handler is invoked if the user's utterance doesn't
// match any intents handled by other dialogs.
var bot = new builder.UniversalBot(connector, function (session, args) {
  session.send("Hi... I'm the note bot sample. I can create new notes, read saved notes to you and delete
notes.");

  // If the object for storing notes in session.userData doesn't exist yet, initialize it
  if (!session.userData.notes) {
    session.userData.notes = {};
    console.log("initializing userData.notes in default message handler");
  }
});
```

Add a LuisRecognizer

The [LuisRecognizer](#) class calls the LUIS app. You initialize a [LuisRecognizer](#) using the **Endpoint URL** of the LUIS app that you copied from the [Publish App](#) page.

After you create the `UniversalBot`, add code to create the `LuisRecognizer` and add it to the bot:

```
// Add global LUIS recognizer to bot
var luisAppUrl = process.env.LUIS_APP_URL ||
'https://westus.api.cognitive.microsoft.com/luis/v2.0/apps/<YOUR_APP_ID>?subscription-key=<YOUR_KEY>';
bot.recognizer(new builder.LuisRecognizer(luisAppUrl));
```

Add dialogs

Now that the notes recognizer is set up to point to the LUIS app, you can add code for the dialogs. The `matches` option on the `triggerAction` attached to the dialog specifies the name of the intent. The recognizer runs each time the bot receives an utterance from the user. If the highest scoring intent that it detects matches a `triggerAction` bound to a dialog, the bot invokes that dialog.

Add the CreateNote dialog

Any entities in the utterance are passed to the dialog using the `args` parameter. The first step of the waterfall calls `EntityRecognizer.findEntity` to get the title of the note from any `Note.Title` entities in the LUIS response. If the LUIS app didn't detect a `Note.Title` entity, the bot prompts the user for the name of the note. The second step of the waterfall prompts for the text to include in the note. Once the bot has the text of the note, the third step uses `session.userData` to save the note in a `notes` object, using the title as the key. For more information on `session(userData)` see [Manage state data](#).

The following code for a CreateNote dialog handles the `Note.Create` intent.

```

// CreateNote dialog
bot.dialog('CreateNote', [
    function (session, args, next) {
        // Resolve and store any Note.Title entity passed from LUIS.
        var intent = args.intent;
        var title = builder.EntityRecognizer.findEntity(intent.entities, 'Note.Title');

        var note = session.dialogData.note = {
            title: title ? title.entity : null,
        };

        // Prompt for title
        if (!note.title) {
            builder.Prompts.text(session, 'What would you like to call your note?');
        } else {
            next();
        }
    },
    function (session, results, next) {
        var note = session.dialogData.note;
        if (results.response) {
            note.title = results.response;
        }

        // Prompt for the text of the note
        if (!note.text) {
            builder.Prompts.text(session, 'What would you like to say in your note?');
        } else {
            next();
        }
    },
    function (session, results) {
        var note = session.dialogData.note;
        if (results.response) {
            note.text = results.response;
        }

        // If the object for storing notes in session.userData doesn't exist yet, initialize it
        if (!session.userData.notes) {
            session.userData.notes = {};
            console.log("initializing session.userData.notes in CreateNote dialog");
        }
        // Save notes in the notes object
        session.userData.notes[note.title] = note;

        // Send confirmation to user
        session.endDialog('Creating note named "%s" with text "%s"',
            note.title, note.text);
    }
]).triggerAction({
    matches: 'Note.Create',
    confirmPrompt: "This will cancel the creation of the note you started. Are you sure?"
}).cancelAction('cancelCreateNote', "Note canceled.", {
    matches: /^(cancel|nevermind)/i,
    confirmPrompt: "Are you sure?"
});

```

If the LUIS app detects an intent that interrupts the `CreateNote` dialog, the `confirmPrompt` option on the dialog's `triggerAction` provides a prompt to confirm the interruption. For example, if the bot says "What would you like to call your note?", and the user replies "Actually, I want to delete a note instead", the bot prompts the user using the `confirmPrompt` message.

Add the DeleteNote dialog

In the `DeleteNote` dialog, the `triggerAction` matches the `Note.Delete` intent. As in the `CreateNote` dialog, the bot examines the `args` parameter for a title. If no title is detected, the bot prompts the user. The title is used to look up

the note to delete from `session.userData.notes`.

```
// Delete note dialog
bot.dialog('DeleteNote', [
    function (session, args, next) {
        if (noteCount(session.userData.notes) > 0) {
            // Resolve and store any Note.Title entity passed from LUIS.
            var title;
            var intent = args.intent;
            var entity = builder.EntityRecognizer.findEntity(intent.entities, 'Note.Title');
            if (entity) {
                // Verify that the title is in our set of notes.
                title = builder.EntityRecognizer.findBestMatch(session.userData.notes, entity.entity);
            }

            // Prompt for note name
            if (!title) {
                builder.Prompts.choice(session, 'Which note would you like to delete?',
                    session.userData.notes);
            } else {
                next({ response: title });
            }
        } else {
            session.endDialog("No notes to delete.");
        }
    },
    function (session, results) {
        delete session.userData.notes[results.response.entity];
        session.endDialog("Deleted the '%s' note.", results.response.entity);
    }
]).triggerAction({
    matches: 'Note.Delete'
}).cancelAction('cancelDeleteNote', 'Ok - canceled note deletion.', {
    matches: /^(cancel|nevermind)/i
});
```

The `DeleteNote` dialog uses the `noteCount` function to determine whether the `notes` object contains notes.

```
// Helper function to count the number of notes stored in session.userData.notes
function noteCount(notes) {

    var i = 0;
    for (var name in notes) {
        i++;
    }
    return i;
}
```

Add the ReadNote dialog

For reading a note, the `triggerAction` matches the `Note.ReadAloud` intent. The `session.userData.notes` object is passed as the third argument to `builder.Prompts.choice`, so that the prompt displays a list of notes to the user.

```

// Read note dialog
bot.dialog('ReadNote', [
  function (session, args, next) {
    if (noteCount(session.userData.notes) > 0) {

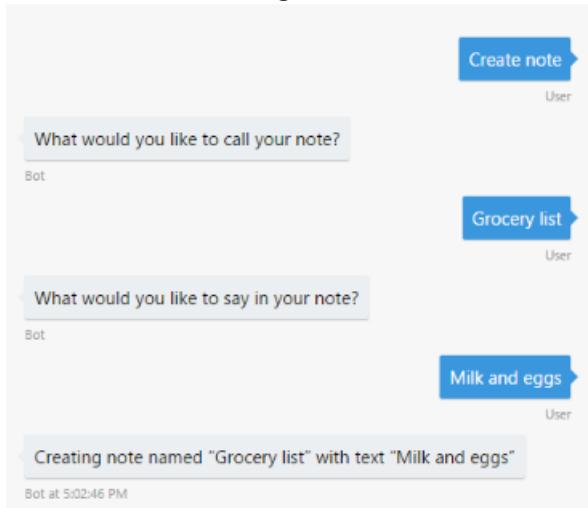
      // Resolve and store any Note.Title entity passed from LUIS.
      var title;
      var intent = args.intent;
      var entity = builder.EntityRecognizer.findEntity(intent.entities, 'Note.Title');
      if (entity) {
        // Verify it's in our set of notes.
        title = builder.EntityRecognizer.findBestMatch(session.userData.notes, entity.entity);
      }

      // Prompt for note name
      if (!title) {
        builder.Prompts.choice(session, 'Which note would you like to read?', session.userData.notes);
      } else {
        next({ response: title });
      }
    } else {
      session.endDialog("No notes to read.");
    }
  },
  function (session, results) {
    session.endDialog("Here's the '%s' note: '%s'.", results.response.entity,
    session.userData.notes[results.response.entity].text);
  }
]).triggerAction({
  matches: 'Note.ReadAloud'
}).cancelAction('cancelReadNote', "Ok.", {
  matches: /^(cancel|nevermind)/i
});

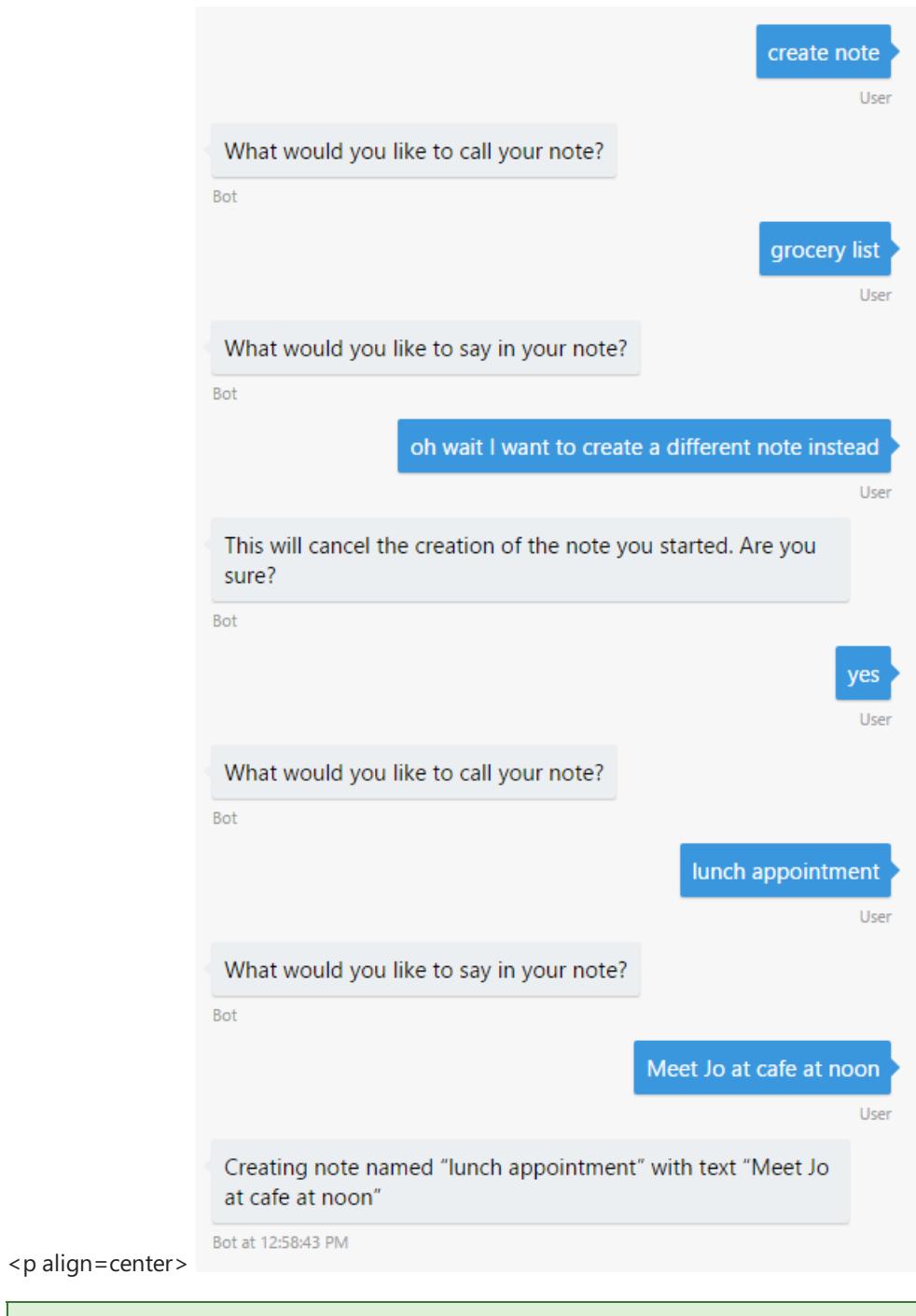
```

Try the bot

You can run the bot using the Bot Framework Emulator and tell it to create a note. <p align=center>



The use of `triggerAction` to match intents means that the bot can detect and react to intents for every utterance, even utterances that occur in the middle of the steps of a dialog. If the user is in the `CreateNote` dialog, but asks to create a different note before the dialog's conversation flow is complete, the bot detects the second `Note.Create` intent, and prompts the user to verify the interruption. See [Managing conversation flow](#) for more information on how `triggerAction` works with dialogs.



TIP

A LUIS app learns from example, so you can improve its performance by giving it more example utterances to train it. You can retrain your LUIS app without any modification to your bot's code. See [Add example utterances](#) and [train and test your LUIS app](#).

Next steps

From trying the bot, you can see that the recognizer can trigger interruption of the currently active dialog. Allowing and handling interruptions is a flexible design that accounts for what users really do. Learn more about the various actions you can associate with a recognized intent.

[Handle user actions](#)

Handle user and conversation events

9/27/2017 • 1 min to read • [Edit Online](#)

This article demonstrates how your bot can handle events such as a user joining a conversation, adding a bot to a contacts list, or saying **Goodbye** when a bot is being removed from a conversation.

Greet a user on conversation join

The Bot Framework provides the [conversationUpdate](#) event for notifying your bot whenever a member joins or leaves a conversation. A conversation member can be a user or a bot.

The following code snippet allows the bot to greet new member(s) to a conversation or say **Goodbye** if it is being removed from the conversation.

```
bot.on('conversationUpdate', function (message) {
  if (message.membersAdded && message.membersAdded.length > 0) {
    // Say hello
    var isGroup = message.address.conversation.isGroup;
    var txt = isGroup ? "Hello everyone!" : "Hello...";
    var reply = new builder.Message()
      .address(message.address)
      .text(txt);
    bot.send(reply);
  } else if (message.membersRemoved) {
    // See if bot was removed
    var botId = message.address.bot.id;
    for (var i = 0; i < message.membersRemoved.length; i++) {
      if (message.membersRemoved[i].id === botId) {
        // Say goodbye
        var reply = new builder.Message()
          .address(message.address)
          .text("Goodbye");
        bot.send(reply);
        break;
      }
    }
  }
});
```

Acknowledge add to contacts list

The [contactRelationUpdate](#) event notifies your bot that a user has added you to their contacts list.

```
bot.on('contactRelationUpdate', function (message) {
  if (message.action === 'add') {
    var name = message.user ? message.user.name : null;
    var reply = new builder.Message()
      .address(message.address)
      .text("Hello %s... Thanks for adding me.", name || 'there');
    bot.send(reply);
  }
});
```

Add a first-run dialog

Since the **conversationUpdate** and the **contactRelationUpdate** event are not supported by all channels, a universal way to greet a user who joins a conversation is to add a first-run dialog.

In the following example we've added a function that triggers the dialog any time we've never seen a user before. You can customize the way an action is triggered by providing an [onFindAction](#) handler for your action.

```
// Add first run dialog
bot.dialog('firstRun', function (session) {
    session.userData.firstRun = true;
    session.send("Hello...").endDialog();
}).triggerAction({
    onFindAction: function (context, callback) {
        // Only trigger if we've never seen user before
        if (!context.userData.firstRun) {
            // Return a score of 1.1 to ensure the first run dialog wins
            callback(null, 1.1);
        } else {
            callback(null, 0.0);
        }
    }
});
```

You can also customize what an action does after its been triggered by providing an [onSelectAction](#) handler. For trigger actions you can provide an [onInterrupted](#) handler to intercept an interruption before it occurs. For more information, see [Handle user actions](#)

Additional resources

- [conversationUpdate](#)
- [contactRelationUpdate](#)

Support localization

9/6/2017 • 4 min to read • [Edit Online](#)

Bot Builder includes a rich localization system for building bots that can communicate with the user in multiple languages. All of your bot's prompts can be localized using JSON files stored in your bot's directory structure. If you're using a system like LUIS to perform natural language processing, you can configure your [LuisRecognizer](#) with a separate model for each language your bot supports and the SDK will automatically select the model that matches the user's preferred locale.

Determine the locale by prompting the user

The first step to localizing your bot is adding the ability to identify the user's preferred language. The SDK provides a [session.preferredLocale\(\)](#) method to both save and retrieve this preference on a per-user basis. The following example is a dialog to prompt the user for their preferred language and then save their choice.

```
bot.dialog('/localePicker', [
  function (session) {
    // Prompt the user to select their preferred locale
    builder.Prompts.choice(session, "What's your preferred language?", 'English|Español|Italiano');
  },
  function (session, results) {
    // Update preferred locale
    var locale;
    switch (results.response.entity) {
      case 'English':
        locale = 'en';
        break;
      case 'Español':
        locale = 'es';
        break;
      case 'Italiano':
        locale = 'it';
        break;
    }
    session.preferredLocale(locale, function (err) {
      if (!err) {
        // Locale files loaded
        session.endDialog(`Your preferred language is now ${results.response.entity}`);
      } else {
        // Problem loading the selected locale
        session.error(err);
      }
    });
  }
]);
```

Determine the locale by using analytics

Another way to determine the user's locale is to use a service like the [Text Analytics API](#) to automatically detect the user's language based upon the text of the message they sent.

The code snippet below illustrates how you can incorporate this service into your own bot.

```

var request = require('request');

bot.use({
  receive: function (event, next) {
    if (event.text && !event.textLocale) {
      var options = {
        method: 'POST',
        url: 'https://westus.api.cognitive.microsoft.com/text/analytics/v2.0/languages?numberOfLanguagesToDetect=1',
        body: { documents: [{ id: 'message', text: event.text }] },
        json: true,
        headers: {
          'Ocp-Apim-Subscription-Key': '<YOUR API KEY>'
        }
      };
      request(options, function (error, response, body) {
        if (!error && body) {
          if (body.documents && body.documents.length > 0) {
            var languages = body.documents[0].detectedLanguages;
            if (languages && languages.length > 0) {
              event.textLocale = languages[0].iso6391Name;
            }
          }
        }
        next();
      });
    } else {
      next();
    }
  }
});

```

Once you add the above code snippet to your bot, calling `session.preferredLocale()` will automatically return the detected language. The search order for `preferredLocale()` is as follows:

1. Locale saved by calling `session.preferredLocale()`. This value is stored in `session.userData['BotBuilder.Data.PreferredLocale']`.
2. Detected locale assigned to `session.message.textLocale`.
3. The configured default locale for the bot (e.g: English ('en')).

You can configure the bot's default locale using its constructor:

```

var bot = new builder.UniversalBot(connector, {
  localizerSettings: {
    defaultLocale: "es"
  }
});

```

Localize prompts

The default localization system for the Bot Builder SDK is file-based and allows a bot to support multiple languages using JSON files stored on disk. By default, the localization system will search for the bot's prompts in the **`./locale//index.json`** file where is a valid [IETF language tag](#) representing the preferred locale for which to find prompts.

The following screenshot shows the directory structure for a bot that supports three languages: English, Italian, and Spanish.

```
1  {
2      "greeting": ["Welcome to the localization sample bot!"],
3      "instructions": "This demo will first ask you to select your preferred language.",
4      "locale_prompt": "What's your preferred language?",
5      "locale_updated": "Your preferred language is now set to %s",
6      "text_prompt": "Prompts.text(): enter some text and see what happens!",
7      "number_prompt": "Prompts.number(): enter a number and see what happens!",
8      "listStyle_prompt": "Prompts.choice() supports a range of options",
9      "choice_prompt": "Prompts.choice(): choose an option from the list",
10     "choice_options": "Option A|Option B|Option C",
11     "confirm_prompt": "Prompts.confirm(): enter 'yes' or 'no'",
12     "time_prompt": "Prompts.time(): the built-in time picker",
13     "input_response": "You entered '%s', did I get that right?",
14     "choice_response": "You chose '%s', nice!",
15     "time_response": "JSON for time entered: %s",
16     "demo_finished": ["That's the end of the demo.", "See you next time!"]
17 }
```

The structure of the file is a simple JSON map of message IDs to localized text strings. If the value is an array instead of a string, one prompt from the array is chosen at random when that value is retrieved using `session.localizer.gettext()`.

The bot automatically retrieves the localized version of a message if you pass the message ID in a call to `session.send()` instead of language-specific text:

```
var bot = new builder.UniversalBot(connector, [
    function (session) {
        session.send("greeting");
        session.send("instructions");
        session.beginDialog('/localePicker');
    },
    function (session) {
        builder.Prompts.text(session, "text_prompt");
    }
]);
```

Internally, the SDK calls `session.preferredLocale()` to get the user's preferred locale and then uses that in a call to `session.localizer.gettext()` to map the message ID to its localized text string. There are times where you may need to manually call the localizer. For instance, the enum values passed to `Prompts.choice()` are never automatically localized so you may need to manually retrieve a localized list prior to calling the prompt:

```
var options = session.localizer.gettext(session.preferredLocale(), "choice_options");
builder.Prompts.choice(session, "choice_prompt", options);
```

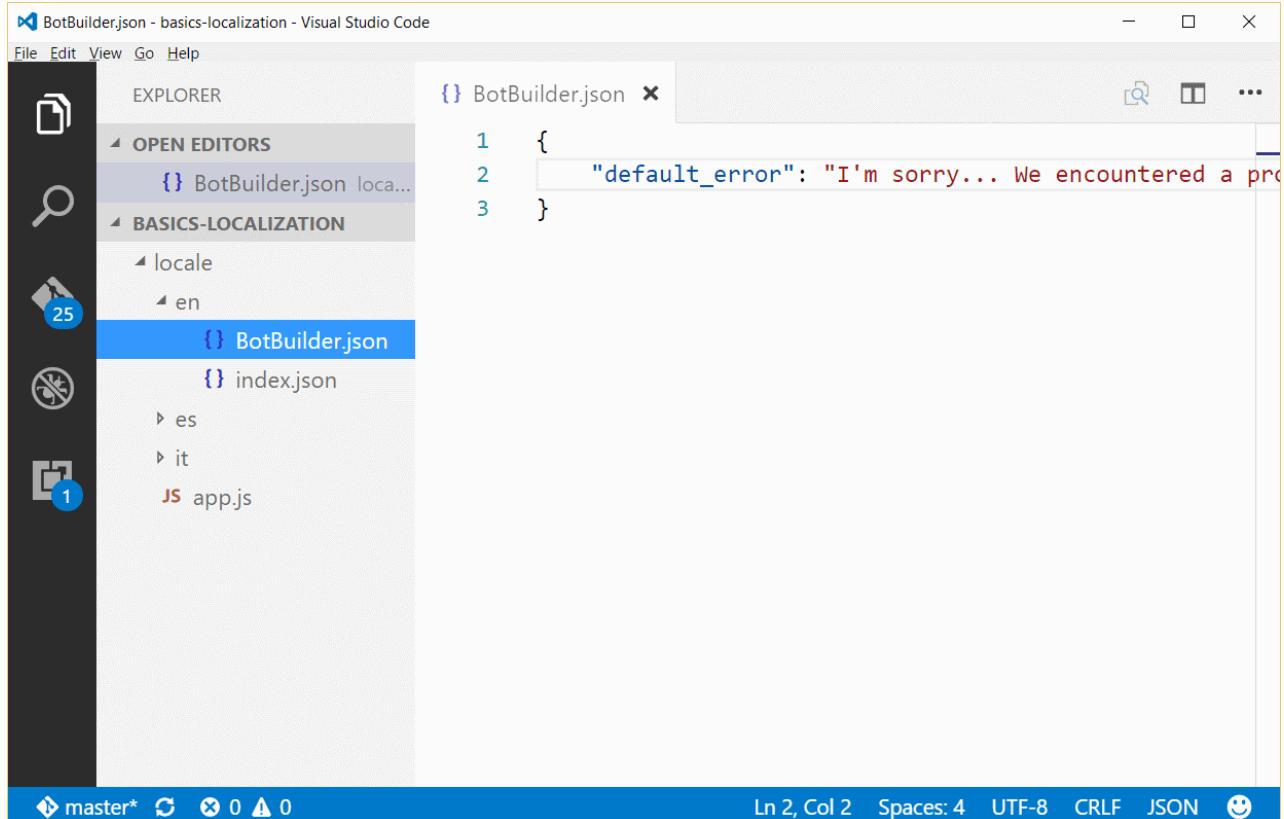
The default localizer searches for a message ID across multiple files and if it can't find an ID (or if no localization files were provided) it will simply return the text of ID, making the use of localization files transparent and optional. Files are searched in the following order:

1. The **index.json** file under the locale returned by `session.preferredLocale()` is searched.
2. If the locale included an optional subtag like **en-US** then the root tag of **en** is searched.
3. The bot's configured default locale is searched.

Use namespaces to customize and localize prompts

The default localizer supports the namespacing of prompts to avoid collisions between message IDs. Your bot can override namespaced prompts to customize or reword the prompts from another namespace. You can leverage this capability to customize the SDK's built-in messages, letting you either add support for additional languages or to simply reword the SDK's current messages. For instance, you can change the SDK's default error message by simply adding a file called **BotBuilder.json** to your bot's locale directory and then adding an entry for the

`default_error` message ID:



A screenshot of the Visual Studio Code interface. The title bar says "BotBuilder.json - basics-localization - Visual Studio Code". The menu bar includes File, Edit, View, Go, and Help. The Explorer sidebar shows a tree structure with "OPEN EDITORS" and "BASICS-LOCALIZATION" sections. Under "BASICS-LOCALIZATION", there is a "locale" section with "en", "es", and "it" sub-folders. Inside "en", the "BotBuilder.json" file is selected and highlighted in blue. The main editor area displays the JSON content:

```
1  {
2      "default_error": "I'm sorry... We encountered a problem with your request."
3  }
```

The status bar at the bottom shows "Ln 2, Col 2" and "Spaces: 4" and other file details.

Additional resources

To learn about how to localize a recognizer, see [Recognizing intent](#).

Use the backchannel mechanism

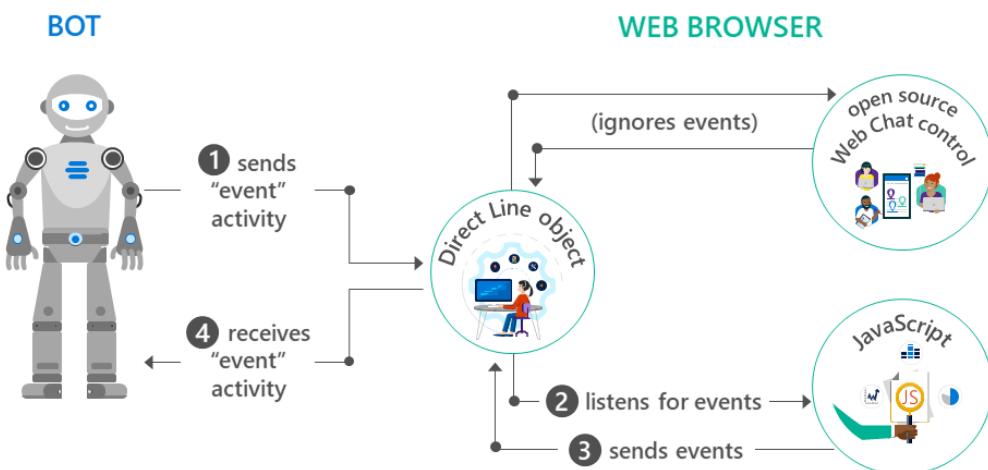
9/6/2017 • 2 min to read • [Edit Online](#)

The [open source web chat control](#) communicates with bots by using the [Direct Line API](#), which allows `activities` to be sent back and forth between client and bot. The most common type of activity is `message`, but there are other types as well. For example, the activity type `typing` indicates that a user is typing or that the bot is working to compile a response.

You can use the backchannel mechanism to exchange information between client and bot without presenting it to the user by setting the activity type to `event`. The web chat control will automatically ignore any activities where `type="event"`.

Walk through

The open source web chat control accesses the Direct Line API by using a JavaScript class called [DirectLineJS](#). The control can either create its own instance of Direct Line, or it can share one with the hosting page. If the control shares an instance of Direct Line with the hosting page, both the control and the page will be capable of sending and receiving activities. The following diagram shows the high-level architecture of a website that supports bot functionality by using the open source web (chat) control and the Direct Line API.



Sample code

In this example, the bot and web page will use the backchannel mechanism to exchange information that is invisible to the user. The bot will request that the web page change its background color, and the web page will notify the bot when the user clicks a button on the page.

NOTE

The code snippets in this article originate from the [backchannel sample](#) and the [backchannel bot](#).

Client-side code

First, the web page creates a **DirectLine** object.

```
var botConnection = new BotChat.DirectLine(...);
```

Then, it shares the **DirectLine** object when creating the WebChat instance.

```
BotChat.App({
  botConnection: botConnection,
  user: user,
  bot: bot
}, document.getElementById("BotChatGoesHere"));
```

When the user clicks a button on the web page, the web page posts an activity of type "event" to notify the bot that the button was clicked.

```
const postButtonMessage = () => {
  botConnection
    .postActivity({type: "event", value: "", from: {id: "me"}, name: "buttonClicked"})
    .subscribe(id => console.log("success"));
}
```

TIP

Use the attributes `name` and `value` to communicate any information that the bot may need in order to properly interpret and/or respond to the event.

Finally, the web page also listens for a specific event from the bot. In this example, the web page listens for an activity where `type="event"` and `name="changeBackground"`. When it receives this type of activity, it changes the background color of the web page to the `value` specified by the activity.

```
botConnection.activity$
  .filter(activity => activity.type === "event" && activity.name === "changeBackground")
  .subscribe(activity => changeBackgroundColor(activity.value))
```

Server-side code

The [backchannel bot](#) creates an event by using a helper function.

```
var bot = new builder.UniversalBot(connector, [
  function (session) {
    var reply = createEvent("changeBackground", session.message.text, session.message.address);
    session.endDialog(reply);
  }
);

const createEvent = (eventName, value, address) => {
  var msg = new builder.Message().address(address);
  msg.data.type = "event";
  msg.data.name = eventName;
  msg.data.value = value;
  return msg;
}
```

Likewise, the bot also listens for events from the client. In this example, if the bot receives an event with `name="buttonClicked"`, it will send a message to the user to say "I see that you clicked a button."

```
bot.on("event", function (event) {
  var msg = new builder.Message().address(event.address);
  msg.data.textLocale = "en-us";
  if (event.name === "buttonClicked") {
    msg.data.text = "I see that you clicked a button.";
  }
  bot.send(msg);
})
```

Additional resources

- [Direct Line API](#)
- [Microsoft Bot Framework WebChat control](#)
- [Backchannel sample](#)
- [Backchannel bot](#)

Request payment

7/26/2017 • 9 min to read • [Edit Online](#)

If your bot enables users to purchase items, it can request payment by including a special type of button within a [rich card](#). This article describes how to send a payment request using the Bot Builder SDK for Node.js.

Prerequisites

Before you can send a payment request using the Bot Builder SDK for Node.js, you must complete these prerequisite tasks.

Register and configure your bot

1. [Register](#) your bot with the Bot Framework.
2. Set the `MICROSOFT_APP_ID` and `MICROSOFT_APP_PASSWORD` environment variables to the app ID and password values that were generated for your bot during the registration process.

Create and configure merchant account

1. [Create and activate a Stripe account if you don't have one already.](#)
2. [Sign in to Seller Center with your Microsoft account.](#)
3. Within Seller Center, connect your account with Stripe.
4. Within Seller Center, navigate to the Dashboard and copy the value of **MerchantID**.
5. Update the `PAYMENTS_MERCHANT_ID` environment variable to the value that you copied from the Seller Center Dashboard.

Payment process overview

The payment process comprises three distinct parts:

1. The bot sends a payment request.
2. The user signs in with a Microsoft account to provide payment, shipping, and contact information. Callbacks are sent to the bot to indicate when the bot needs to perform certain operations (update shipping address, update shipping option, complete payment).
3. The bot processes the callbacks that it receives, including shipping address update, shipping option update, and payment complete.

Your bot must implement only step one and step three of this process; step two takes place outside the context of your bot.

Payment Bot sample

The [Payment Bot](#) sample provides an example of a bot that sends a payment request by using Node.js. To see this sample bot in action, you can [try it out in web chat](#), [add it as a Skype contact](#), or download the payment bot sample and run it locally using the Bot Framework Emulator.

NOTE

To complete the end-to-end payment process using the **Payment Bot** sample in web chat or Skype, you must specify a valid credit card or debit card within your Microsoft account (i.e., a valid card from a U.S. card issuer). Your card will not be charged and the card's CVV will not be verified, because the **Payment Bot** sample runs in test mode (i.e., `PAYMENTS_LIVEMODE` is set to `false` in `.env`).

The next few sections of this article describe the three parts of the payment process, in the context of the **Payment Bot** sample.

Requesting payment

Your bot can request payment from a user by sending a message that contains a [rich card](#) with a button that specifies `type` of "payment". This code snippet from the **Payment Bot** sample creates a message that contains a Hero card with a **Buy** button that the user can click (or tap) to initiate the payment process.

```
var bot = new builder.UniversalBot(connector, (session) => {

  catalog.getPromotedItem().then(product => {

    // Store userId for later, when reading relatedTo to resume dialog with the receipt.
    var cartId = product.id;
    session.conversationData[CartIdKey] = cartId;
    session.conversationData[cartId] = session.message.address.user.id;

    // Create PaymentRequest obj based on product information.
    var paymentRequest = createPaymentRequest(cartId, product);

    var buyCard = new builder.HeroCard(session)
      .title(product.name)
      .subtitle(util.format('%s %s', product.currency, product.price))
      .text(product.description)
      .images([
        new builder.CardImage(session).url(product.imageUrl)
      ])
      .buttons([
        new builder.CardAction(session)
          .title('Buy')
          .type(payments.PaymentActionType)
          .value(paymentRequest)
      ]);
    session.send(new builder.Message(session)
      .addAttachment(buyCard));
  });
});
```

In this example, the button's type is specified as `payments.PaymentActionType`, which the app defines as "payment". The button's value is populated by the `createPaymentRequest` function, which returns a `PaymentRequest` object that contains information about supported payment methods, details, and options. For more information about implementation details, see [app.js](#) within the **Payment Bot** sample.

This screenshot shows the Hero card (with **Buy** button) that's generated by the code snippet above.

paymentsample +

← → ⌂ | 🔒 webchat.botframework.com/embed/paymentsample?si= Chat



Scott Gu - Favorite Shirt
USD 1.99
Shiny red, ready to rock on Keynotes
Buy

paymentsample at 3:33:59 PM

Type your message... 

IMPORTANT

Any user that has access to the **Buy** button may use it to initiate the payment process. Within the context of a group conversation, it is not possible to designate a button for use by only a specific user.

User experience

When a user clicks the **Buy** button, he or she is directed to the payment web experience to provide all required payment, shipping, and contact information via their Microsoft account.

Confirm and Pay

Pay with

Ship to Select shipping address

Shipping options

Email receipt to

Phone number

Total (USD) [Show details](#) \$1.99*

* - Indicates the cost isn't final

Pay

HTTP callbacks

HTTP callbacks will be sent to your bot to indicate that it should perform certain operations. Each callback will be an event that contains these property values:

PROPERTY	VALUE
<code>type</code>	invoke
<code>name</code>	Indicates the type of operation that the bot should perform (e.g., shipping address update, shipping option update, payment complete).
<code>value</code>	The request payload in JSON format.
<code>relatesTo</code>	Describes the channel and user that are associated with the payment request.

NOTE

`invoke` is a special event type that is reserved for use by the Microsoft Bot Framework. The sender of an `invoke` event will expect your bot to acknowledge the callback by sending an HTTP response.

Processing callbacks

When your bot receives a callback, it should verify that the information specified in the callback is valid and acknowledge the callback by sending an HTTP response.

Shipping Address Update and Shipping Option Update callbacks

When receiving a Shipping Address Update or a Shipping Option Update callback, your bot will be provided with the current state of the payment details from the client in the event's `value` property. As a merchant, you should treat these callbacks as static, given input payment details you will calculate some output payment details and fail if the input state provided by the client is invalid for any reason. If the bot determines the given information is valid as-is, simply send HTTP status code `200 OK` along with the unmodified payment details. Alternatively, the bot may send HTTP status code `200 OK` along with an updated payment details that should be applied before the order can be processed. In some cases, your bot may determine that the updated information is invalid and the order cannot be processed as-is. For example, the user's shipping address may specify a country to which the product supplier does not ship. In that case, the bot may send HTTP status code `200 OK` and a message populating the `error` property of the payment details object. Sending any HTTP status code in the `400` or `500` range to will result in a generic error for the customer.

Payment Complete callbacks

When receiving a Payment Complete callback, your bot will be provided with a copy of the initial, unmodified payment request as well as the payment response objects in the event's `value` property. The payment response object will contain the final selections made by the customer along with a payment token. Your bot should take the opportunity to recalculate the final payment request based on the initial payment request and the customer's final selections. Assuming the customer's selections are determined to be valid, the bot should verify the amount and currency in the payment token header to ensure that they match the final payment request. If the bot decides to charge the customer it should only charge the amount in the payment token header as this is the price the customer confirmed. If there is a mismatch between the values that the bot expects and the values that it received in the Payment Complete callback, it can fail the payment request by sending HTTP status code `200 OK` along with setting the `result` field to `failure`.

In addition to verifying payment details, the bot should also verify that the order can be fulfilled, before it initiates payment processing. For example, it may want to verify that the item(s) being purchased are still available in stock. If the values are correct and your payment processor has successfully charged the payment token, then the bot should respond with HTTP status code `200 OK` along with setting the `result` field to `success` in order for the payment web experience to display the payment confirmation. The payment token that the bot receives can only be used once, by the merchant that requested it, and must be submitted to Stripe (the only payment processor that the Bot Framework currently supports). Sending any HTTP status code in the `400` or `500` range to will result in a generic error for the customer.

This code snippet from the **Payment Bot** sample processes the callbacks that the bot receives.

```
connector.onInvoke((invoke, callback) => {
    console.log('onInvoke', invoke);

    // This is a temporary workaround for the issue that the channelId for "webchat" is mapped to "directline"
    // in the incoming RelatesTo object
    invoke.relatesTo.channelId = invoke.relatesTo.channelId === 'directline' ? 'webchat' :
    invoke.relatesTo.channelId;

    var storageCtx = {
        address: invoke.relatesTo,
        persistConversationData: true,
        conversationId: invoke.relatesTo.conversation.id
    };

    connector.getData(storageCtx, (err, data) => {
        var cartId = data.conversationData[CartIdKey];
        if (!invoke.relatesTo.user && cartId) {
            // Bot keeps the userId in context.ConversationData[cartId]
```

```

var userId = data.conversationData[cartId];
invoke.relatesTo.useAuth = true;
invoke.relatesTo.user = { id: userId };
}

// Continue based on PaymentRequest event.
var paymentRequest = null;
switch (invoke.name) {
  case payments.Operations.UpdateShippingAddressOperation:
  case payments.Operations.UpdateShippingOptionOperation:
    paymentRequest = invoke.value;

    // Validate address AND shipping method (if selected).
    checkout
      .validateAndCalculateDetails(paymentRequest, paymentRequest.shippingAddress,
paymentRequest.shippingOption)
      .then(updatedPaymentRequest => {
        // Return new paymentRequest with updated details.
        callback(null, updatedPaymentRequest, 200);
      })
      .catch(err => {
        // Return error to onInvoke handler.
        callback(err);
        // Send error message back to user.
        bot.beginDialog(invoke.relatesTo, 'checkout_failed', {
          errorMessage: err.message
        });
      });
    });

break;

case payments.Operations.PaymentCompleteOperation:
  var paymentRequestComplete = invoke.value;
  paymentRequest = paymentRequestComplete.paymentRequest;
  var paymentResponse = paymentRequestComplete.paymentResponse;

  // Validate address AND shipping method.
  checkout
    .validateAndCalculateDetails(paymentRequest, paymentResponse.shippingAddress,
paymentResponse.shippingOption)
    .then(updatedPaymentRequest =>
      // Process payment.
      checkout
        .processPayment(updatedPaymentRequest, paymentResponse)
        .then(chargeResult => {
          // Return success.
          callback(null, { result: "success" }, 200);
          // Send receipt to user.
          bot.beginDialog(invoke.relatesTo, 'checkout_receipt', {
            paymentRequest: updatedPaymentRequest,
            chargeResult: chargeResult
          });
        })
        .catch(err => {
          // Return error to onInvoke handler.
          callback(err);
          // Send error message back to user.
          bot.beginDialog(invoke.relatesTo, 'checkout_failed', {
            errorMessage: err.message
          });
        });
    );
  });

break;
}
);
}
);

```

In this example, the bot examines the `name` property of the incoming event to identify the type of operation it

needs to perform, and then calls the appropriate method(s) to process the callback. For more information about implementation details, see [app.js](#) within the [Payment Bot](#) sample.

Testing a payment bot

To fully test a bot that requests payment, you can [deploy](#) it to the cloud. After you have deployed your bot to the cloud, [configure](#) it to run on channels that support Bot Framework payments, like Web Chat and Skype.

Alternatively, you can test your bot locally using the [Bot Framework Emulator](#).

TIP

Callbacks are sent to your bot when a user changes data or clicks **Pay** during the payment web experience. Therefore, you can test your bot's ability to receive and process callbacks by interacting with the payment web experience yourself.

In the [Payment Bot](#) sample, the `PAYMENTS_LIVEMODE` environment variable in `.env` determines whether Payment Complete callbacks will contain emulated payment tokens or real payment tokens. If `PAYMENTS_LIVEMODE` is set to `false`, a header is added to the bot's outbound payment request to indicate that the bot is in test mode, and the Payment Complete callback will contain an emulated payment token that cannot be charged. If `PAYMENTS_LIVEMODE` is set to `true`, the header which indicates that the bot is in test mode is omitted from the bot's outbound payment request, and the Payment Complete callback will contain a real payment token that the bot will submit to Stripe for payment processing. This will be a real transaction that results in charges to the specified payment instrument.

Additional resources

- [Payment Bot sample](#)
- [Add rich card attachments to messages](#)
- [Web Payments at W3C](#)

Create data-driven experiences with Azure Search

8/7/2017 • 4 min to read • [Edit Online](#)

You can add [Azure Search](#) to your bot to help the user navigate large amounts of content and create a data-driven exploration experience for users of your bot.

Azure Search is an Azure service that offers keyword search, built-in linguistics, custom scoring, faceted navigation and more. Azure Search can also index content from various sources, including Azure SQL DB, DocumentDB, Blob Storage, and Table Storage. It supports "push" indexing for other sources of data, and it can open PDFs, Office documents, and other formats containing unstructured data. Once collected, the content goes into an Azure Search index, which the bot can then query.

Install dependencies

From a command prompt, navigate to your bot's project directory and install the following modules with the Node Package Manager (NPM):

- [bluebird](#)
- [lodash](#)
- [request](#)

Prerequisites

The following are **required**:

- Have an Azure subscription and an Azure Search Primary Key. You can find this in the Azure portal.
- Copy the [SearchDialogLibrary](#) library to your bot's project directory. This library contains general dialogs for the user to search, but can be customized as needed to suit your bot.
- Copy the [SearchProviders](#) library to your bot's project directory. This library contains all of the components required to create a request and submit it to Azure Search.

Connect to the Azure Service

In your bot's main program file (e.g.: app.js), create the reference paths to the two libraries you installed previously.

```
var SearchLibrary = require('./SearchDialogLibrary');
var AzureSearch = require('./SearchProviders/azure-search');
```

Add the following sample code to your bot. In the `AzureSearch` object, pass in your own Azure Search settings into the `.create` method. At run time, this will bind your bot to the Azure Search service and wait for a completed user query in the form of a `Promise`.

```
// Azure Search
var azureSearchClient = AzureSearch.create('Your-Azure-Search-Service-Name', 'Your-Azure-Search-Primary-Key',
  'Your-Azure-Search-Service-Index');
var ResultsMapper = SearchLibrary.defaultResultsMapper(ToSearchHit);
```

The `azureSearchClient` reference creates the Azure Search model, which passes the Azure Service's authorization settings from the bot's `.env` settings. `ResultsMapper` parses the Azure response object and maps the data as we

define in `ToSearchHit` method. For an implementation example of this method, see [After Azure Search responds](#).

Register the search library

You can customize your search dialogs directly in the `SearchLibrary` module itself. The `SearchLibrary` performs most of the heavy lifting, including making the call to Azure Search.

Add the following code in your bot's main program file to register the Search Dialogs library with your bot.

```
bot.library(SearchLibrary.create({
  multipleSelection: true,
  search: function (query) { return azureSearchClient.search(query).then(ResultsMapper); },
  refiners: ['refiner1', 'refiner2', 'refiner3'], // customize your own refiners
  refineFormatter: function (refiners) {
    return _.zipObject(
      refiners.map(function (r) { return 'By ' + _.capitalize(r); }),
      refiners);
  }
}));
```

The `SearchLibrary` not only stores all of your search related dialogs, but also takes the user query to submit to Azure Search. You will need to define your own refiners in the `refiners` array to specify entities you wish to allow your user to narrow or filter their search results.

Create a search dialog

You may choose to structure your dialogs however you want. The only requirement to setting up an Azure Search dialog is to invoke the `.begin` method from the `SearchLibrary` object, passing in the `session` object generated by the Bot Builder SDK.

```
function (session) {
  // Trigger Azure Search dialogs
  SearchLibrary.begin(session);
},
function (session, args) {
  // Process selected search results
  session.send(
    'Search Completed!',
    args.selection.map( )); // format your response
}
```

For more information about dialogs, see [Manage a conversation with dialogs](#).

After Azure Search responds

Once a successful Azure Search resolves, you now need to store the data you want from the response object, and display it in a meaningful way to the user.

TIP

Consider including the `util module`. It will help you format and map the response from Azure Search.

In your bot's main program file, create a `ToSearchHit` method. This method returns an object which formats the relevant data you need from the Azure Response. The following code shows how you can define your own parameters in the `ToSearchHit` method.

```

function ToSearchHit(azureResponse) {
    return {
        // define your own parameters
        key: azureResponse.id,
        title: azureResponse.title,
        description: azureResponse.description,
        imageUrl: azureResponse.thumbnail
    };
}

```

After this is done, all you need to do is display the data to the user.

In the **index.js** file of the **SearchDialogLibrary** project, the `searchHitAsCard` method parses each response from the Azure Search and creates a new card object to display to the user. The fields you defined in the `ToSearchHit` method from your bot's main program file needs to synced with the properties in the `searchHitAsCard` method.

The following shows how and where your defined parameters from the `ToSearchHit` method are used to build a card attachment UI to render to the user.

```

function searchHitAsCard(showSave, searchHit) {
    var buttons = showSave
        ? [new builder.CardAction().type('imBack').title('Save').value(searchHit.key)]
        : [];

    var card = new builder.HeroCard()
        .title(searchHit.title)
        .buttons(buttons);

    if (searchHit.description) {
        card.subtitle(searchHit.description);
    }

    if (searchHit.imageUrl) {
        card.images([new builder.CardImage().url(searchHit.imageUrl)]);
    }

    return card;
}

```

Sample code

For two complete samples that show how to support Azure Search with bots using the Bot Builder SDK for Node.js, see the [Real Estate Bot sample](#) or [Job Listing Bot sample](#) in GitHub.

Additional resources

- [Azure Search](#)
- [Node Util](#)
- [Dialogs](#)

Bot Framework REST APIs

7/26/2017 • 1 min to read • [Edit Online](#)

The Bot Framework REST APIs enable you to build bots that exchange messages with channels configured in the [Bot Framework Portal](#), store and retrieve state data, and connect your own client applications to your bots. All Bot Framework services use industry-standard REST and JSON over HTTPS.

Build a bot

You can create a bot with any programming language by using the Bot Connector service to exchange messages with channels configured in the Bot Framework Portal. You can use the Bot State service to store and retrieve state data that is related to conversations that your bot conducts using the Bot Connector service.

TIP

The Bot Framework provides client libraries that can be used to build bots in either C# or Node.js. To build a bot using C#, use the [Bot Builder SDK for C#](#). To build a bot using Node.js, use the [Bot Builder SDK for Node.js](#).

To learn more about building bots using the Bot Connector service and the Bot State service, see [Key concepts](#).

Build a client

You can enable your own client application to communicate with your bot by using the Direct Line API. The Direct Line API implements an authentication mechanism that uses standard secret/token patterns and provides a stable schema, even if your bot changes its protocol version. To learn more about using the Direct Line API to enable communication between a client and your bot, see [Key concepts](#).

Key concepts

8/7/2017 • 2 min to read • [Edit Online](#)

You can use the Bot Connector service and the Bot State service to communicate with users across multiple channels such as Skype, Email, Slack, and more. This article introduces key concepts in the Bot Connector service and Bot State service.

Bot Connector service

The Bot Connector service enables your bot to exchange messages with channels configured in the [Bot Framework Portal](#). It uses industry-standard REST and JSON over HTTPS and enables authentication with JWT Bearer tokens. For detailed information about how to use the Bot Connector service, see [Authentication](#) and the remaining articles in this section.

Activity

The Bot Connector service exchanges information between bot and channel (user) by passing an [Activity](#) object. The most common type of activity is **message**, but there are other activity types that can be used to communicate various types of information to a bot or channel. For details about Activities in the Bot Connector service, see [Activities overview](#).

Bot State service

The Bot State service enables your bot to store and retrieve state data that is associated with a user, a conversation, or a specific user within the context of a specific conversation. It uses industry-standard REST and JSON over HTTPS and enables authentication with JWT Bearer tokens. Any data that you save by using the Bot State service will be stored in Azure and encrypted at rest.

The Bot State service is only useful in conjunction with the Bot Connector service. That is, it can only be used to store state data that is related to conversations that your bot conducts using the Bot Connector service. For detailed information about how to use the Bot State service, see [Authentication](#) and [Manage state data](#).

Authentication

Both the Bot Connector service and the Bot State service enable authentication with JWT Bearer tokens. For detailed information about how to authenticate outbound requests that your bot sends to the Bot Framework, how to authenticate inbound requests that your bot receives from the Bot Framework, and more, see [Authentication](#).

Client libraries

The Bot Framework provides client libraries that can be used to build bots in either C# or Node.js.

- To build a bot using C#, use the [Bot Builder SDK for C#](#).
- To build a bot using Node.js, use the [Bot Builder SDK for Node.js](#).

In addition to modeling the Bot Connector service and the Bot State service, each Bot Builder SDK also provides a powerful system for building dialogs that encapsulate conversational logic, built-in prompts for simple things such as Yes/No, strings, numbers, and enumerations, built-in support for powerful AI frameworks such as [LUIS](#), and more.

NOTE

As an alternative to using the C# SDK or Node.js SDK, you can generate your own client library in the language of your choice by using the [Bot Connector Swagger file](#) and the [Bot State Swagger file](#).

Additional resources

Learn more about building bots using the Bot Connector service and Bot State service by reviewing articles throughout this section, beginning with [Authentication](#). If you encounter problems or have suggestions regarding the Bot Connector service or Bot State service, see [Support](#) for a list of available resources.

Authentication

10/12/2017 • 13 min to read • [Edit Online](#)

Your bot communicates with the Bot Connector service using HTTP over a secured channel (SSL/TLS). When your bot sends a request to the Connector service, it must include information that the Connector service can use to verify its identity. Likewise, when the Connector service sends a request to your bot, it must include information that the bot can use to verify its identity. This article describes the authentication technologies and requirements for the service-level authentication that takes place between a bot and the Bot Connector service. If you are writing your own authentication code, you must implement the security procedures described in this article to enable your bot to exchange messages with the Bot Connector service.

IMPORTANT

If you are writing your own authentication code, it is critical that you implement all security procedures correctly. By implementing all steps in this article, you can mitigate the risk of an attacker being able to read messages that are sent to your bot, send messages that impersonate your bot, and steal secret keys.

If you are using the [Bot Builder SDK for .NET](#) or the [Bot Builder SDK for Node.js](#), you do not need to implement the security procedures described in this article, because the SDK automatically does it for you. Simply configure your project with the App ID and password that you obtained for your bot during [registration](#) and the SDK will handle the rest.

WARNING

In December 2016, v3.1 of the Bot Framework security protocol introduced changes to several values that are used during token generation and validation. In late fall of 2017, v3.2 of the Bot Framework security protocol will be introduced which will again include changes to values that are used during token generation and validation. For more information, see [Security protocol changes](#).

Authentication technologies

Four authentication technologies are used to establish trust between a bot and the Bot Connector:

TECHNOLOGY	DESCRIPTION
SSL/TLS	SSL/TLS is used for all service-to-service connections. X.509v3 certificates are used to establish the identity of all HTTPS services. Clients should always inspect service certificates to ensure they are trusted and valid. (Client certificates are NOT used as part of this scheme.)
OAuth 2.0	OAuth 2.0 login to the Microsoft Account (MSA)/AAD v2 login service is used to generate a secure token that a bot can use to send messages. This token is a service-to-service token; no user login is required.
JSON Web Token (JWT)	JSON Web Tokens are used to encode tokens that are sent to and from the bot. Clients should fully verify all JWT tokens that they receive , according to the requirements outlined in this article.
OpenID metadata	The Bot Connector service publishes a list of valid tokens that it uses to sign its own JWT tokens to OpenID metadata at a well-known, static endpoint.

This article describes how to use these technologies via standard HTTPS and JSON. No special SDKs are required,

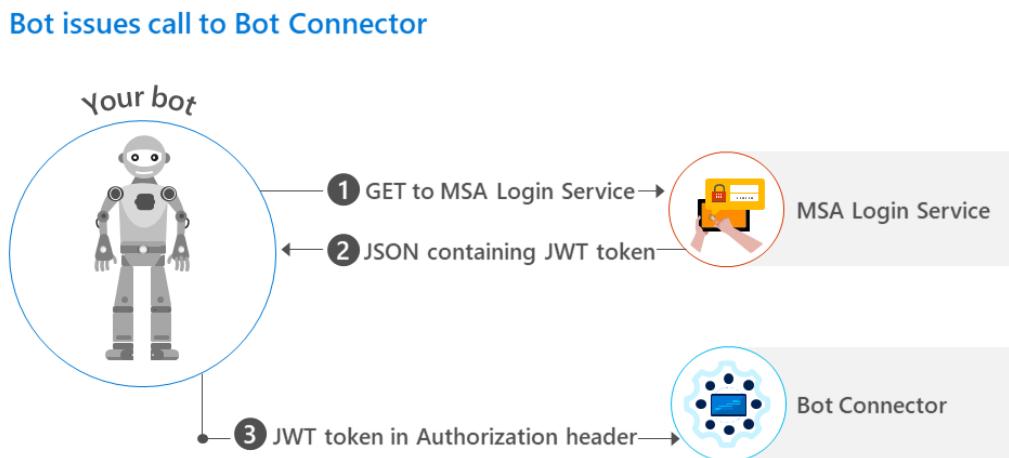
although you may find that helpers for OpenID etc. are useful.

Authenticate requests from your bot to the Bot Connector service

To communicate with the Bot Connector service, you must specify an access token in the `Authorization` header of each API request, using this format:

```
Authorization: Bearer ACCESS_TOKEN
```

This diagram shows the steps for bot-to-connector authentication:



IMPORTANT

If you have not already done so, you must [register](#) your bot with the Bot Framework to obtain its App ID and password. You will need the bot's App ID and password to request an access token.

Step 1: Request an access token from the MSA/AAD v2 login service

To request an access token from the MSA/AAD v2 login service, issue the following request, replacing **MICROSOFT-APP-ID** and **MICROSOFT-APP-PASSWORD** with the App ID and password that you obtained when you [registered](#) your bot with the Bot Framework.

```
POST https://login.microsoftonline.com/botframework.com/oauth2/v2.0/token
Host: login.microsoftonline.com
Content-Type: application/x-www-form-urlencoded

grant_type=client_credentials&client_id=MICROSOFT-APP-ID&client_secret=MICROSOFT-APP-
PASSWORD&scope=https%3A%2F%2Fapi.botframework.com%2F.default
```

Step 2: Obtain the JWT token from the MSA/AAD v2 login service response

If your application is authorized by the MSA/AAD v2 login service, the JSON response body will specify your access token, its type, and its expiration (in seconds).

When adding the token to the `Authorization` header of a request, you must use the exact value that is specified in this response (i.e., do not escape or encode the token value). The access token is valid until its expiration. To prevent token expiration from impacting your bot's performance, you may choose to cache and proactively refresh the token.

This example shows a response from the MSA/AAD v2 login service:

```
HTTP/1.1 200 OK
...
... (other headers)
```

```
{  
  "token_type": "Bearer",  
  "expires_in": 3600,  
  "ext_expires_in": 3600,  
  "access_token": "eyJhbGciOiJIUzI1Ni..."  
}
```

Step 3: Specify the JWT token in the Authorization header of requests

When you send an API request to the Bot Connector service, specify the access token in the `Authorization` header of the request using this format:

```
Authorization: Bearer ACCESS_TOKEN
```

All requests that you send to the Bot Connector service must include the access token in the `Authorization` header. If the token is correctly formed, is not expired, and was generated by the MSA/AAD v2 login service, the Bot Connector service will authorize the request. Additional checks are performed to ensure that the token belongs to the bot that sent the request.

The following example shows how to specify the access token in the `Authorization` header of the request.

```
POST https://smaba.trafficmanager.net/apis/v3/conversations/12345/activities  
Authorization: Bearer eyJhbGciOiJIUzI1Ni...  
  
(JSON-serialized Activity message goes here)
```

IMPORTANT

Only specify the JWT token in the `Authorization` header of requests you send to the Bot Connector service. Do NOT send the token over unsecured channels and do NOT include it in HTTP requests that you send to other services. The JWT token that you obtain from the MSA/AAD v2 login service is like a password and should be handled with great care. Anyone that possesses the token may use it to perform operations on behalf of your bot.

Bot to Connector: example JWT components

```
header:  
{  
  typ: "JWT",  
  alg: "RS256",  
  x5t: "<SIGNING KEY ID>",  
  kid: "<SIGNING KEY ID>"  
},  
payload:  
{  
  aud: "https://api.botframework.com",  
  iss: "https://sts.windows.net/d6d49420-f39b-4df7-a1dc-d59a935871db/",  
  nbf: 1481049243,  
  exp: 1481053143,  
  appid: "<YOUR MICROSOFT APP ID>",  
  ... other fields follow  
}
```

NOTE

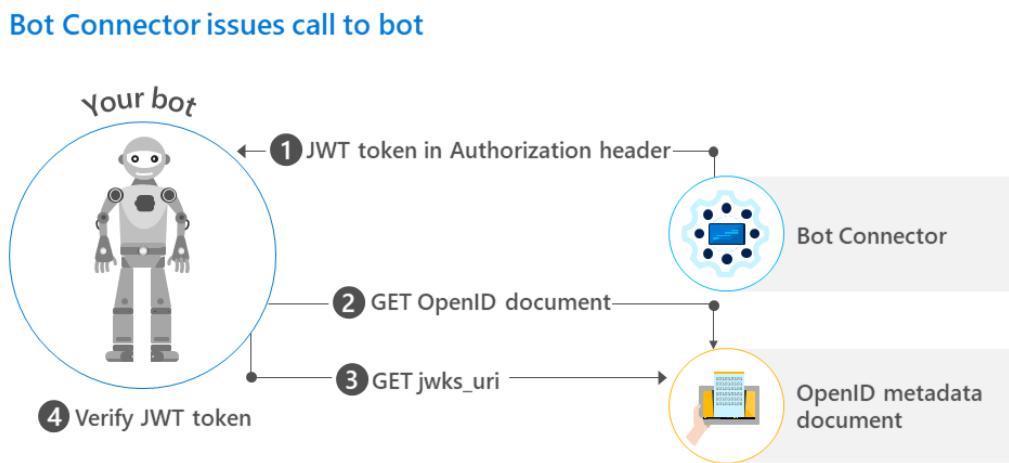
Actual fields may vary in practice. Create and validate all JWT tokens as specified above.

Authenticate requests from the Bot Connector service to your bot

When the Bot Connector service sends a request to your bot, it specifies a signed JWT token in the `Authorization` header

of the request. Your bot can authenticate calls from the Bot Connector service by verifying the authenticity of the signed JWT token.

This diagram shows the steps for connector-to-bot authentication:



Step 2: Get the OpenID metadata document

The OpenID metadata document specifies the location of a second document that lists the Bot Connector service's valid signing keys. To get the OpenID metadata document, issue this request via HTTPS:

```
GET https://login.botframework.com/v1/.well-known/openidconfiguration
```

TIP

This is a static URL that you can hardcode into your application.

The following example shows an OpenID metadata document that is returned in response to the `GET` request. The `jwks_uri` property specifies the location of the document that contains the Bot Connector service's valid signing keys.

```
{
  "issuer": "https://api.botframework.com",
  "authorization_endpoint": "https://invalid.botframework.com",
  "jwks_uri": "https://login.botframework.com/v1/.well-known/keys",
  "id_token_signing_alg_values_supported": [
    "RS256"
  ],
  "token_endpoint_auth_methods_supported": [
    "private_key_jwt"
  ]
}
```

Step 3: Get the list of valid signing keys

To get the list of valid signing keys, issue a `GET` request via HTTPS to the URL specified by the `jwks_uri` property in the OpenID metadata document. For example:

```
GET https://login.botframework.com/v1/.well-known/keys
```

The response body specifies the document in the [JWK format](#) but also includes an additional property for each key: `endorsements`. The list of keys is relatively stable and may be cached for long periods of time (by default, 5 days within the Bot Builder SDK).

The `endorsements` property within each key contains one or more endorsement strings which you can use to verify that

the channel ID specified in the `channelId` property within the [Activity](#) object of the incoming request is authentic. The list of channel IDs that require endorsements is configurable within each bot. By default, it will be the list of all published channel IDs, although bot developers may override selected channel ID values either way. If endorsement for a channel ID is required:

- You should require that any [Activity](#) object sent to your bot with that channel ID is accompanied by a JWT token that is signed with an endorsement for that channel.
- If the endorsement is not present, your bot should reject the request by returning an **HTTP 403 (Forbidden)** status code.

Step 4: Verify the JWT token

To verify the authenticity of the token that was sent by the Bot Connector service, you must extract the token from the `Authorization` header of the request, parse the token, verify its contents, and verify its signature.

JWT parsing libraries are available for many platforms and most implement secure and reliable parsing for JWT tokens, although you must typically configure these libraries to require that certain characteristics of the token (its issuer, audience, etc.) contain correct values. When parsing the token, you must configure the parsing library or write your own validation to ensure the token meets these requirements:

1. The token was sent in the HTTP `Authorization` header with "Bearer" scheme.
2. The token is valid JSON that conforms to the [JWT standard](#).
3. The token contains an "issuer" claim with value of <https://api.botframework.com>.
4. The token contains an "audience" claim with a value equal to the bot's Microsoft App ID.
5. The token has not yet expired. Industry-standard clock-skew is 5 minutes.
6. The token has a valid cryptographic signature, with a key listed in the OpenID keys document that was retrieved in [Step 3](#), using the signing algorithm that is specified in the `id_token_signing_alg_values_supported` property of the Open ID Metadata document that was retrieved in [Step 2](#).
7. The token contains a "serviceUrl" claim with value that matches the `servieUrl` property at the root of the [Activity](#) object of the incoming request.

If the token does not meet all of these requirements, your bot should reject the request by returning an **HTTP 403 (Forbidden)** status code.

IMPORTANT

All of these requirements are important, particularly requirements 4 and 6. Failure to implement ALL of these verification requirements will leave the bot open to attacks which could cause the bot to divulge its JWT token.

Implementers should not expose a way to disable validation of the JWT token that is sent to the bot.

Connector to Bot: example JWT components

```
header:  
{  
  typ: "JWT",  
  alg: "RS256",  
  x5t: "<SIGNING KEY ID>",  
  kid: "<SIGNING KEY ID>"  
},  
payload:  
{  
  aud: "<YOU MICROSOFT APP ID>",  
  iss: "https://api.botframework.com",  

```

NOTE

Actual fields may vary in practice. Create and validate all JWT tokens as specified above.

Authenticate requests from the Bot Framework Emulator to your bot

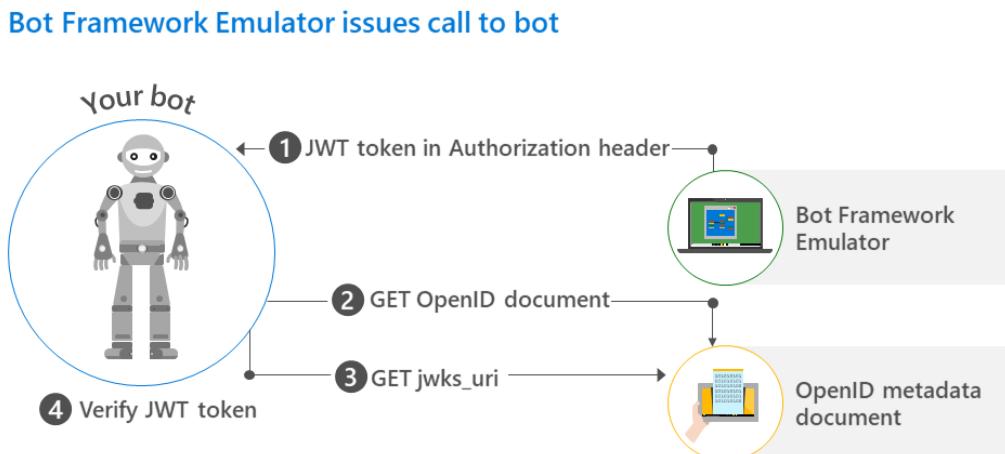
WARNING

In late fall of 2017, v3.2 of the Bot Framework security protocol will be introduced. This new version includes a new "issuer" value within tokens that are exchanged between the Bot Framework Emulator and your bot. To prepare for this change, the below steps outline how to check for both the v3.1 and v3.2 issuer values.

The [Bot Framework Emulator](#) is a desktop tool that you can use to test the functionality of your bot. Although the Bot Framework Emulator uses the same [authentication technologies](#) as described above, it is unable to impersonate the real Bot Connector service. Instead, it uses the Microsoft App ID and Microsoft App Password that you specify when you connect the emulator to your bot to create tokens that are identical to those that the bot creates. When the emulator sends a request to your bot, it specifies the JWT token in the `Authorization` header of the request -- in essence, using the bot's own credentials to authenticate the request.

If you are implementing an authentication library and want to accept requests from the Bot Framework Emulator, you must add this additional verification path. The path is structurally similar to the [Connector -> Bot](#) verification path, but it uses MSA's OpenID document instead of the Bot Connector's OpenID document.

This diagram shows the steps for emulator-to-bot authentication:



Step 2: Get the MSA OpenID metadata document

The OpenID metadata document specifies the location of a second document that lists the valid signing keys. To get the MSA OpenID metadata document, issue this request via HTTPS:

```
GET https://login.microsoftonline.com/botframework.com/v2.0/.well-known/openid-configuration
```

The following example shows an OpenID metadata document that is returned in response to the `GET` request. The `jwks_uri` property specifies the location of the document that contains the valid signing keys.

```
{  
    "authorization_endpoint": "https://login.microsoftonline.com/common/oauth2/v2.0/authorize",  
    "token_endpoint": "https://login.microsoftonline.com/common/oauth2/v2.0/token",  
    "token_endpoint_auth_methods_supported": ["client_secret_post", "private_key_jwt"],  
    "jwks_uri": "https://login.microsoftonline.com/common/discovery/v2.0/keys",  
    ...  
}
```

Step 3: Get the list of valid signing keys

To get the list of valid signing keys, issue a `GET` request via HTTPS to the URL specified by the `jwks_uri` property in the OpenID metadata document. For example:

```
GET https://login.microsoftonline.com/common/discovery/v2.0/keys  
Host: login.microsoftonline.com
```

The response body specifies the document in the [JWK format](#).

Step 4: Verify the JWT token

To verify the authenticity of the token that was sent by the emulator, you must extract the token from the `Authorization` header of the request, parse the token, verify its contents, and verify its signature.

JWT parsing libraries are available for many platforms and most implement secure and reliable parsing for JWT tokens, although you must typically configure these libraries to require that certain characteristics of the token (its issuer, audience, etc.) contain correct values. When parsing the token, you must configure the parsing library or write your own validation to ensure the token meets these requirements:

1. The token was sent in the HTTP `Authorization` header with "Bearer" scheme.
2. The token is valid JSON that conforms to the [JWT standard](#).
3. The token contains an "issuer" claim with value of `https://sts.windows.net/d6d49420-f39b-4df7-a1dc-d59a935871db/` or `https://sts.windows.net/f8cdef31-a31e-4b4a-93e4-5f571e91255a/`. (Checking for both issuer values will ensure you are checking for both the security protocol v3.1 and v3.2 issuer values)
4. The token contains an "audience" claim with a value equal to the bot's Microsoft App ID.
5. The token contains an "appid" claim with the value equal to the bot's Microsoft App ID.
6. The token has not yet expired. Industry-standard clock-skew is 5 minutes.
7. The token has a valid cryptographic signature with a key listed in the OpenID keys document that was retrieved in [Step 3](#).

NOTE

Requirement 5 is a specific to the emulator verification path.

If the token does not meet all of these requirements, your bot should terminate the request by returning an **HTTP 403 (Forbidden)** status code.

IMPORTANT

All of these requirements are important, particularly requirements 4 and 7. Failure to implement ALL of these verification requirements will leave the bot open to attacks which could cause the bot to divulge its JWT token.

Emulator to Bot: example JWT components

```

header:
{
  typ: "JWT",
  alg: "RS256",
  x5t: "<SIGNING KEY ID>",
  kid: "<SIGNING KEY ID>"
},
payload:
{
  aud: "<YOUR MICROSOFT APP ID>",
  iss: "https://sts.windows.net/d6d49420-f39b-4df7-a1dc-d59a935871db/",
  nbf: 1481049243,
  exp: 1481053143,
  ... other fields follow
}

```

NOTE

Actual fields may vary in practice. Create and validate all JWT tokens as specified above.

Security protocol changes

WARNING

Support for v3.0 of the security protocol was discontinued on **July 31, 2017**. If you have written your own authentication code (i.e., did not use the Bot Builder SDK to create your bot), you must upgrade to v3.1 of the security protocol by updating your application to use the v3.1 values that are listed below.

Bot to Connector authentication

OAuth login URL

PROTOCOL VERSION	VALID VALUE
v3.1 & v3.2	https://login.microsoftonline.com/botframework.com/oauth2/v2.0/token

OAuth scope

PROTOCOL VERSION	VALID VALUE
v3.1 & v3.2	https://api.botframework.com/.default

Connector to Bot authentication

OpenID metadata document

PROTOCOL VERSION	VALID VALUE
v3.1 & v3.2	https://login.botframework.com/v1/.well-known/openidconfiguration

JWT Issuer

PROTOCOL VERSION	VALID VALUE
v3.1 & v3.2	https://api.botframework.com

Emulator to Bot authentication

OAuth login URL

PROTOCOL VERSION	VALID VALUE
v3.1 & v3.2	<code>https://login.microsoftonline.com/botframework.com/oauth2/v2.0/token</code>
OAuth scope	
PROTOCOL VERSION	VALID VALUE
v3.1 & v3.2	Your bot's Microsoft App ID + <code>/.default</code>
JWT Audience	
PROTOCOL VERSION	VALID VALUE
v3.1 & v3.2	Your bot's Microsoft App ID
JWT Issuer	
PROTOCOL VERSION	VALID VALUE
v3.1	<code>https://sts.windows.net/d6d49420-f39b-4df7-a1dc-d59a935871db/</code>
v3.2	<code>https://sts.windows.net/f8cdef31-a31e-4b4a-93e4-5f571e91255a/</code>
OpenID metadata document	
PROTOCOL VERSION	VALID VALUE
v3.1 & v3.2	<code>https://login.microsoftonline.com/botframework.com/v2.0/.well-known/openid-configuration</code>

Additional resources

- [Troubleshooting Bot Framework authentication](#)
- [JSON Web Token \(JWT\) draft-jones-json-web-token-07](#)
- [JSON Web Signature \(JWS\) draft-jones-json-web-signature-04](#)
- [JSON Web Key \(JWK\) RFC 7517](#)

Activities overview

7/26/2017 • 2 min to read • [Edit Online](#)

The Bot Connector service exchanges information between bot and channel (user) by passing an [Activity](#) object. The most common type of activity is **message**, but there are other activity types that can be used to communicate various types of information to a bot or channel.

Activity types in the Bot Connector service

The following activity types are supported by the Bot Connector service.

ACTIVITY TYPE	DESCRIPTION
message	Represents a communication between bot and user.
conversationUpdate	Indicates that the bot was added to a conversation, other members were added to or removed from the conversation, or conversation metadata has changed.
contactRelationUpdate	Indicates that the bot was added or removed from a user's contact list.
typing	Indicates that the user or bot on the other end of the conversation is compiling a response.
ping	Represents an attempt to determine whether a bot's endpoint is accessible.
deleteUserData	Indicates to a bot that a user has requested that the bot delete any user data it may have stored.
endOfConversation	Indicates the end of a conversation.

message

Your bot will send **message** activities to communicate information to and receive **message** activities from users. Some messages may be plain text, while others may contain richer content such as [media attachments](#), [buttons](#), [and cards](#) or [channel-specific data](#). For information about commonly-used message properties, see [Create messages](#). For information about how to send and receive messages, see [Send and receive messages](#).

conversationUpdate

A bot receives a **conversationUpdate** activity whenever it has been added to a conversation, other members have been added to or removed from a conversation, or conversation metadata has changed.

If members have been added to the conversation, the activity's `addedMembers` property will identify the new members. If members have been removed from the conversation, the `removedMembers` property will identify the removed members.

TIP

If your bot receives a **conversationUpdate** activity indicating that a user has joined the conversation, you may choose to respond by sending a welcome message to that user.

contactRelationUpdate

A bot receives a **contactRelationUpdate** activity whenever it is added to or removed from a user's contact list. The value of the activity's `action` property (add | remove) indicates whether the bot has been added or removed from the user's contact list.

typing

A bot receives a **typing** activity to indicate that the user is typing a response. A bot may send a **typing** activity to indicate to the user that it is working to fulfill a request or compile a response.

ping

A bot receives a **ping** activity to determine whether its endpoint is accessible. The bot should respond with HTTP status code 200 (OK), 403 (Forbidden), or 401 (Unauthorized).

deleteUserData

A bot receives a **deleteUserData** activity when a user requests deletion of any data that the bot has previously persisted for him or her. If your bot receives this type of activity, it should delete any personally identifiable information (PII) that it has previously stored for the user that made the request.

endOfConversation

A bot receives an **endOfConversation** activity to indicate that the user has ended the conversation. A bot may send an **endOfConversation** activity to indicate to the user that the conversation is ending.

Additional resources

- [Create messages](#)
- [Send and receive messages](#)

Create messages

9/25/2017 • 2 min to read • [Edit Online](#)

Your bot will send [Activity](#) objects of type **message** to communicate information to users, and likewise, will also receive **message** activities from users. Some messages may simply consist of plain text, while others may contain richer content such as [text to be spoken](#), [suggested actions](#), [media attachments](#), [rich cards](#), and [channel-specific data](#). This article describes some of the commonly-used message properties.

Message text and formatting

Message text can be formatted using **plain**, **markdown**, or **xml**. The default format for the `textFormat` property is **markdown** and interprets text using Markdown formatting standards. The level of text format support varies across channels. To see if a feature you want to use is supported on the channel you target, preview the feature using [Channel Inspector](#).

The `textFormat` property of the [Activity](#) object can be used to specify the format of the text. For example, to create a basic message that contains only plain text, set the `textFormat` property of the [Activity](#) object to **plain**, set the `text` property to the contents of the message and set the `locale` property to the locale of the sender.

For a list of commonly supported text formatting, see [Text formatting](#).

Attachments

The `attachments` property of the [Activity](#) object can be used to send simple media attachments (image, audio, video, file) and rich cards. For details, see [Add media attachments to messages](#) and [Add rich cards to messages](#).

Entities

The `entities` property of the [Activity](#) object is an array of open-ended [schema.org](#) objects that allows the exchange of common contextual metadata between the channel and bot.

Mention entities

Many channels support the ability for a bot or user to "mention" someone within the context of a conversation. To mention a user in a message, populate the message's `entities` property with a [Mention](#) object.

Place entities

To convey [location-related information](#) within a message, populate the message's `entities` property with [Place](#) object.

Channel data

The `channelData` property of the [Activity](#) object can be used to implement channel-specific functionality. For details, see [Implement channel-specific functionality](#).

Text to speech

The `speak` property of the [Activity](#) object can be used to specify the text to be spoken by your bot on a speech-enabled channel and the `inputHint` property of the [Activity](#) object can be used to influence the state of the client's microphone. For details, see [Add speech to messages](#) and [Add input hints to messages](#).

Suggested actions

The `suggestedActions` property of the [Activity](#) object can be used to present buttons that the user can tap to provide input. Unlike buttons that appear within rich cards (which remain visible and accessible to the user even after being tapped), buttons that appear within the suggested actions pane will disappear after the user makes a selection. For details, see [Add suggested actions to messages](#).

Additional resources

- [Preview features with the Channel Inspector](#)
- [Activities overview](#)
- [Send and receive messages](#)
- [Add media attachments to messages](#)
- [Add rich cards to messages](#)
- [Add speech to messages](#)
- [Add input hints to messages](#)
- [Add suggested actions to messages](#)
- [Implement channel-specific functionality](#)

Send and receive messages

7/26/2017 • 4 min to read • [Edit Online](#)

The Bot Connector service enables a bot to communicate across multiple channels such as Skype, Email, Slack, and more. It facilitates communication between bot and user, by relaying **activities** from bot to channel and from channel to bot. Every activity contains information used for routing the message to the appropriate destination along with information about who created the message, the context of the message, and the recipient of the message. This article describes how to use the Bot Connector service to exchange **message** activities between bot and user on a channel.

Reply to a message

Create a reply

When the user sends a message to your bot, your bot will receive the message as an **Activity** object of type **message**. To create a reply to a user's message, create a new **Activity** object and start by setting these properties:

PROPERTY	VALUE
conversation	Set this property to the contents of the <code>conversation</code> property in the user's message.
from	Set this property to the contents of the <code>recipient</code> property in the user's message.
locale	Set this property to the contents of the <code>locale</code> property in the user's message, if specified.
recipient	Set this property to the contents of the <code>from</code> property in the user's message.
replyTold	Set this property to the contents of the <code>id</code> property in the user's message.
type	Set this property to message .

Next, set the properties that specify the information that you want to communicate to the user. For example, you can set the `text` property to specify the text to be displayed in the message, set the `speak` property to specify text to be spoken by your bot, and set the `attachments` property to specify media attachments or rich cards to include in the message. For detailed information about commonly-used message properties, see [Create messages](#).

Send the reply

Use the `serviceUrl` property in the incoming activity to identify the base URI that your bot should use to issue its response.

To send the reply, issue this request:

```
POST /v3/conversations/{conversationId}/activities/{activityId}
```

In this request URI, replace **{conversationId}** with the value of the `conversation` object's `id` property within

your (reply) Activity and replace **{activityId}** with the value of the `replyToId` property within your (reply) Activity. Set the body of the request to the [Activity](#) object that you created to represent your reply.

The following example shows a request that sends a simple text-only reply to a user's message. In this example request, `https://smiba.trafficmanager.net/apis` represents the base URI; the base URI for requests that your bot issues may be different. For details about setting the base URI, see [API Reference](#).

```
POST https://smiba.trafficmanager.net/apis/v3/conversations/abcd1234/activities/5d5cdc723
Authorization: Bearer ACCESS_TOKEN
Content-Type: application/json
```

```
{
  "type": "message",
  "from": {
    "id": "12345678",
    "name": "Pepper's News Feed"
  },
  "conversation": {
    "id": "abcd1234",
    "name": "Convo1"
  },
  "recipient": {
    "id": "1234abcd",
    "name": "SteveW"
  },
  "text": "My bot's reply",
  "replyToId": "5d5cdc723"
}
```

Send a (non-reply) message

A majority of the messages that your bot sends will be in reply to messages that it receives from the user. However, there may be times when your bot needs to send a message to the conversation that is not a direct reply to any message from the user. For example, your bot may need to start a new topic of conversation or send a goodbye message at the end of the conversation.

To send a message to a conversation that is not a direct reply to any message from the user, issue this request:

```
POST /v3/conversations/{conversationId}/activities
```

In this request URI, replace **{conversationId}** with the ID of the conversation.

Set the body of the request to an [Activity](#) object that you create to represent your reply.

NOTE

The Bot Framework does not impose any restrictions on the number of messages that a bot may send. However, most channels enforce throttling limits to restrict bots from sending a large number of messages in a short period of time. Additionally, if the bot sends multiple messages in quick succession, the channel may not always render the messages in the proper sequence.

Start a conversation

There may be times when your bot needs to initiate a conversation with one or more users. To start a conversation with a user on a channel, your bot must know its account information and the user's account information on that channel.

TIP

If your bot may need to start conversations with its users in the future, cache user account information, other relevant information such as user preferences and locale, and the service URL (to use as the base URI in the Start Conversation request).

To start a conversation, issue this request:

```
POST /v3/conversations
```

Set the body of the request to a [Conversation](#) object that specifies your bot's account information and the account information of the user(s) that you want to include in the conversation.

NOTE

Not all channels support group conversations. Consult the channel's documentation to determine whether a channel supports group conversations and to identify the maximum number of participants that a channel allows in a conversation.

The following example shows a request that starts a conversation. In this example request,

`https://smba.trafficmanager.net/apis` represents the base URI; the base URI for requests that your bot issues may be different. For details about setting the base URI, see [API Reference](#).

```
POST https://smba.trafficmanager.net/apis/v3/conversations
Authorization: Bearer ACCESS_TOKEN
Content-Type: application/json
```

```
{
  "bot": {
    "id": "12345678",
    "name": "bot's name"
  },
  "isGroup": false,
  "members": [
    {
      "id": "1234abcd",
      "name": "recipient's name"
    }
  ],
  "topicName": "News Alert"
}
```

If the conversation is established with the specified users, the response will contain an ID that identifies the conversation.

```
{
  "id": "abcd1234"
}
```

Your bot can then use this conversation ID to [send a message](#) to the user(s) within the conversation.

Additional resources

- [Activities overview](#)

- Create messages

Add media attachments to messages

9/26/2017 • 3 min to read • [Edit Online](#)

Bots and channels typically exchange text strings but some channels also support exchanging attachments, which lets your bot send richer messages to users. For example, your bot can send media attachments (e.g., images, videos, audio, files) and [rich cards](#). This article describes how to add media attachments to messages using the Bot Connector service.

TIP

To determine the type and number of attachments that a channel supports, and how the channel renders attachments, see the [Channel Inspector](#).

Add a media attachment

To add a media attachment to a message, create an [Attachment](#) object, set the `name` property, set the `contentUrl` property to the URL of the media file, and set the `contentType` property to the appropriate media type (e.g., **image/png, audio/wav, video/mp4**). Then within the [Activity](#) object that represents your message, specify your [Attachment](#) object within the `attachments` array.

The following example shows a request that sends a message containing text and a single image attachment. In this example request, `https://smba.trafficmanager.net/apis` represents the base URI; the base URI for requests that your bot issues may be different. For details about setting the base URI, see [API Reference](#).

```
POST https://smba.trafficmanager.net/apis/v3/conversations/abcd1234/activities/5d5cdc723
Authorization: Bearer ACCESS_TOKEN
Content-Type: application/json
```

```
{
  "type": "message",
  "from": {
    "id": "12345678",
    "name": "sender's name"
  },
  "conversation": {
    "id": "abcd1234",
    "name": "conversation's name"
  },
  "recipient": {
    "id": "1234abcd",
    "name": "recipient's name"
  },
  "text": "Here's a picture of the duck I was telling you about.",
  "attachments": [
    {
      "contentType": "image/png",
      "contentUrl": "http://aka.ms/Fo983c",
      "name": "duck-on-a-rock.jpg"
    }
  ],
  "replyToId": "5d5cdc723"
}
```

For channels that support inline binaries of an image, you can set the `content` property of the `Attachment` to a base64 binary of the image (for example, `data:image/png;base64,iVBORw0KGgo...`). The channel will display the image or the image's URL next to the message's text string.

You can attach a video file or audio file to a message by using the same process as described above for an image file. Depending on the channel, the video and audio may be played inline or it may be displayed as a link.

NOTE

Your bot may also receive messages that contain media attachments. For example, a message that your bot receives may contain an attachment if the channel enables the user to upload a photo to be analyzed or a document to be stored.

Add an AudioCard attachment

Adding an [AudioCard](#) or [VideoCard](#) attachment is the same as adding a media attachment. For example, the following JSON shows how to add an audio card in the media attachment.

```
{
  "type": "message",
  "from": {
    "id": "12345678",
    "name": "sender's name"
  },
  "conversation": {
    "id": "abcd1234",
    "name": "conversation's name"
  },
  "recipient": {
    "id": "1234abcd",
    "name": "recipient's name"
  },
  "attachments": [
    {
      "contentType": "application/vnd.microsoft.card.audio",
      "content": {
        "title": "Allegro in C Major",
        "subtitle": "Allegro Duet",
        "text": "No Image, No Buttons, Autoloop, Autostart, Sharable",
        "media": [
          {
            "url": "https://contoso.com/media/AllegrofromDuetinCMajor.mp3"
          }
        ],
        "shareable": true,
        "autoloop": true,
        "autostart": true,
        "value": {
          // Supplementary parameter for this card
        }
      }
    ],
    "replyToId": "5d5cdc723"
  }
}
```

Once the channel receives this attachment, it will start playing the audio file. If a user interacts with audio by clicking the **Pause** button, for example, the channel will send a callback to the bot with a JSON that look something like this:

```
{
  ...
  "type": "event",
  "name": "media/pause",
  "value": {
    "url": // URL for media
    "cardValue": {
      // Supplementary parameter for this card
    }
  }
}
```

The media event name **media/pause** will appear in the `activity.name` field. Reference the below table for a list of all media event names.

EVENT	DESCRIPTION
media/next	The client skipped to the next media
media/pause	The client paused playing media
media/play	The client began playing media
media/previous	The client skipped to the previous media
media/resume	The client resumed playing media
media/stop	The client stopped playing media

Additional resources

- [Create messages](#)
- [Send and receive messages](#)
- [Add rich cards to messages](#)
- [Channel Inspector](#)

Add rich card attachments to messages

9/14/2017 • 6 min to read • [Edit Online](#)

Bots and channels typically exchange text strings but some channels also support exchanging attachments, which lets your bot send richer messages to users. For example, your bot can send rich cards and media attachments (e.g., images, videos, audio, files). This article describes how to add rich card attachments to messages using the Bot Connector service.

NOTE

For information about how to add media attachments to messages, see [Add media attachments to messages](#).

Types of rich cards

A rich card comprises a title, description, link, and images. A message can contain multiple rich cards, displayed in either list format or carousel format. The Bot Framework currently supports eight types of rich cards:

CARD TYPE	DESCRIPTION
AdaptiveCard	A customizable card that can contain any combination of text, speech, images, buttons, and input fields. See per-channel support .
AnimationCard	A card that can play animated GIFs or short videos.
AudioCard	A card that can play an audio file.
HeroCard	A card that typically contains a single large image, one or more buttons, and text.
ThumbnailCard	A card that typically contains a single thumbnail image, one or more buttons, and text.
ReceiptCard	A card that enables a bot to provide a receipt to the user. It typically contains the list of items to include on the receipt, tax and total information, and other text.
SignInCard	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.
VideoCard	A card that can play videos.

TIP

To determine the type of rich cards that a channel supports and see how the channel renders rich cards, see the [Channel Inspector](#). Consult the channel's documentation for information about limitations on the contents of cards (e.g., the maximum number of buttons or maximum length of title).

Process events within rich cards

To process events within rich cards, use [CardAction](#) objects to specify what should happen when the user clicks a button or taps a section of the card. Each [CardAction](#) object contains these properties:

PROPERTY	TYPE	DESCRIPTION
type	string	type of action (one of the values specified in the table below)
title	string	title of the button
image	string	image URL for the button
value	string	value needed to perform the specified type of action

NOTE

Buttons within Adaptive Cards are not created using [CardAction](#) objects, but instead using the schema that is defined by [Adaptive Cards](#). See [Add an Adaptive Card to a message](#) for an example that shows how to add buttons to an Adaptive Card.

This table lists the valid values for the `type` property of a [CardAction](#) object and describes the expected contents of the `value` property for each type:

TYPE	VALUE
openUrl	URL to be opened in the built-in browser
imBack	Text of the message to send to the bot (from the user who clicked the button or tapped the card). This message (from user to bot) will be visible to all conversation participants via the client application that is hosting the conversation.
postBack	Text of the message to send to the bot (from the user who clicked the button or tapped the card). Some client applications may display this text in the message feed, where it will be visible to all conversation participants.
call	Destination for a phone call in this format: tel:123123123123
playAudio	URL of audio to be played
playVideo	URL of video to be played
showImage	URL of image to be displayed
downloadFile	URL of file to be downloaded
signin	URL of OAuth flow to be initiated

Add a Hero card to a message

To add a rich card attachment to a message, first create an object that corresponds to the [type of card](#) that you want to add to the message. Then create an [Attachment](#) object, set its `contentType` property to the card's media type and its `content` property to the object you created to represent the card. Specify your [Attachment](#) object within the `attachments` array of the message.

TIP

Messages that contain rich card attachments typically do not specify `text`.

Some channels allow you to add multiple rich cards to the `attachments` array within a message. This capability can be useful in scenarios where you want to provide the user with multiple options. For example, if your bot lets users book hotel rooms, it could present the user with a list of rich cards that shows the types of available rooms. Each card could contain a picture and list of amenities corresponding to the room type and the user could select a room type by tapping a card or clicking a button.

TIP

To display multiple rich cards in list format, set the [Activity](#) object's `attachmentLayout` property to "list". To display multiple rich cards in carousel format, set the [Activity](#) object's `attachmentLayout` property to "carousel". If the channel does not support carousel format, it will display the rich cards in list format, even if the `attachmentLayout` property specifies "carousel".

The following example shows a request that sends a message containing a single Hero card attachment. In this example request, `https://smileybot.smba.trafficmanager.net/apis` represents the base URI; the base URI for requests that your bot issues may be different. For details about setting the base URI, see [API Reference](#).

```
POST https://smileybot.smba.trafficmanager.net/apis/v3/conversations/abcd1234/activities/5d5cdc723
Authorization: Bearer ACCESS_TOKEN
Content-Type: application/json
```

```
{
  "type": "message",
  "from": {
    "id": "12345678",
    "name": "sender's name"
  },
  "conversation": {
    "id": "abcd1234",
    "name": "conversation's name"
  },
  "recipient": {
    "id": "1234abcd",
    "name": "recipient's name"
  },
  "attachments": [
    {
      "contentType": "application/vnd.microsoft.card.hero",
      "content": {
        "title": "title goes here",
        "subtitle": "subtitle goes here",
        "text": "descriptive text goes here",
        "images": [
          {
            "url": "http://aka.ms/Fo983c",
            "alt": "picture of a duck",
            "tap": {
              "type": "playAudio",
              "value": "url to an audio track of a duck call goes here"
            }
          }
        ],
        "buttons": [
          {
            "type": "playAudio",
            "title": "Duck Call",
            "value": "url to an audio track of a duck call goes here"
          },
          {
            "type": "openUrl",
            "title": "Watch Video",
            "image": "http://aka.ms/Fo983c",
            "value": "url goes here of the duck in flight"
          }
        ]
      }
    ],
    "replyToId": "5d5cdc723"
  }
}
```

Add an Adaptive card to a message

The Adaptive Card can contain any combination of text, speech, images, buttons, and input fields. Adaptive Cards are created using the JSON format specified in [Adaptive Cards](#), which gives you full control over card content and format.

Leverage the information within the [Adaptive Cards](#) site to understand Adaptive Card schema, explore Adaptive Card elements, and see JSON samples that can be used to create cards of varying composition and complexity. Additionally, you can use the Interactive Visualizer to design Adaptive Card payloads and preview card output.

The following example shows a request that sends a message containing a single Adaptive Card for a calendar reminder. In this example request, `https://smbs.trafficmanager.net/apis` represents the base URL; the base URI for requests that your bot issues may be different. For details about setting the base URI, see [API Reference](#).

```
POST https://smba.trafficmanager.net/apis/v3/conversations/abcd1234/activities/5d5cdc723
```

```
Authorization: Bearer ACCESS_TOKEN
```

```
Content-Type: application/json
```

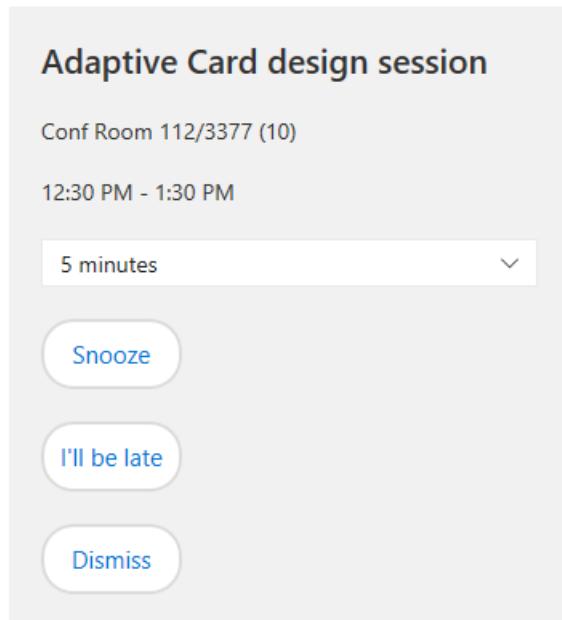
```
{
    "type": "message",
    "from": {
        "id": "12345678",
        "name": "sender's name"
    },
    "conversation": {
        "id": "abcd1234",
        "name": "conversation's name"
    },
    "recipient": {
        "id": "1234abcd",
        "name": "recipient's name"
    },
    "attachments": [
        {
            "contentType": "application/vnd.microsoft.card.adaptive",
            "content": {
                "type": "AdaptiveCard",
                "body": [
                    {
                        "type": "TextBlock",
                        "text": "Adaptive Card design session",
                        "size": "large",
                        "weight": "bolder"
                    },
                    {
                        "type": "TextBlock",
                        "text": "Conf Room 112/3377 (10)"
                    },
                    {
                        "type": "TextBlock",
                        "text": "12:30 PM - 1:30 PM"
                    },
                    {
                        "type": "TextBlock",
                        "text": "Snooze for"
                    },
                    {
                        "type": "Input.ChoiceSet",
                        "id": "snooze",
                        "style": "compact",
                        "choices": [
                            {
                                "title": "5 minutes",
                                "value": "5",
                                "isSelected": true
                            },
                            {
                                "title": "15 minutes",
                                "value": "15"
                            },
                            {
                                "title": "30 minutes",
                                "value": "30"
                            }
                        ]
                    }
                ],
                "actions": [
                    {
                        "type": "Action.Http",
                        "url": "https://www.example.com/snooze"
                    }
                ]
            }
        }
    ]
}
```

```

        "method": "POST",
        "url": "http://foo.com",
        "title": "Snooze"
    },
{
    "type": "Action.Http",
    "method": "POST",
    "url": "http://foo.com",
    "title": "I'll be late"
},
{
    "type": "Action.Http",
    "method": "POST",
    "url": "http://foo.com",
    "title": "Dismiss"
}
]
}
],
"replyToId": "5d5cdc723"
}

```

The resulting card contains three blocks of text, an input field (choice list), and three buttons:



Additional resources

- [Create messages](#)
- [Send and receive messages](#)
- [Add media attachments to messages](#)
- [Channel Inspector](#)
- [Adaptive Cards](#)

Add speech to messages

7/26/2017 • 1 min to read • [Edit Online](#)

If you are building a bot for a speech-enabled channel such as Cortana, you can construct messages that specify the text to be spoken by your bot. You can also attempt to influence the state of the client's microphone by specifying an [input hint](#) to indicate whether your bot is accepting, expecting, or ignoring user input.

Specify text to be spoken by your bot

To specify text to be spoken by your bot on a speech-enabled channel, set the `speak` property within the [Activity](#) object that represents your message. You can set the `speak` property to either a plain text string or a string that is formatted as [Speech Synthesis Markup Language \(SSML\)](#), an XML-based markup language that enables you to control various characteristics of your bot's speech such as voice, rate, volume, pronunciation, pitch, and more.

The following request sends a message that specifies text to be displayed and text to be spoken and indicates that the bot is [expecting user input](#). It specifies the `speak` property using [SSML](#) format to indicate that the word "sure" should be spoken with a moderate amount of emphasis. In this example request,

`https://smba.trafficmanager.net/apis` represents the base URI; the base URI for requests that your bot issues may be different. For details about setting the base URI, see [API Reference](#).

```
POST https://smba.trafficmanager.net/apis/v3/conversations/abcd1234/activities/5d5cdc723
Authorization: Bearer ACCESS_TOKEN
Content-Type: application/json
```

```
{
  "type": "message",
  "from": {
    "id": "12345678",
    "name": "sender's name"
  },
  "conversation": {
    "id": "abcd1234",
    "name": "conversation's name"
  },
  "recipient": {
    "id": "1234abcd",
    "name": "recipient's name"
  },
  "text": "Are you sure that you want to cancel this transaction?",
  "speak": "Are you <emphasis level='moderate'>sure</emphasis> that you want to cancel this transaction?",
  "inputHint": "expectingInput",
  "replyToId": "5d5cdc723"
}
```

Input hints

When you send a message on a speech-enabled channel, you can attempt to influence the state of the client's microphone by also including an input hint to indicate whether your bot is accepting, expecting, or ignoring user input. For more information, see [Add input hints to messages](#).

Additional resources

- Create messages
- Send and receive messages
- Add input hints to messages
- Speech Synthesis Markup Language (SSML)

Add input hints to messages

7/26/2017 • 2 min to read • [Edit Online](#)

By specifying an input hint for a message, you can indicate whether your bot is accepting, expecting, or ignoring user input after the message is delivered to the client. For many channels, this enables clients to set the state of user input controls accordingly. For example, if a message's input hint indicates that the bot is ignoring user input, the client may close the microphone and disable the input box to prevent the user from providing input.

Accepting input

To indicate that your bot is passively ready for input but is not awaiting a response from the user, set the `inputHint` property to **acceptingInput** within the [Activity](#) object that represents your message. On many channels, this will cause the client's input box to be enabled and microphone to be closed, but still accessible to the user. For example, Cortana will open the microphone to accept input from the user if the user holds down the microphone button.

The following example shows a request that sends a message and specifies that the bot is accepting input. In this example request, `https://smba.trafficmanager.net/apis` represents the base URI; the base URI for requests that your bot issues may be different. For details about setting the base URI, see [API Reference](#).

```
POST https://smba.trafficmanager.net/apis/v3/conversations/abcd1234/activities/5d5cdc723
Authorization: Bearer ACCESS_TOKEN
Content-Type: application/json
```

```
{
  "type": "message",
  "from": {
    "id": "12345678",
    "name": "sender's name"
  },
  "conversation": {
    "id": "abcd1234",
    "name": "conversation's name"
  },
  "recipient": {
    "id": "1234abcd",
    "name": "recipient's name"
  },
  "text": "Here's a picture of the house I was telling you about.",
  "inputHint": "acceptingInput",
  "replyToId": "5d5cdc723"
}
```

Expecting input

To indicate that your bot is awaiting a response from the user, set the `inputHint` property to **expectingInput** within the [Activity](#) object that represents your message. On many channels, this will cause the client's input box to be enabled and microphone to be open.

The following example shows a request that sends a message and specifies that the bot is expecting input. In this example request, `https://smba.trafficmanager.net/apis` represents the base URI; the base URI for requests that your bot issues may be different. For details about setting the base URI, see [API Reference](#).

```
POST https://smba.trafficmanager.net/apis/v3/conversations/abcd1234/activities/5d5cdc723
Authorization: Bearer ACCESS_TOKEN
Content-Type: application/json
```

```
{
  "type": "message",
  "from": {
    "id": "12345678",
    "name": "sender's name"
  },
  "conversation": {
    "id": "abcd1234",
    "name": "conversation's name"
  },
  "recipient": {
    "id": "1234abcd",
    "name": "recipient's name"
  },
  "text": "What is your favorite color?",
  "inputHint": "expectingInput",
  "replyToId": "5d5cdc723"
}
```

Ignoring input

To indicate that your bot is not ready to receive input from the user, set the `inputHint` property to **ignoringInput** within the [Activity](#) object that represents your message. On many channels, this will cause the client's input box to be disabled and microphone to be closed.

The following example shows a request that sends a message and specifies that the bot is ignoring input. In this example request, `https://smba.trafficmanager.net/apis` represents the base URI; the base URI for requests that your bot issues may be different. For details about setting the base URI, see [API Reference](#).

```
POST https://smba.trafficmanager.net/apis/v3/conversations/abcd1234/activities/5d5cdc723
Authorization: Bearer ACCESS_TOKEN
Content-Type: application/json
```

```
{
  "type": "message",
  "from": {
    "id": "12345678",
    "name": "sender's name"
  },
  "conversation": {
    "id": "abcd1234",
    "name": "conversation's name"
  },
  "recipient": {
    "id": "1234abcd",
    "name": "recipient's name"
  },
  "text": "Please hold while I perform the calculation.",
  "inputHint": "ignoringInput",
  "replyToId": "5d5cdc723"
}
```

Additional resources

- [Create messages](#)
- [Send and receive messages](#)

Add suggested actions to messages

9/13/2017 • 1 min to read • [Edit Online](#)

Suggested actions enable your bot to present buttons that the user can tap to provide input. Suggested actions appear close to the composer and enhance user experience by enabling the user to answer a question or make a selection with a simple tap of a button, rather than having to type a response with a keyboard. Unlike buttons that appear within rich cards (which remain visible and accessible to the user even after being tapped), buttons that appear within the suggested actions pane will disappear after the user makes a selection. This prevents the user from tapping stale buttons within a conversation and simplifies bot development (since you will not need to account for that scenario).

TIP

To learn how various channels render suggested actions, see the [Channel Inspector](#).

Send suggested actions

To add suggested actions to a message, set the `suggestedActions` property of the [Activity](#) to specify the list of [CardAction](#) objects that represent the buttons to be presented to the user.

The following request sends a message that presents three suggested actions to the user. In this example request, `https://smileybot.smba.trafficmanager.net/apis` represents the base URI; the base URI for requests that your bot issues may be different. For details about setting the base URI, see [API Reference](#).

```
POST https://smileybot.smba.trafficmanager.net/apis/v3/conversations/abcd1234/activities/5d5cdc723
Authorization: Bearer ACCESS_TOKEN
Content-Type: application/json
```

```
{
  "type": "message",
  "from": {
    "id": "12345678",
    "name": "sender's name"
  },
  "conversation": {
    "id": "abcd1234",
    "name": "conversation's name"
  },
  "recipient": {
    "id": "1234abcd",
    "name": "recipient's name"
  },
  "text": "I have colors in mind, but need your help to choose the best one.",
  "inputHint": "expectingInput",
  "suggestedActions": {
    "actions": [
      {
        "type": "imBack",
        "title": "Blue",
        "value": "Blue"
      },
      {
        "type": "imBack",
        "title": "Red",
        "value": "Red"
      },
      {
        "type": "imBack",
        "title": "Green",
        "value": "Green"
      }
    ]
  },
  "replyToId": "5d5cdc723"
}
```

When the user taps one of the suggested actions, the bot will receive a message from the user that contains the **value** of the corresponding action.

Additional resources

- [Create messages](#)
- [Send and receive messages](#)

Implement channel-specific functionality

9/13/2017 • 5 min to read • [Edit Online](#)

Some channels provide features that cannot be implemented by using only [message text and attachments](#). To implement channel-specific functionality, you can pass native metadata to a channel in the [Activity](#) object's `channelData` property. For example, your bot can use the `channelData` property to instruct Telegram to send a sticker or to instruct Office365 to send an email.

This article describes how to use a message activity's `channelData` property to implement this channel-specific functionality:

CHANNEL	FUNCTIONALITY
Email	Send and receive an email that contains body, subject, and importance metadata
Slack	Send full fidelity Slack messages
Facebook	Send Facebook notifications natively
Telegram	Perform Telegram-specific actions, such as sharing a voice memo or a sticker
Kik	Send and receive native Kik messages

NOTE

The value of an [Activity](#) object's `channelData` property is a JSON object. The structure of the JSON object will vary according to the channel and the functionality being implemented, as described below.

Create a custom Email message

To create an email message, set the [Activity](#) object's `channelData` property to a JSON object that contains these properties:

PROPERTY	DESCRIPTION
<code>htmlBody</code>	An HTML document that specifies the body of the email message. See the channel's documentation for information about supported HTML elements and attributes.
<code>importance</code>	The email's importance level. Valid values are high , normal , and low . The default value is normal .
<code>subject</code>	The email's subject. See the channel's documentation for information about field requirements.

This snippet shows an example of the `channelData` property for a custom email message.

```
"channelData": {  
    "htmlBody" : "<html><body style=\"font-family: Calibri; font-size: 11pt;\">This is the email body!</body>  
    </html>",  
    "subject": "This is the email subject",  
    "importance": "high"  
}
```

Create a full-fidelity Slack message

To create a full-fidelity Slack message, set the [Activity](#) object's `channelData` property to a JSON object that specifies [Slack messages](#), [Slack attachments](#), and/or [Slack buttons](#).

NOTE

To support buttons in Slack messages, you must enable **Interactive Messages** when you [connect your bot](#) to the Slack channel.

This snippet shows an example of the `channelData` property for a custom Slack message.

```

"channelData": {
    "text": "Now back in stock! :tada:",
    "attachments": [
        {
            "title": "The Further Adventures of Slackbot",
            "author_name": "Stanford S. Strickland",
            "author_icon": "https://api.slack.com/img/api/homepage_custom_integrations-2x.png",
            "image_url": "http://i.imgur.com/OJkaVOI.jpg?1"
        },
        {
            "fields": [
                {
                    "title": "Volume",
                    "value": "1",
                    "short": true
                },
                {
                    "title": "Issue",
                    "value": "3",
                    "short": true
                }
            ]
        },
        {
            "title": "Synopsis",
            "text": "After @episod pushed exciting changes to a devious new branch back in Issue 1, Slackbot notifies @don about an unexpected deploy..."
        },
        {
            "fallback": "Would you recommend it to customers?",
            "title": "Would you recommend it to customers?",
            "callback_id": "comic_1234_xyz",
            "color": "#3AA3E3",
            "attachment_type": "default",
            "actions": [
                {
                    "name": "recommend",
                    "text": "Recommend",
                    "type": "button",
                    "value": "recommend"
                },
                {
                    "name": "no",
                    "text": "No",
                    "type": "button",
                    "value": "bad"
                }
            ]
        }
    ]
}

```

When a user clicks a button within a Slack message, your bot will receive a response message in which the `channelData` property is populated with a `payload` JSON object. The `payload` object specifies contents of the original message, identifies the button that was clicked, and identifies the user who clicked the button.

This snippet shows an example of the `channelData` property in the message that a bot receives when a user clicks a button in the Slack message.

```

"channelData": {
  "payload": {
    "actions": [
      {
        "name": "recommend",
        "value": "yes"
      }
    ],
    . . .
    "original_message": "...",
    "response_url": "https://hooks.slack.com/actions/..."
  }
}

```

Your bot can reply to this message in the [normal manner](#), or it can post its response directly to the endpoint that is specified by the `payload` object's `response_url` property. For information about when and how to post a response to the `response_url`, see [Slack Buttons](#).

Create a Facebook notification

To create a Facebook notification, set the [Activity](#) object's `channelData` property to a JSON object that specifies these properties:

PROPERTY	DESCRIPTION
<code>notification_type</code>	The type of notification (e.g., REGULAR , SILENT_PUSH , NO_PUSH).
<code>attachment</code>	An attachment that specifies an image, video, or other multimedia type, or a templated attachment such as a receipt.

NOTE

For details about format and contents of the `notification_type` property and `attachment` property, see the [Facebook API documentation](https://developers.facebook.com/docs/messenger-platform/send-api-reference#guidelines).

This snippet shows an example of the `channelData` property for a Facebook receipt attachment.

```

"channelData": {
  "notification_type": "NO_PUSH",
  "attachment": {
    "type": "template",
    "payload": {
      "template_type": "receipt",
      . . .
    }
  }
}

```

Create a Telegram message

To create a message that implements Telegram-specific actions, such as sharing a voice memo or a sticker, set the [Activity](#) object's `channelData` property to a JSON object that specifies these properties:

PROPERTY	DESCRIPTION
method	The Telegram Bot API method to call.
parameters	The parameters of the specified method.

These Telegram methods are supported:

- answerInlineQuery
- editMessageCaption
- editMessageReplyMarkup
- editMessageText
- forwardMessage
- kickChatMember
- sendAudio
- sendChatAction
- sendContact
- sendDocument
- sendLocation
- sendMessage
- sendPhoto
- sendSticker
- sendVenue
- sendVideo
- sendVoice
- unbanChateMember

For details about these Telegram methods and their parameters, see the [Telegram Bot API documentation](#).

NOTE

- The `chat_id` parameter is common to all Telegram methods. If you do not specify `chat_id` as a parameter, the framework will provide the ID for you.
- Instead of passing file contents inline, specify the file using a URL and media type as shown in the example below.
- Within each message that your bot receives from the Telegram channel, the `channelData` property will include the message that your bot sent previously.

This snippet shows an example of a `channelData` property that specifies a single Telegram method.

```
"channelData": {
    "method": "sendSticker",
    "parameters": {
        "sticker": {
            "url": "https://domain.com/path/gif",
            "mediaType": "image/gif",
        }
    }
}
```

This snippet shows an example of a `channelData` property that specifies an array of Telegram methods.

```

"channelData": [
    {
        "method": "sendSticker",
        "parameters": {
            "sticker": {
                "url": "https://domain.com/path/gif",
                "mediaType": "image/gif",
            }
        }
    },
    {
        "method": "sendMessage",
        "parameters": {
            "text": "<b>This message is HTML formatted.</b>",
            "parse_mode": "HTML"
        }
    }
]

```

Create a native Kik message

To create a native Kik message, set the [Activity](#) object's `channelData` property to a JSON object that specifies this property:

PROPERTY	DESCRIPTION
messages	An array of Kik messages. For details about Kik message format, see Kik Message Formats .

This snippet shows an example of the `channelData` property for a native Kik message.

```

"channelData": {
    "messages": [
        {
            "chatId": "c6dd8165...",
            "type": "link",
            "to": "kikhandle",
            "title": "My Webpage",
            "text": "Some text to display",
            "url": "http://botframework.com",
            "picUrl": "http://lorempixel.com/400/200/",
            "attribution": {
                "name": "My App",
                "iconUrl": "http://lorempixel.com/50/50/"
            },
            "noForward": true,
            "kikJsData": {
                "key": "value"
            }
        }
    ]
}

```

Additional resources

- [Activities overview](#)
- [Create messages](#)
- [Send and receive messages](#)
- [Preview features with the Channel Inspector](#)

Manage state data

7/26/2017 • 8 min to read • [Edit Online](#)

The Bot State service enables your bot to store and retrieve state data that is associated with a user, a conversation, or a specific user within the context of a specific conversation. You may store up to 32 kilobytes of data for each user on a channel, each conversation on a channel, and each user within the context of a conversation on a channel. State data can be used for many purposes, such as determining where a prior conversation left off or simply greeting a returning user by name. If you store a user's preferences, you can use that information to customize the conversation the next time you chat. For example, you might alert the user to a news article about a topic that interests her, or alert a user when an appointment becomes available.

IMPORTANT

The Bot State service is intended for prototyping only and is not designed for use by bots in a production environment. For performance and security reasons, you should implement your own mechanism for managing state data in the production environment.

Data concurrency

To control concurrency of data that is stored using the Bot State service, the framework uses the entity tag (ETag) for `POST` requests. The framework does not use the standard headers for ETags. Instead, the body of the request and response specifies the ETag using the `eTag` property.

For example, if you issue a `GET` request to retrieve user data from the store, the response will contain the `eTag` property. If you change the data and issue a `POST` request to save the updated data to the store, your request may include the `eTag` property that specifies the same value as you received earlier in the `GET` response. If the ETag specified in your `POST` request matches the current value in the store, the server will save the user's data and respond with **HTTP 200 Success** and specify a new `eTag` value in the body of the response. If the ETag specified in your `POST` request does not match the current value in the store, the server will respond with **HTTP 412 Precondition Failed** to indicate that the user's data in the store has changed since you last saved or retrieved it.

TIP

A `GET` response will always include an `eTag` property, but you do not need to specify the `eTag` property in `GET` requests. An `eTag` property value of asterisk (`*`) indicates that you have not previously saved data for the specified combination of channel, user, and conversation.

Specifying the `eTag` property in `POST` requests is optional. If concurrency is not an issue for your bot, do not include the `eTag` property in `POST` requests.

Save user data

To save state data for a user on a specific channel, issue this request:

```
POST /v3/botstate/{channelId}/users/{userId}
```

In this request URL, replace `{channelId}` with the channel's ID and replace `{userId}` with the user's ID on that channel. The `channelId` and `from` properties within any message that your bot has previously received from the

user will contain these IDs. You may also choose to cache these values in a secure location so that you can access the user's data in the future without having to extract them from a message.

Set the body of the request to a [BotData](#) object, where the `data` property specifies the data that you want to save. If you use entity tags for [concurrency control](#), set the `eTag` property to the ETag that you received in the response the last time you saved or retrieved the user's data (whichever is the most recent). If you do not use entity tags for concurrency, then do not include the `eTag` property in the request.

NOTE

This `POST` request will overwrite user data in the store only if the specified ETag matches the server's ETag, or if no ETag is specified in the request.

Request

The following example shows a request that saves data for a user on a specific channel. In this example request, `https://smba.trafficmanager.net/apis` represents the base URI; the base URI for requests that your bot issues may be different. For details about setting the base URI, see [API Reference](#).

```
POST https://smba.trafficmanager.net/apis/v3/botstate/abcd1234/users/12345678
Authorization: Bearer ACCESS_TOKEN
Content-Type: application/json
```

```
{
  "data": [
    {
      "trail": "Lake Serene",
      "miles": 8.2,
      "difficulty": "Difficult",
    },
    {
      "trail": "Rainbow Falls",
      "miles": 6.3,
      "difficulty": "Moderate",
    }
  ],
  "eTag": "a1b2c3d4"
}
```

Response

The response will contain a [BotData](#) object with a new `eTag` value.

Get user data

To get state data that has previously been saved for the user on a specific channel, issue this request:

```
GET /v3/botstate/{channelId}/users/{userId}
```

In this request URI, replace `{channelId}` with the channel's ID and replace `{userId}` with the user's ID on that channel. The `channelId` and `from` properties within any message that your bot has previously received from the user will contain these IDs. You may also choose to cache these values in a secure location so that you can access the user's data in the future without having to extract them from a message.

Request

The following example shows a request that gets data that has previously been saved for the user. In this example

request, `https://smba.trafficmanager.net/apis` represents the base URI; the base URI for requests that your bot issues may be different. For details about setting the base URI, see [API Reference](#).

```
GET https://smba.trafficmanager.net/apis/v3/botstate/abcd1234/users/12345678
Authorization: Bearer ACCESS_TOKEN
Content-Type: application/json
```

Response

The response will contain a `BotData` object, where the `data` property contains the data that you have previously saved for the user and the `eTag` property contains the ETag that you may use within a subsequent request to save the user's data. If you have not previously saved data for the user, the `data` property will be `null` and the `eTag` property will contain an asterisk (`*`).

This example shows the response to the `GET` request:

```
{
  "data": [
    {
      "trail": "Lake Serene",
      "miles": 8.2,
      "difficulty": "Difficult",
    },
    {
      "trail": "Rainbow Falls",
      "miles": 6.3,
      "difficulty": "Moderate",
    }
  ],
  "eTag": "xyz12345"
}
```

Save conversation data

To save state data for a conversation on a specific channel, issue this request:

```
POST /v3/botstate/{channelId}/conversations/{conversationId}
```

In this request URI, replace `{channelId}` with the channel's ID, and replace `{conversationId}` with ID of the conversation. The `channelId` and `conversation` properties within any message that your bot has previously sent to or received in the context of the conversation will contain these IDs. You may also choose to cache these values in a secure location so that you can access the conversation's data in the future without having to extract them from a message.

Set the request body to a `BotData` object, as described in [Save user data](#).

IMPORTANT

Because the `Delete user data` operation does not delete data that has been stored using the `Save conversation data` operation, you must NOT use this operation to store a user's personally identifiable information (PII).

Response

The response will contain a `BotData` object with a new `eTag` value.

Get conversation data

To get state data that has previously been saved for a conversation on a specific channel, issue this request:

```
GET /v3/botstate/{channelId}/conversations/{conversationId}
```

In this request URI, replace **{channelId}** with the channel's ID and replace **{conversationId}** with ID of the conversation. The `channelId` and `conversation` properties within any message that your bot has previously sent or received in the context of the conversation will contain these IDs. You may also choose to cache these values in a secure location so that you can access the conversation's data in the future without having to extract them from a message.

Response

The response will contain a [BotData](#) object with a new `eTag` value.

Save private conversation data

To save state data for a user within the context of a specific conversation, issue this request:

```
POST /v3/botstate/{channelId}/conversations/{conversationId}/users/{userId}
```

In this request URI, replace **{channelId}** with the channel's ID, replace **{conversationId}** with ID of the conversation, and replace **{userId}** with the user's ID on that channel. The `channelId`, `conversation`, and `from` properties within any message that your bot has previously received from the user in the context of the conversation will contain these IDs. You may also choose to cache these values in a secure location so that you can access the conversation's data in the future without having to extract them from a message.

Set the request body to a [BotData](#) object, as described in [Save user data](#).

Response

The response will contain a [BotData](#) object with a new `eTag` value.

Get private conversation data

To get state data that has previously been saved for a user within the context of a specific conversation, issue this request:

```
GET /v3/botstate/{channelId}/conversations/{conversationId}/users/{userId}
```

In this request URI, replace **{channelId}** with the channel's ID, replace **{conversationId}** with ID of the conversation, and replace **{userId}** with the user's ID on that channel. The `channelId`, `conversation`, and `from` properties within any message that your bot has previously received from the user in the context of the conversation will contain these IDs. You may also choose to cache these values in a secure location so that you can access the conversation's data in the future without having to extract them from a message.

Response

The response will contain a [BotData](#) object with a new `eTag` value.

Delete user data

To delete state data for a user on a specific channel, issue this request:

```
DELETE /v3/botstate/{channelId}/users/{userId}
```

In this request URI, replace **{channelId}** with the channel's ID and replace **{userId}** with the user's ID on that channel. The `channelId` and `from` properties within any message that your bot has previously received from the user will contain these IDs. You may also choose to cache these values in a secure location so that you can access the user's data in the future without having to extract them from a message.

IMPORTANT

This operation deletes data that has previously been stored for the user by using either the [Save user data](#) operation or the [Save private conversation data](#) operation. It does NOT delete data that has previously been stored by using the [Save conversation data](#) operation. Therefore, you must NOT use the **Save conversation data** operation to store a user's personally identifiable information (PII).

Your bot should execute the **Delete user data** operation when it receives an [Activity](#) of type `deleteUserData` or an activity of type `contactRelationUpdate` that indicates the bot has been removed from the user's contact list.

Additional resources

- [Key concepts](#)
- [Activities overview](#)

API reference

10/12/2017 • 29 min to read • [Edit Online](#)

Within the Bot Framework, the Bot Connector service enables your bot to exchange messages with users on channels that are configured in the Bot Framework Portal and the Bot State service enables your bot to store and retrieve state data that is related to the conversations that your bot conducts using the Bot Connector service. Both services use industry-standard REST and JSON over HTTPS.

Base URI

When a user sends a message to your bot, the incoming request contains an `Activity` object with a `serviceUrl` property that specifies the endpoint to which your bot should send its response. To access the Bot Connector service or the Bot State service, use the `serviceUrl` value as the base URI for API requests.

For example, assume that your bot receives the following activity when the user sends a message to the bot.

```
{  
  "type": "message",  
  "id": "bf3cc9a2f5de...",  
  "timestamp": "2016-10-19T20:17:52.2891902Z",  
  "serviceUrl": "https://smba.trafficmanager.net/apis",  
  "channelId": "channel's name/id",  
  "from": {  
    "id": "1234abcd",  
    "name": "user's name"  
  },  
  "conversation": {  
    "id": "abcd1234",  
    "name": "conversation's name"  
  },  
  "recipient": {  
    "id": "12345678",  
    "name": "bot's name"  
  },  
  "text": "Haircut on Saturday"  
}
```

The `serviceUrl` property within the user's message indicates that the bot should send its response to the endpoint `https://smba.trafficmanager.net/apis`; this will be the base URI for any subsequent requests that the bot issues in the context of this conversation. If your bot will need to send a proactive message to the user, be sure to save the value of `serviceUrl`.

The following example shows the request that the bot issues to respond to the user's message.

```
POST https://smba.trafficmanager.net/apis/v3/conversations/abcd1234/activities/bf3cc9a2f5de...  
Authorization: Bearer eyJhbGciOiJIUzI1Ni...  
Content-Type: application/json
```

```
{
  "type": "message",
  "from": {
    "id": "12345678",
    "name": "bot's name"
  },
  "conversation": {
    "id": "abcd1234",
    "name": "conversation's name"
  },
  "recipient": {
    "id": "1234abcd",
    "name": "user's name"
  },
  "text": "I have several times available on Saturday!",
  "replyToId": "bf3cc9a2f5de..."
}
```

Headers

Request headers

In addition to the standard HTTP request headers, every API request that you issue must include an `Authorization` header that specifies an access token to authenticate your bot. Specify the `Authorization` header using this format:

```
Authorization: Bearer ACCESS_TOKEN
```

For details about how to obtain an access token for your bot, see [Authenticate requests from your bot to the Bot Connector service](#).

Response headers

In addition to the standard HTTP response headers, every response will contain an `X-Correlating-OperationId` header. The value of this header is an ID that corresponds to the Bot Framework log entry which contains details about the request. Any time that you receive an error response, you should capture the value of this header. If you are not able to independently resolve the issue, include this value in the information that you provide to the Support team when you report the issue.

HTTP status codes

The [HTTP status code](#) that is returned with each response indicates the outcome of the corresponding request.

HTTP STATUS CODE	MEANING
200	The request succeeded.
201	The request succeeded.
202	The request has been accepted for processing.
204	The request succeeded but no content was returned.
400	The request was malformed or otherwise incorrect.
401	The bot is not authorized to make the request.

HTTP STATUS CODE	MEANING
403	The bot is not allowed to perform the requested operation.
404	The requested resource was not found.
500	An internal server error occurred.
503	The service is unavailable.

Errors

Any response that specifies an HTTP status code in the 4xx range or 5xx range will include an [ErrorResponse](#) object in the body of the response that provides information about the error. If you receive an error response in the 4xx range, inspect the [ErrorResponse](#) object to identify the cause of the error and resolve your issue prior to resubmitting the request.

Conversation operations

Use these operations to create conversations, send messages (activities), and manage the contents of conversations.

OPERATION	DESCRIPTION
Create Conversation	Creates a new conversation.
Send to Conversation	Sends an activity (message) to the end of the specified conversation.
Reply to Activity	Sends an activity (message) to the specified conversation, as a reply to the specified activity.
Get Conversation Members	Gets the members of the specified conversation.
Get Activity Members	Gets the members of the specified activity within the specified conversation.
Update Activity	Updates an existing activity.
Delete Activity	Deletes an existing activity.
Upload Attachment to Channel	Uploads an attachment directly into a channel's blob storage.

Create Conversation

Creates a new conversation.

```
POST /v3/conversations
```

Request body	A Conversation object
Returns	A ResourceResponse object

Send to Conversation

Sends an activity (message) to the specified conversation. The activity will be appended to the end of the conversation according to the timestamp or semantics of the channel. To reply to a specific message within the conversation, use [Reply to Activity](#) instead.

```
POST /v3/conversations/{conversationId}/activities
```

Request body	An Activity object
Returns	An Identification object

Reply to Activity

Sends an activity (message) to the specified conversation, as a reply to the specified activity. The activity will be added as a reply to another activity, if the channel supports it. If the channel does not support nested replies, then this operation behaves like [Send to Conversation](#).

```
POST /v3/conversations/{conversationId}/activities/{activityId}
```

Request body	An Activity object
Returns	An Identification object

Get Conversation Members

Gets the members of the specified conversation.

```
GET /v3/conversations/{conversationId}/members
```

Request body	n/a
Returns	An array of ChannelAccount objects

Get Activity Members

Gets the members of the specified activity within the specified conversation.

```
GET /v3/conversations/{conversationId}/activities/{activityId}/members
```

Request body	n/a
Returns	An array of ChannelAccount objects

Update Activity

Some channels allow you to edit an existing activity to reflect the new state of a bot conversation. For example, you might remove buttons from a message in the conversation after the user has clicked one of the buttons. If successful, this operation updates the specified activity within the specified conversation.

```
PUT /v3/conversations/{conversationId}/activities/{activityId}
```

Request body	An Activity object
Returns	An Identification object

Delete Activity

Some channels allow you to delete an existing activity. If successful, this operation removes the specified activity from the specified conversation.

```
DELETE /v3/conversations/{conversationId}/activities/{activityId}
```

Request body	n/a
Returns	An HTTP Status code that indicates the outcome of the operation. Nothing is specified in the body of the response.

Upload Attachment to Channel

Uploads an attachment for the specified conversation directly into a channel's blob storage. This enables you to store data in a compliant store.

```
POST /v3/conversations/{conversationId}/attachments
```

Request body	An AttachmentUpload object.
Returns	A ResourceResponse object. The id property specifies the attachment ID that can be used with the Get Attachments Info operation and the Get Attachment operation.

Attachment operations

Use these operations to retrieve information about an attachment as well the binary data for the file itself.

OPERATION	DESCRIPTION
Get Attachment Info	Gets information about the specified attachment, including file name, file type, and the available views (e.g., original or thumbnail).
Get Attachment	Gets the specified view of the specified attachment as binary content.

Get Attachment Info

Gets information about the specified attachment, including file name, type, and the available views (e.g., original or thumbnail).

```
GET /v3/attachments/{attachmentId}
```

Request body	n/a
Returns	An AttachmentInfo object

Get Attachment

Gets the specified view of the specified attachment as binary content.

```
GET /v3/attachments/{attachmentId}/views/{viewId}
```

Request body	n/a
Returns	Binary content that represents the specified view of the specified attachment

State operations

Use these operations to store and retrieve state data.

OPERATION	DESCRIPTION
Set User Data	Stores state data for a specific user on a channel.
Set Conversation Data	Stores state data for a specific conversation on a channel.
Set Private Conversation Data	Stores state data for a specific user within the context of a specific conversation on a channel.
Get User Data	Retrieves state data that has previously been stored for a specific user across all conversations on a channel.
Get Conversation Data	Retrieves state data that has previously been stored for a specific conversation on a channel.
Get Private Conversation Data	Retrieves state data that has previously been stored for a specific user within the context of a specific conversation on a channel.
Delete State For User	Deletes state data that has previously been stored for a user by using the Set User Data operation or the Set Private Conversation Data operation.

Set User Data

Stores state data for the specified user on the specified channel.

```
POST /v3/botstate/{channelId}/users/{userId}
```

Request body	A BotData object
Returns	A BotData object

Set Conversation Data

Stores state data for the specified conversation on the specified channel.

```
POST /v3/botstate/{channelId}/conversations/{conversationId}
```

Request body	A BotData object
Returns	A BotData object

Set Private Conversation Data

Stores state data for the specified user within the context of the specified conversation on the specified channel.

```
POST /v3/botstate/{channelId}/conversations/{conversationId}/users/{userId}
```

Request body	A BotData object
Returns	A BotData object

Get User Data

Retrieves state data that has previously been stored for the specified user across all conversations on the specified channel.

```
GET /v3/botstate/{channelId}/users/{userId}
```

Request body	n/a
Returns	A BotData object

Get Conversation Data

Retrieves state data that has previously been stored for the specified conversation on the specified channel.

```
GET /v3/botstate/{channelId}/conversations/{conversationId}
```

Request body	n/a
Returns	A BotData object

Get Private Conversation Data

Retrieves state data that has previously been stored for the specified user within the context of the specified conversation on the specified channel.

```
GET /v3/botstate/{channelId}/conversations/{conversationId}/users/{userId}
```

Request body	n/a
Returns	A BotData object

Delete State For User

Deletes state data that has previously been stored for the specified user on the specified channel by using either the [Set User Data](#) operation or the [Set Private Conversation Data](#) operation.

```
DELETE /v3/botstate/{channelId}/users/{userId}
```

Request body	n/a
Returns	An array of strings (IDs)

Schema

Schema defines the object and its properties that your bot can use to communicate with a user.

OBJECT	DESCRIPTION
Activity object	Defines a message that is exchanged between bot and user.
AnimationCard object	Defines a card that can play animated GIFs or short videos.
Attachment object	Defines additional information to include in the message. An attachment may be a media file (e.g., audio, video, image, file) or a rich card.
AttachmentData object	Describes an attachment data.
AttachmentInfo object	Describes an attachment.
AttachmentView object	Defines a attachment view.
AttachmentUpload object	Defines an attachment to be uploaded.
AudioCard object	Defines a card that can play an audio file.
BotData object	Defines state data for a user, a conversation, or a user in the context of a specific conversation that is stored using the Bot State service.
CardAction object	Defines an action to perform.

OBJECT	DESCRIPTION
CardImage object	Defines an image to display on a card.
ChannelAccount object	Defines a bot or user account on the channel.
Conversation object	Defines a conversation, including the bot and users that are included within the conversation.
ConversationAccount object	Defines a conversation in a channel.
ConversationParameters object	Define parameters for creating a new conversation
ConversationReference object	Defines a particular point in a conversation.
ConversationResourceResponse object	"A response containing a resource
Entity object	Defines an entity object.
Error object	Defines an error.
ErrorResponse object	Defines an HTTP API response.
Fact object	Defines a key-value pair that contains a fact.
Geocoordinates object	Defines a geographical location using World Geodetic System (WSG84) coordinates.
HeroCard object	Defines a card with a large image, title, text, and action buttons.
Identification object	Identifies a resource.
MediaEventValue object	Supplementary parameter for media events.
MediaUrl object	Defines the URL to a media file's source.
Mention object	Defines a user or bot that was mentioned in the conversation.
MessageReaction object	Defines a reaction to a message.
Place object	Defines a place that was mentioned in the conversation.
ReceiptCard object	Defines a card that contains a receipt for a purchase.
ReceiptItem object	Defines a line item within a receipt.
ResourceResponse object	Defines a resource.
SignInCard object	Defines a card that lets a user sign in to a service.
SuggestedActions object	Defines the options from which a user can choose.

OBJECT	DESCRIPTION
ThumbnailCard object	Defines a card with a thumbnail image, title, text, and action buttons.
ThumbnailUrl object	Defines the URL to an image's source.
VideoCard object	Defines a card that can play videos.

Activity object

Defines a message that is exchanged between bot and user.

PROPERTY	TYPE	DESCRIPTION
action	string	The action to apply or that was applied. Use the type property to determine context for the action. For example, if type is contactRelationUpdate , the value of the action property would be add if the user added your bot to their contacts list, or remove if they removed your bot from their contacts list.
attachments	Attachment[]	Array of Attachment objects that defines additional information to include in the message. Each attachment may be either a media file (e.g., audio, video, image, file) or a rich card.
attachmentLayout	string	Layout of the rich card attachments that the message includes. One of these values: carousel , list . For more information about rich card attachments, see Add rich card attachments to messages .
channelData	object	An object that contains channel-specific content. Some channels provide features that require additional information that cannot be represented using the attachment schema. For those cases, set this property to the channel-specific content as defined in the channel's documentation. For more information, see Implement channel-specific functionality .
channelId	string	An ID that uniquely identifies the channel. Set by the channel.
conversation	ConversationAccount	A ConversationAccount object that defines the conversation to which the activity belongs.
code	string	Code indicating why the conversation has ended.

PROPERTY	TYPE	DESCRIPTION
entities	object[]	Array of objects that represents the entities that were mentioned in the message. Objects in this array may be any Schema.org object. For example, the array may include Mention objects that identify someone who was mentioned in the conversation and Place objects that identify a place that was mentioned in the conversation.
from	ChannelAccount	A ChannelAccount object that specifies the sender of the message.
historyDisclosed	boolean	Flag that indicates whether or not history is disclosed. Default value is false .
id	string	ID that uniquely identifies the activity on the channel.
inputHint	string	Value that indicates whether your bot is accepting, expecting, or ignoring user input after the message is delivered to the client. One of these values: acceptingInput , expectingInput , ignoringInput .
locale	string	Locale of the language that should be used to display text within the message, in the format <code><language>-<country></code> . The channel uses this property to indicate the user's language, so that your bot may specify display strings in that language. Default value is en-US .
localTimestamp	string	Date and time that the message was sent in the local time zone, expressed in ISO-8601 format.
membersAdded	ChannelAccount[]	Array of ChannelAccount objects that represents the list of users that joined the conversation. Present only if activity type is "conversationUpdate" and users joined the conversation.
membersRemoved	ChannelAccount[]	Array of ChannelAccount objects that represents the list of users that left the conversation. Present only if activity type is "conversationUpdate" and users left the conversation.
name	string	Name of the operation to invoke or the name of the event.
recipient	ChannelAccount	A ChannelAccount object that specifies the recipient of the message.

PROPERTY	TYPE	DESCRIPTION
relatesTo	ConversationReference	A ConversationReference object that defines a particular point in a conversation.
replyToId	string	The ID of the message to which this message replies. To reply to a message that the user sent, set this property to the ID of the user's message. Not all channels support threaded replies. In these cases, the channel will ignore this property and use time ordered semantics (timestamp) to append the message to the conversation.
serviceUrl	string	URL that specifies the channel's service endpoint. Set by the channel.
speak	string	Text to be spoken by your bot on a speech-enabled channel. To control various characteristics of your bot's speech such as voice, rate, volume, pronunciation, and pitch, specify this property in Speech Synthesis Markup Language (SSML) format.
suggestedActions	SuggestedActions	A SuggestedActions object that defines the options from which the user can choose.
summary	string	Summary of the information that the message contains. For example, for a message that is sent on an email channel, this property may specify the first 50 characters of the email message.
text	string	Text of the message that is sent from user to bot or bot to user. See the channel's documentation for limits imposed upon the contents of this property.
textFormat	string	Format of the message's text . One of these values: markdown , plain , xml . For details about text format, see Create messages .
timestamp	string	Date and time that the message was sent in the UTC time zone, expressed in ISO-8601 format.
topicName	string	Topic of the conversation to which the activity belongs.

PROPERTY	TYPE	DESCRIPTION
type	string	Type of activity. One of these values: contactRelationUpdate , conversationUpdate , deleteUserData , message , ping , typing , endOfConversation . For details about activity types, see Activities overview .
value	object	Open-ended value.

[Back to Schema table](#)

AnimationCard object

Defines a card that can play animated GIFs or short videos.

PROPERTY	TYPE	DESCRIPTION
autoloop	boolean	Flag that indicates whether to replay the list of animated GIFs when the last one ends. Set this property to true to automatically replay the animation; otherwise, false . The default value is true .
autoplay	boolean	Flag that indicates whether to automatically play the animation when the card is displayed. Set this property to true to automatically play the animation; otherwise, false . The default value is true .
buttons	CardAction[]	Array of CardAction objects that enable the user to perform one or more actions. The channel determines the number of buttons that you may specify.
image	ThumbnailUrl	A ThumbnailUrl object that specifies the image to display on the card.
media	MediaUrl[]	Array of MediaUrl objects that specifies the list of animated GIFs to play.
shareable	boolean	Flag that indicates whether the animation may be shared with others. Set this property to true if the animation may be shared; otherwise, false . The default value is true .
subtitle	string	Subtitle to display under the card's title.
text	string	Description or prompt to display under the card's title or subtitle.

PROPERTY	TYPE	DESCRIPTION
title	string	Title of the card.
value	object	Supplementary parameter for this card

[Back to Schema table](#)

Attachment object

Defines additional information to include in the message. An attachment may be a media file (e.g., audio, video, image, file) or a rich card.

PROPERTY	TYPE	DESCRIPTION
contentType	string	<p>The media type of the content in the attachment. For media files, set this property to known media types such as image/png, audio/wav, and video/mp4. For rich cards, set this property to one of these vendor-specific types:</p> <ul style="list-style-type: none"> • application/vnd.microsoft.card.adaptive: A rich card that can contain any combination of text, speech, images, buttons, and input fields. Set the content property to an AdaptiveCard object. • application/vnd.microsoft.card.animation: A rich card that plays animation. Set the content property to an AnimationCard object. • application/vnd.microsoft.card.audio: A rich card that plays audio files. Set the content property to an AudioCard object. • application/vnd.microsoft.card.video: A rich card that plays videos. Set the content property to a VideoCard object. • application/vnd.microsoft.card.hero: A Hero card. Set the content property to a HeroCard object. • application/vnd.microsoft.card.thumbnail: A Thumbnail card. Set the content property to a ThumbnailCard object. • application/vnd.microsoft.connector.card.receipt: A Receipt card. Set the content property to a ReceiptCard object. • application/vnd.microsoft.connector.card.signin: A user Sign In card. Set the content property to a SignInCard object.

PROPERTY	TYPE	DESCRIPTION
contentUrl	string	URL for the content of the attachment. For example, if the attachment is an image, set contentUrl to the URL that represents the location of the image. Supported protocols are: HTTP, HTTPS, File, and Data.
content	object	The content of the attachment. If the attachment is a rich card, set this property to the rich card object. This property and the contentUrl property are mutually exclusive.
name	string	Name of the attachment.
thumbnailUrl	string	URL to a thumbnail image that the channel can use if it supports using an alternative, smaller form of content or contentUrl . For example, if you set contentType to application/word and set contentUrl to the location of the Word document, you might include a thumbnail image that represents the document. The channel could display the thumbnail image instead of the document. When the user clicks the image, the channel would open the document.

[Back to Schema table](#)

AttachmentData object

Describes an attachment data.

PROPERTY	TYPE	DESCRIPTION
name	string	Name of the attachment.
originalBase64	string	Attachment content.
thumbnailBase64	string	Attachment thumbnail content.
type	string	Content-type of the attachment.

AttachmentInfo object

Describes an attachment.

PROPERTY	TYPE	DESCRIPTION
name	string	Name of the attachment.
type	string	ContentType of the attachment.

PROPERTY	TYPE	DESCRIPTION
views	AttachmentView[]	Array of AttachmentView objects that represent the available views for the attachment.

[Back to Schema table](#)

AttachmentView object

Defines a attachment view.

PROPERTY	TYPE	DESCRIPTION
viewId	string	View ID.
size	number	Size of the file.

[Back to Schema table](#)

AttachmentUpload object

Defines an attachment to be uploaded.

PROPERTY	TYPE	DESCRIPTION
type	string	ContentType of the attachment.
name	string	Name of the attachment.
originalBase64	string	Binary data that represents the contents of the original version of the file.
thumbnailBase64	string	Binary data that represents the contents of the thumbnail version of the file.

[Back to Schema table](#)

AudioCard object

Defines a card that can play an audio file.

PROPERTY	TYPE	DESCRIPTION
aspect	string	Aspect ratio of the thumbnail that is specified in the image property. Valid values are 16:9 and 9:16 .
autoloop	boolean	Flag that indicates whether to replay the list of audio files when the last one ends. Set this property to true to automatically replay the audio files; otherwise, false . The default value is true .

PROPERTY	TYPE	DESCRIPTION
autoplay	boolean	Flag that indicates whether to automatically play the audio when the card is displayed. Set this property to true to automatically play the audio; otherwise, false . The default value is true .
buttons	CardAction[]	Array of CardAction objects that enable the user to perform one or more actions. The channel determines the number of buttons that you may specify.
image	ThumbnailUrl	A ThumbnailUrl object that specifies the image to display on the card.
media	MediaUrl[]	Array of MediaUrl objects that specifies the list of audio files to play.
shareable	boolean	Flag that indicates whether the audio files may be shared with others. Set this property to true if the audio may be shared; otherwise, false . The default value is true .
subtitle	string	Subtitle to display under the card's title.
text	string	Description or prompt to display under the card's title or subtitle.
title	string	Title of the card.
value	object	Supplementary parameter for this card

[Back to Schema table](#)

BotData object

Defines state data for a user, a conversation, or a user in the context of a specific conversation that is stored using the Bot State service.

PROPERTY	TYPE	DESCRIPTION
data	object	In a request, a JSON object that specifies the properties and values to store using the Bot State service. In a response, a JSON object that specifies the properties and values that have been stored using the Bot State service.
eTag	string	The entity tag value that you can use to control data concurrency for the data that you store using the Bot State service. For more information, see Manage state data .

[Back to Schema table](#)

CardAction object

Defines an action to perform.

PROPERTY	TYPE	DESCRIPTION
image	string	URL of an image to display on the
text	string	Text for the action
title	string	Text of the button. Only applicable for a button's action.
button. Only applicable for a button's action.		
type	string	Type of action to perform. For a list of valid values, see Add rich card attachments to messages .
value	object	Supplementary parameter for the action. The value of this property will vary according to the action type . For more information, see Add rich card attachments to messages .

[Back to Schema table](#)

CardImage object

Defines an image to display on a card.

PROPERTY	TYPE	DESCRIPTION
alt	string	Description of the image. You should include the description to support accessibility.
tap	CardAction	A CardAction object that specifies the action to perform if the user taps or clicks the image.
url	string	URL to the source of the image or the base64 binary of the image (for example, <code>data:image/png;base64,iVBORw0KGgo...</code>).

[Back to Schema table](#)

ChannelAccount object

Defines a bot or user account on the channel.

PROPERTY	TYPE	DESCRIPTION
id	string	ID that uniquely identifies the bot or user on the channel.
name	string	Name of the bot or user.

[Back to Schema table](#)

Conversation object

Defines a conversation, including the bot and users that are included within the conversation.

PROPERTY	TYPE	DESCRIPTION
bot	ChannelAccount	A ChannelAccount object that identifies the bot.
isGroup	boolean	Flag to indicate whether or not this is a group conversation. Set to true if this is a group conversation; otherwise, false . The default is false . To start a group conversation, the channel must support group conversations.
members	ChannelAccount[]	Array of ChannelAccount objects that identify the members of the conversation. This list must contain a single user unless isGroup is set to true . This list may include other bots.
topicName	string	Title of the conversation.
activity	Activity	In a Create Conversation request, an Activity object that defines the first message to post to the new conversation.

[Back to Schema table](#)

ConversationAccount object

Defines a conversation in a channel.

PROPERTY	TYPE	DESCRIPTION
id	string	The ID that identifies the conversation. The ID is unique per channel. If the channel starts the conversion, it sets this ID; otherwise, the bot sets this property to the ID that it gets back in the response when it starts the conversation (see Starting a conversation).

PROPERTY	TYPE	DESCRIPTION
isGroup	boolean	Flag to indicate whether or not this is a group conversation. Set to true if this is a group conversation; otherwise, false . The default is false .
name	string	A display name that can be used to identify the conversation.

[Back to Schema table](#)

ConversationParameters object

Define parameters for creating a new conversation

PROPERTY	TYPE	DESCRIPTION
isGroup	boolean	Indicates if this is a group conversation.
bot	ChannelAccount	Address of the bot in the conversation.
members	array	List of members to add to the conversation.
topicName	string	Topic title of a conversation. This property is only used if a channel supports it.
activity	Activity	(optional) Use this activity as the initial message to the conversation when creating a new conversation.
channelData	object	Channel specific payload for creating the conversation.

ConversationReference object

Defines a particular point in a conversation.

PROPERTY	TYPE	DESCRIPTION
activityId	string	ID that uniquely identifies the activity that this object references.
bot	ChannelAccount	A ChannelAccount object that identifies the bot in the conversation that this object references.
channelId	string	An ID that uniquely identifies the channel in the conversation that this object references.
conversation	ConversationAccount	A ConversationAccount object that defines the conversation that this object references.

PROPERTY	TYPE	DESCRIPTION
serviceUrl	string	URL that specifies the channel's service endpoint in the conversation that this object references.
user	ChannelAccount	A ChannelAccount object that identifies the user in the conversation that this object references.

[Back to Schema table](#)

ConversationResourceResponse object

Defines a response that contains a resource.

PROPERTY	TYPE	DESCRIPTION
activityId	string	ID of the activity.
id	string	ID of the resource.
serviceUrl	string	Service endpoint.

Error object

Defines an error.

PROPERTY	TYPE	DESCRIPTION
code	string	Error code.
message	string	A description of the error.

[Back to Schema table](#)

Entity object

Defines an entity object.

PROPERTY	TYPE	DESCRIPTION
type	string	Entity type. Typically contain types from schema.org.

ErrorResponse object

Defines an HTTP API response.

PROPERTY	TYPE	DESCRIPTION
error	Error	An Error object that contains information about the error.

[Back to Schema table](#)

Fact object

Defines a key-value pair that contains a fact.

PROPERTY	TYPE	DESCRIPTION
key	string	Name of the fact. For example, Check-in . The key is used as a label when displaying the fact's value.
value	string	Value of the fact. For example, 10 October 2016 .

[Back to Schema table](#)

Geocoordinates object

Defines a geographical location using World Geodetic System (WSG84) coordinates.

PROPERTY	TYPE	DESCRIPTION
elevation	number	Elevation of the location.
name	string	Name of the location.
latitude	number	Latitude of the location.
longitude	number	Longitude of the location.
type	string	The type of this object. Always set to GeoCoordinates .

[Back to Schema table](#)

HeroCard object

Defines a card with a large image, title, text, and action buttons.

PROPERTY	TYPE	DESCRIPTION
buttons	CardAction[]	Array of CardAction objects that enable the user to perform one or more actions. The channel determines the number of buttons that you may specify.
images	CardImage[]	Array of CardImage objects that specifies the image to display on the card. A Hero card contains only one image.
subtitle	string	Subtitle to display under the card's title.

PROPERTY	TYPE	DESCRIPTION
tap	CardAction	A CardAction object that specifies the action to perform if the user taps or clicks the card. This can be the same action as one of the buttons or a different action.
text	string	Description or prompt to display under the card's title or subtitle.
title	string	Title of the card.

[Back to Schema table](#)

Identification object

Identifies a resource.

PROPERTY	TYPE	DESCRIPTION
id	string	ID that uniquely identifies the resource.

[Back to Schema table](#)

MediaEventValue object

Supplementary parameter for media events.

PROPERTY	TYPE	DESCRIPTION
cardValue	object	Callback parameter specified in the Value field of the media card that originated this event.

MediaUrl object

Defines the URL to a media file's source.

PROPERTY	TYPE	DESCRIPTION
profile	string	Hint that describes the media's content.
url	string	URL to the source of the media file.

[Back to Schema table](#)

Mention object

Defines a user or bot that was mentioned in the conversation.

PROPERTY	TYPE	DESCRIPTION

PROPERTY	TYPE	DESCRIPTION
mentioned	ChannelAccount	A ChannelAccount object that specifies the user or the bot that was mentioned. Note that some channels such as Slack assign names per conversation, so it is possible that your bot's mentioned name (in the message's recipient property) may be different from the handle that you specified when you registered your bot. However, the account IDs for both would be the same.
text	string	The user or bot as mentioned in the conversation. For example, if the message is "@ColorBot pick me a new color," this property would be set to @ColorBot . Not all channels set this property.
type	string	This object's type. Always set to Mention .

[Back to Schema table](#)

MessageReaction object

Defines a reaction to a message.

PROPERTY	TYPE	DESCRIPTION
type	string	Type of reaction.

Place object

Defines a place that was mentioned in the conversation.

PROPERTY	TYPE	DESCRIPTION
address	object	Address of a place. This property can be a string or a complex object of type PostalAddress .
geo	GeoCoordinates	A GeoCoordinates object that specifies the geographical coordinates of the place.
hasMap	object	Map to the place. This property can be a string (URL) or a complex object of type Map .
name	string	Name of the place.
type	string	This object's type. Always set to Place .

[Back to Schema table](#)

ReceiptCard object

Defines a card that contains a receipt for a purchase.

PROPERTY	TYPE	DESCRIPTION
buttons	CardAction[]	Array of CardAction objects that enable the user to perform one or more actions. The channel determines the number of buttons that you may specify.
facts	Fact[]	Array of Fact objects that specify information about the purchase. For example, the list of facts for a hotel stay receipt might include the check-in date and check-out date. The channel determines the number of facts that you may specify.
items	ReceiptItem[]	Array of ReceiptItem objects that specify the purchased items
tap	CardAction	A CardAction object that specifies the action to perform if the user taps or clicks the card. This can be the same action as one of the buttons or a different action.
tax	string	A currency-formatted string that specifies the amount of tax applied to the purchase.
title	string	Title displayed at the top of the receipt.
total	string	A currency-formatted string that specifies the total purchase price, including all applicable taxes.
vat	string	A currency-formatted string that specifies the amount of value added tax (VAT) applied to the purchase price.

[Back to Schema table](#)

ReceiptItem object

Defines a line item within a receipt.

PROPERTY	TYPE	DESCRIPTION
image	CardImage	A CardImage object that specifies thumbnail image to display next to the line item.

PROPERTY	TYPE	DESCRIPTION
price	string	A currency-formatted string that specifies the total price of all units purchased.
quantity	string	A numeric string that specifies the number of units purchased.
subtitle	string	Subtitle to be displayed under the line item's title.
tap	CardAction	A CardAction object that specifies the action to perform if the user taps or clicks the line item.
text	string	Description of the line item.
title	string	Title of the line item.

[Back to Schema table](#)

ResourceResponse object

Defines a response that contains a resource ID.

PROPERTY	TYPE	DESCRIPTION
id	string	ID that uniquely identifies the resource.

[Back to Schema table](#)

SignInCard object

Defines a card that lets a user sign in to a service.

PROPERTY	TYPE	DESCRIPTION
buttons	CardAction[]	Array of CardAction objects that enable the user to sign in to a service. The channel determines the number of buttons that you may specify.
text	string	Description or prompt to include on the sign in card.

[Back to Schema table](#)

SuggestedActions object

Defines the options from which a user can choose.

PROPERTY	TYPE	DESCRIPTION

PROPERTY	TYPE	DESCRIPTION
actions	CardAction[]	Array of CardAction objects that define the suggested actions.
to	string[]	Array of strings that contains the IDs of the recipients to whom the suggested actions should be displayed.

[Back to Schema table](#)

ThumbnailCard object

Defines a card with a thumbnail image, title, text, and action buttons.

PROPERTY	TYPE	DESCRIPTION
buttons	CardAction[]	Array of CardAction objects that enable the user to perform one or more actions. The channel determines the number of buttons that you may specify.
images	CardImage[]	Array of CardImage objects that specify thumbnail images to display on the card. The channel determines the number of thumbnail images that you may specify.
subtitle	string	Subtitle to display under the card's title.
tap	CardAction	A CardAction object that specifies the action to perform if the user taps or clicks the card. This can be the same action as one of the buttons or a different action.
text	string	Description or prompt to display under the card's title or subtitle.
title	string	Title of the card.

[Back to Schema table](#)

ThumbnailUrl object

Defines the URL to an image's source.

PROPERTY	TYPE	DESCRIPTION
alt	string	Description of the image. You should include the description to support accessibility.

PROPERTY	TYPE	DESCRIPTION
url	string	URL to the source of the image or the base64 binary of the image (for example, data:image/png;base64,iVBORw0KGgo...).

[Back to Schema table](#)

VideoCard object

Defines a card that can play videos.

PROPERTY	TYPE	DESCRIPTION
aspect	string	Aspect ratio of a video (e.g.: 16:9, 4:3).
autoloop	boolean	Flag that indicates whether to replay the list of videos when the last one ends. Set this property to true to automatically replay the videos; otherwise, false . The default value is true .
autoplay	boolean	Flag that indicates whether to automatically play the videos when the card is displayed. Set this property to true to automatically play the videos; otherwise, false . The default value is true .
buttons	CardAction[]	Array of CardAction objects that enable the user to perform one or more actions. The channel determines the number of buttons that you may specify.
image	ThumbnailUrl	A ThumbnailUrl object that specifies the image to display on the card.
media	MediaUrl[]	Array of MediaUrl objects that specifies the list of videos to play.
shareable	boolean	Flag that indicates whether the videos may be shared with others. Set this property to true if the videos may be shared; otherwise, false . The default value is true .
subtitle	string	Subtitle to display under the card's title.
text	string	Description or prompt to display under the card's title or subtitle.
title	string	Title of the card.

PROPERTY	TYPE	DESCRIPTION
value	object	Supplementary parameter for this card

[Back to Schema table](#)

Key concepts in Direct Line API 3.0

7/26/2017 • 2 min to read • [Edit Online](#)

You can enable communication between your bot and your own client application by using the Direct Line API. This article introduces key concepts in Direct Line API 3.0 and provides information about relevant developer resources.

Authentication

Direct Line API 3.0 requests can be authenticated either by using a **secret** that you obtain from the Direct Line channel configuration page in the [Bot Framework Portal](#) or by using a **token** that you obtain at runtime. For more information, see [Authentication](#).

Starting a conversation

Direct Line conversations are explicitly opened by clients and may run as long as the bot and client participate and have valid credentials. For more information, see [Start a conversation](#).

Sending messages

Using Direct Line API 3.0, a client can send messages to your bot by issuing `HTTP POST` requests. A client may send a single message per request. For more information, see [Send an activity to the bot](#).

Receiving messages

Using Direct Line API 3.0, a client can receive messages from your bot either via `WebSocket` stream or by issuing `HTTP GET` requests. Using either of these techniques, a client may receive multiple messages from the bot at a time as part of an `ActivitySet`. For more information, see [Receive activities from the bot](#).

Developer resources

Client libraries

The Bot Framework provides client libraries that facilitate access to Direct Line API 3.0 via C# and Node.js.

- To use the .NET client library within a Visual Studio project, install the [Microsoft.Bot.Connector.DirectLine NuGet package](#).
- To use the Node.js client library, install the `botframework-directlinejs` library using [NPM](#) (or [download](#) the source).

As an alternative to using the C# or Node.js client libraries, you can generate your own client library in the language of your choice by using the [Direct Line API 3.0 Swagger file](#).

Sample code

The [BotBuilder-Samples](#) GitHub repo contains multiple samples that show how to use Direct Line API 3.0 with C# and Node.js.

SAMPLE	LANGUAGE	DESCRIPTION
--------	----------	-------------

SAMPLE	LANGUAGE	DESCRIPTION
Direct Line Bot Sample	C#	A sample bot and a custom client communicating to each other using the Direct Line API.
Direct Line Bot Sample (using client WebSockets)	C#	A sample bot and a custom client communicating to each other using the Direct Line API and WebSockets.
Direct Line Bot Sample	JavaScript	A sample bot and a custom client communicating to each other using the Direct Line API.
Direct Line Bot Sample (using client WebSockets)	JavaScript	A sample bot and a custom client communicating to each other using the Direct Line API and WebSockets.

Web chat control

The Bot Framework provides a control that enables you to embed a Direct-Line-powered bot into your client application. For more information, see the [Microsoft Bot Framework WebChat control](#).

Authentication

7/26/2017 • 3 min to read • [Edit Online](#)

A client can authenticate requests to Direct Line API 3.0 either by using a **secret** that you [obtain from the Direct Line channel configuration page](#) in the Bot Framework Portal or by using a **token** that you obtain at runtime. The secret or token should be specified in the `Authorization` header of each request, using this format:

```
Authorization: Bearer SECRET_OR_TOKEN
```

Secrets and tokens

A Direct Line **secret** is a master key that can be used to access any conversation that belongs to the associated bot. A **secret** can also be used to obtain a **token**. Secrets do not expire.

A Direct Line **token** is a key that can be used to access a single conversation. A token expires but can be refreshed.

If you're creating a service-to-service application, specifying the **secret** in the `Authorization` header of Direct Line API requests may be simplest approach. If you're writing an application where the client runs in a web browser or mobile app, you may want to exchange your secret for a token (which only works for a single conversation and will expire unless refreshed) and specify the **token** in the `Authorization` header of Direct Line API requests.

Choose the security model that works best for you.

NOTE

Your Direct Line client credentials are different from your bot's credentials. This enables you to revise your keys independently and lets you share client tokens without disclosing your bot's password.

Get a Direct Line secret

You can [obtain a Direct Line secret](#) via the Direct Line channel configuration page for your bot in the [Bot Framework Portal](#):

Configure Direct Line



+ Add new site KB_Client_1 Disable |

KB_Client_1

Secret keys

Hide | Regenerate

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
 Show | Regenerate

Version
Select which versions of the Direct Line protocol are enabled on this site. More information about these versions can be found in the [Direct Line reference documentation](#).

Generate a Direct Line token

To generate a Direct Line token that can be used to access a single conversation, first obtain the Direct Line secret from the Direct Line channel configuration page in the [Bot Framework Portal](#). Then issue this request to exchange your Direct Line secret for a Direct Line token:

```
POST https://directline.botframework.com/v3/directline/tokens/generate
Authorization: Bearer SECRET
```

In the `Authorization` header of this request, replace **SECRET** with the value of your Direct Line secret.

The following snippets provide an example of the Generate Token request and response.

Request

```
POST https://directline.botframework.com/v3/directline/tokens/generate
Authorization: Bearer RCurR_XV9ZA.cwA.BKA.iaJrC8xpy8qb0F5xnR2vtCX7CZj0LdjAPGfiCpg4Fv0
```

Response

If the request is successful, the response contains a `token` that is valid for one conversation and an `expires_in` value that indicates the number of seconds until the token expires. For the token to remain useful, you must [refresh the token](#) before it expires.

```
HTTP/1.1 200 OK
[other headers]
```

```
{
  "conversationId": "abc123",
  "token": "RCurR_XV9ZA.cwA.BKA.iaJrC8xpy8qb0F5xnR2vtCX7CZj0LdjAPGfiCpg4Fv0y8qb0F5xPGfiCpg4Fv0y8qqb0F5x8qb0F5xn",
  "expires_in": 1800
}
```

Generate Token versus Start Conversation

The Generate Token operation (`POST /v3/directline/tokens/generate`) is similar to the [Start Conversation](#) operation (`POST /v3/directline/conversations`) in that both operations return a `token` that can be used to access a single conversation. However, unlike the Start Conversation operation, the Generate Token operation does not start the conversation, does not contact the bot, and does not create a streaming WebSocket URL.

If you plan to distribute the token to clients and want them to initiate the conversation, use the Generate Token operation. If you intend to start the conversation immediately, use the [Start Conversation](#) operation instead.

Refresh a Direct Line token

A Direct Line token can be refreshed an unlimited amount of times, as long as it has not expired. An expired token cannot be refreshed. To refresh a Direct Line token, issue this request:

```
POST https://directline.botframework.com/v3/directline/tokens/refresh
Authorization: Bearer TOKEN_TO_BE_REFRESHED
```

In the `Authorization` header of this request, replace **TOKEN_TO_BE_REFRESHED** with the Direct Line token that you want to refresh.

The following snippets provide an example of the Refresh Token request and response.

Request

```
POST https://directline.botframework.com/v3/directline/tokens/refresh
Authorization: Bearer
CurR_XV9ZA.cwA.BKA.iaJrC8xpy8qb0F5xnR2vtCX7CZj0LdjAPGfiCpg4Fv0y8qb0F5xPGfiCpg4Fv0y8qqb0F5x8qb0F5xn
```

Response

If the request is successful, the response contains a new `token` that is valid for the same conversation as the previous token and an `expires_in` value that indicates the number of seconds until the new token expires. For the new token to remain useful, you must [refresh the token](#) before it expires.

```
HTTP/1.1 200 OK
[other headers]
```

```
{
  "conversationId": "abc123",
  "token": "RCurR_XV9ZA.cwA.BKA.y8qb0F5xPGfiCpg4Fv0y8qqb0F5x8qb0F5xniaJrC8xpy8qb0F5xnR2vtCX7CZj0LdjAPGfiCpg4Fv0",
  "expires_in": 1800
}
```

Additional resources

- [Key concepts](#)
- [Connect a bot to Direct Line](#)

Start a conversation

7/26/2017 • 1 min to read • [Edit Online](#)

Direct Line conversations are explicitly opened by clients and may run as long as the bot and client participate and have valid credentials. While the conversation is open, both the bot and client may send messages. More than one client may connect to a given conversation and each client may participate on behalf of multiple users.

Open a new conversation

To open a new conversation with a bot, issue this request:

```
POST https://directline.botframework.com/v3/directline/conversations
Authorization: Bearer SECRET_OR_TOKEN
```

The following snippets provide an example of the Start Conversation request and response.

Request

```
POST https://directline.botframework.com/v3/directline/conversations
Authorization: Bearer
RCurR_XV9ZA.cwA.BKA.iaJrc8xpy8qb0F5xnR2vtCX7CZj0LdjAPGfiCpg4Fv0y8qb0F5xPGfiCpg4Fv0y8qqb0F5x8qb0F5xn
```

Response

If the request is successful, the response will contain an ID for the conversation, a token, a value that indicates the number of seconds until the token expires, and a stream URL that the client may use to [receive activities via WebSocket stream](#).

```
HTTP/1.1 201 Created
[other headers]
```

```
{
  "conversationId": "abc123",
  "token": "RCurR_XV9ZA.cwA.BKA.iaJrc8xpy8qb0F5xnR2vtCX7CZj0LdjAPGfiCpg4Fv0y8qb0F5xPGfiCpg4Fv0y8qqb0F5x8qb0F5xn",
  "expires_in": 1800,
  "streamUrl": "https://directline.botframework.com/v3/directline/conversations/abc123/stream?
t=RCurR_XV9ZA.cwA..."
```

Typically, a Start Conversation request is used to open a new conversation and an **HTTP 201** status code is returned if the new conversation is successfully started. However, if a client submits a Start Conversation request with a Direct Line token in the `Authorization` header that has previously been used to start a conversation using the Start Conversation operation, an **HTTP 200** status code will be returned to indicate that the request was acceptable but no conversation was created (as it already existed).

TIP

You have 60 seconds to connect to the WebSocket stream URL. If the connection cannot be established during this time, you can [reconnect to the conversation](#) to generate a new stream URL.

Start Conversation versus Generate Token

The Start Conversation operation (`POST /v3/directline/conversations`) is similar to the [Generate Token](#) operation (`POST /v3/directline/tokens/generate`) in that both operations return a `token` that can be used to access a single conversation. However, the Start Conversation operation also starts the conversation, contacts the bot, and creates a WebSocket stream URL, whereas the Generate Token operation does none of these things.

If you intend to start the conversation immediately, use the Start Conversation operation. If you plan to distribute the token to clients and want them to initiate the conversation, use the [Generate Token](#) operation instead.

Additional resources

- [Key concepts](#)
- [Authentication](#)
- [Receive activities via WebSocket stream](#)
- [Reconnect to a conversation](#)

Reconnect to a conversation

7/26/2017 • 1 min to read • [Edit Online](#)

If a client is using the [WebSocket interface](#) to receive messages but loses its connection, it may need to reconnect. In this scenario, the client must generate a new WebSocket stream URL that it can use to reconnect to the conversation.

Generate a new WebSocket stream URL

To generate a new WebSocket stream URL that can be used to reconnect to an existing conversation, issue this request:

```
GET https://directline.botframework.com/v3/directline/conversations/{conversationId}?watermark={watermark_value}
Authorization: Bearer SECRET_OR_TOKEN
```

In this request URI, replace **{conversationId}** with the conversation ID and replace **{watermark_value}** with the watermark value (if the `watermark` parameter is supplied). The `watermark` parameter is optional. If the `watermark` parameter is specified in the request URI, the conversation replays from the watermark, guaranteeing that no messages are lost. If the `watermark` parameter is omitted from the request URI, only messages received after the reconnection request are replayed.

The following snippets provide an example of the Reconnect request and response.

Request

```
GET https://directline.botframework.com/v3/directline/conversations/abc123?watermark=0000a-42
Authorization: Bearer
RCurR_XV9ZA.cwA.BKA.iaJrC8xpy8qb0F5xnR2vtCX7CZj0LdjAPGfiCpg4Fv0y8qb0F5xPGfiCpg4Fv0y8qb0F5x8qb0F5xn
```

Response

If the request is successful, the response will contain an ID for the conversation, a token, and a new WebSocket stream URL.

```
HTTP/1.1 200 OK
[other headers]
```

```
{
  "conversationId": "abc123",
  "token": "RCurR_XV9ZA.cwA.BKA.iaJrC8xpy8qb0F5xnR2vtCX7CZj0LdjAPGfiCpg4Fv0y8qb0F5xPGfiCpg4Fv0y8qb0F5x8qb0F5xn",
  "streamUrl": "https://directline.botframework.com/v3/directline/conversations/abc123/stream?watermark=000a-4&t=RCurR_XV9ZA.cwA..."
}
```

Reconnect to the conversation

The client must use the new WebSocket stream URL to [reconnect to the conversation](#) within 60 seconds. If the connection cannot be established during this time, the client must issue another Reconnect request to generate a new stream URL.

Additional resources

- [Key concepts](#)
- [Authentication](#)
- [Receive activities via WebSocket stream](#)

Send an activity to the bot

7/26/2017 • 4 min to read • [Edit Online](#)

Using the Direct Line 3.0 protocol, clients and bots may exchange several different types of [activities](#), including **message** activities, **typing** activities, and custom activities that the bot supports. A client may send a single activity per request.

Send an activity

To send an activity to the bot, the client must create an [Activity](#) object to define the activity and then issue a `POST` request to `https://directline.botframework.com/v3/directline/conversations/{conversationId}/activities`, specifying the Activity object in the body of the request.

The following snippets provide an example of the Send Activity request and response.

Request

```
POST https://directline.botframework.com/v3/directline/conversations/abc123/activities
Authorization: Bearer RCurR_XV9ZA.cwA.BKA.iaJrC8xpy8qb0F5xnR2vtCX7CZj0LdjAPGfiCpg4Fv0
[other headers]
```

```
{
  "type": "message",
  "from": {
    "id": "user1"
  },
  "text": "hello"
}
```

Response

When the activity is delivered to the bot, the service responds with an HTTP status code that reflects the bot's status code. If the bot generates an error, an HTTP 502 response ("Bad Gateway") is returned to the client in response to its Send Activity request. If the POST is successful, the response contains a JSON payload that specifies the ID of the Activity that was sent to the bot.

```
HTTP/1.1 200 OK
[other headers]
```

```
{
  "id": "0001"
}
```

Total time for the Send Activity request/response

The total time to POST a message to a Direct Line conversation is the sum of the following:

- Transit time for the HTTP request to travel from the client to the Direct Line service
- Internal processing time within Direct Line (typically less than 120ms)
- Transit time from the Direct Line service to the bot
- Processing time within the bot

- Transit time for the HTTP response to travel back to the client

Send attachment(s) to the bot

In some situations, a client may need to send attachments to the bot such as images or documents. A client may send attachments to the bot either by [specifying the URL\(s\)](#) of the attachment(s) within the [Activity](#) object that it sends using `POST /v3/directline/conversations/{conversationId}/activities` or by [uploading attachment\(s\)](#) using `POST /v3/directline/conversations/{conversationId}/upload`.

Send attachment(s) by URL

To send one or more attachments as part of the [Activity](#) object using

`POST /v3/directline/conversations/{conversationId}/activities`, simply include one or more [Attachment](#) objects within the Activity object and set the `contentUrl` property of each Attachment object to specify the HTTP, HTTPS, or `data` URI of the attachment.

Send attachment(s) by upload

Often, a client may have image(s) or document(s) on a device that it wants to send to the bot, but no URLs corresponding to those files. In this situation, a client can issue a

`POST /v3/directline/conversations/{conversationId}/upload` request to send attachments to the bot by upload. The format and contents of the request will depend upon whether the client is [sending a single attachment](#) or [sending multiple attachments](#).

Send a single attachment by upload

To send a single attachment by upload, issue this request:

```
POST https://directline.botframework.com/v3/directline/conversations/{conversationId}/upload?userId={userId}
Authorization: Bearer SECRET_OR_TOKEN
Content-Type: TYPE_OF_ATTACHMENT
Content-Disposition: ATTACHMENT_INFO
[other headers]

[file content]
```

In this request URI, replace `{conversationId}` with the ID of the conversation and `{userId}` with the ID of the user that is sending the message. The `userId` parameter is required. In the request headers, set `Content-Type` to specify the attachment's type and set `Content-Disposition` to specify the attachment's filename.

The following snippets provide an example of the Send (single) Attachment request and response.

Request

```
POST https://directline.botframework.com/v3/directline/conversations/abc123/upload?userId=user1
Authorization: Bearer RCurR_XV9ZA.cwA.BKA.iaJrC8xpy8qb0F5xnR2vtCX7CZj0LdjAPGfiCpg4Fv0
Content-Type: image/jpeg
Content-Disposition: name="file"; filename="badjokeee1.jpg"
[other headers]

[JPEG content]
```

Response

If the request is successful, a **message** Activity is sent to the bot when the upload completes and the response that the client receives will contain the ID of the Activity that was sent.

```
HTTP/1.1 200 OK
[other headers]
```

```
{
  "id": "0003"
}
```

Send multiple attachments by upload

To send multiple attachments by upload, `POST` a multipart request to the

`/v3/directline/conversations/{conversationId}/upload` endpoint. Set the `Content-Type` header of the request to `multipart/form-data` and include the `Content-Type` header and `Content-Disposition` header for each part to specify each attachment's type and filename. In the request URI, set the `userId` parameter to the ID of the user that is sending the message.

You may include an [Activity](#) object within the request by adding a part that specifies the `Content-Type` header value `application/vnd.microsoft.activity`. If the request includes an Activity, the attachments that are specified by other parts of the payload are added as attachments to that Activity before it is sent. If the request does not include an Activity, an empty Activity is created to serve as the container in which the specified attachments are sent.

The following snippets provide an example of the Send (multiple) Attachments request and response. In this example, the request sends a message that contains some text and a single image attachment. Additional parts could be added to the request to include multiple attachments in this message.

Request

```
POST https://directline.botframework.com/v3/directline/conversations/abc123/upload?userId=user1
Authorization: Bearer RCurr_XV9ZA.cwA.BKA.iaJrC8xpy8qbOF5xnR2vtCX7CZj0LdjAPGfiCpg4Fv0
Content-Type: multipart/form-data; boundary=----DD4E5147-E865-4652-B662-F223701A8A89
[other headers]

----DD4E5147-E865-4652-B662-F223701A8A89
Content-Type: image/jpeg
Content-Disposition: form-data; name="file"; filename="badjokeeel.jpg"
[other headers]

[JPEG content]

----DD4E5147-E865-4652-B662-F223701A8A89
Content-Type: application/vnd.microsoft.activity
[other headers]

{
  "type": "message",
  "from": {
    "id": "user1"
  },
  "text": "Hey I just IM'd you\n\nand this is crazy\n\nbut here's my webhook\n\nso POST me maybe"
}
----DD4E5147-E865-4652-B662-F223701A8A89
```

Response

If the request is successful, a message Activity is sent to the bot when the upload completes and the response that the client receives will contain the ID of the Activity that was sent.

```
HTTP/1.1 200 OK
[other headers]
```

```
{  
  "id": "0004"  
}
```

Additional resources

- [Key concepts](#)
- [Authentication](#)
- [Start a conversation](#)
- [Reconnect to a conversation](#)
- [Receive activities from the bot](#)
- [End a conversation](#)

Receive activities from the bot

7/26/2017 • 4 min to read • [Edit Online](#)

Using the Direct Line 3.0 protocol, clients can receive activities via `WebSocket` stream or retrieve activities by issuing `HTTP GET` requests.

WebSocket vs HTTP GET

A streaming WebSocket efficiently pushes messages to clients, whereas the GET interface enables clients to explicitly request messages. Although the WebSocket mechanism is often preferred due to its efficiency, the GET mechanism can be useful for clients that are unable to use WebSockets or for clients that want to retrieve conversation history.

Not all [activity types](#) are available both via WebSocket and via HTTP GET. The following table describes the availability of the various activity types for clients that use the Direct Line protocol.

ACTIVITY TYPE	AVAILABILITY
message	HTTP GET and WebSocket
typing	WebSocket only
conversationUpdate	Not sent/received via client
contactRelationUpdate	Not supported in Direct Line
endOfConversation	HTTP GET and WebSocket
all other activity types	HTTP GET and WebSocket

Receive activities via WebSocket stream

When a client sends a [Start Conversation](#) request to open a conversation with a bot, the service's response includes a `streamUrl` property that the client can subsequently use to connect via WebSocket. The stream URL is preauthorized and therefore the client's request to connect via WebSocket does NOT require an `Authorization` header.

The following example shows a request that uses a `streamUrl` to connect via WebSocket.

```
-- connect to wss://directline.botframework.com --
GET /v3/directline/conversations/abc123/stream?t=RCurR_XV9ZA.cwA...
Upgrade: websocket
Connection: upgrade
[other headers]
```

The service responds with status code HTTP 101 ("Switching Protocols").

```
HTTP/1.1 101 Switching Protocols
[other headers]
```

Receive messages

After it connects via WebSocket, a client may receive these types of messages from the Direct Line service:

- A message that contains an [ActivitySet](#) that includes one or more activities and a watermark (described below).
- An empty message, which the Direct Line service uses to ensure the connection is still valid.
- Additional types, to be defined later. These types are identified by the properties in the JSON root.

An [ActivitySet](#) contains messages sent by the bot and by all users in the conversation. The following example shows an [ActivitySet](#) that contains a single message.

```
{  
    "activities": [  
        {  
            "type": "message",  
            "channelId": "directline",  
            "conversation": {  
                "id": "abc123"  
            },  
            "id": "abc123|0000",  
            "from": {  
                "id": "user1"  
            },  
            "text": "hello"  
        }  
    ],  
    "watermark": "0000a-42"  
}
```

Process messages

A client should keep track of the [watermark](#) value that it receives in each [ActivitySet](#), so that it may use the watermark to guarantee that no messages are lost if it loses its connection and needs to [reconnect to the conversation](#). If the client receives an [ActivitySet](#) wherein the [watermark](#) property is [null](#) or missing, it should ignore that value and not overwrite the prior watermark that it received.

A client should ignore empty messages that it receives from the Direct Line service.

A client may send empty messages to the Direct Line service to verify connectivity. The Direct Line service will ignore empty messages that it receives from the client.

The Direct Line service may forcibly close the WebSocket connection under certain conditions. If the client has not received an [endOfConversation](#) activity, it may [generate a new WebSocket stream URL](#) that it can use to reconnect to the conversation.

The WebSocket stream contains live updates and very recent messages (since the call to connect via WebSocket was issued) but it does not include messages that were sent prior to the most recent [POST](#) to [/v3/directline/conversations/{id}](#). To retrieve messages that were sent earlier in the conversation, use [HTTP GET](#) as described below.

Retrieve activities with HTTP GET

Clients that are unable to use WebSockets or clients that want to get conversation history can retrieve activities by using [HTTP GET](#).

To retrieve messages for a specific conversation, issue a [GET](#) request to the [/v3/directline/conversations/{conversationId}/activities](#) endpoint, optionally specifying the [watermark](#) parameter to indicate the most recent message seen by the client.

The following snippets provide an example of the Get Conversation Activities request and response. The Get

Conversation Activities response contains `watermark` as a property of the `ActivitySet`. Clients should page through the available activities by advancing the `watermark` value until no activities are returned.

Request

```
GET https://directline.botframework.com/v3/directline/conversations/abc123/activities?watermark=0001a-94
Authorization: Bearer RCurR_XV9ZA.cwA.BKA.iaJrC8xpy8qb0F5xnR2vtCX7CZj0LdjAPGfiCpg4Fv0
```

Response

```
HTTP/1.1 200 OK
[other headers]
```

```
{
  "activities": [
    {
      "type": "message",
      "channelId": "directline",
      "conversation": {
        "id": "abc123"
      },
      "id": "abc123|0000",
      "from": {
        "id": "user1"
      },
      "text": "hello"
    },
    {
      "type": "message",
      "channelId": "directline",
      "conversation": {
        "id": "abc123"
      },
      "id": "abc123|0001",
      "from": {
        "id": "bot1"
      },
      "text": "Nice to see you, user1!"
    }
  ],
  "watermark": "0001a-95"
}
```

Timing considerations

Most clients wish to retain a complete message history. Even though Direct Line is a multi-part protocol with potential timing gaps, the protocol and service is designed to make it easy to build a reliable client.

- The `watermark` property that is sent in the WebSocket stream and Get Conversation Activities response is reliable. A client will not miss any messages as long as it replays the watermark verbatim.
- When a client starts a conversation and connects to the WebSocket stream, any activities that are sent after the POST but before the socket is opened are replayed before new activities.
- When a client issues a Get Conversation Activities request (to refresh history) while it is connected to the WebSocket stream, activities may be duplicated across both channels. Clients should keep track of all known activity IDs so that they are able to reject duplicate activities, should they occur.

Clients that poll using `HTTP GET` should choose a polling interval that matches their intended use.

- Service-to-service applications often use a polling interval of 5s or 10s.
- Client-facing applications often use a polling interval of 1s, and issue an additional request ~300ms after every message that the client sends (to rapidly retrieve a bot's response). This 300ms delay should be adjusted based on the bot's speed and transit time.

Additional resources

- [Key concepts](#)
- [Authentication](#)
- [Start a conversation](#)
- [Reconnect to a conversation](#)
- [Send an activity to the bot](#)
- [End a conversation](#)

End a conversation

7/26/2017 • 1 min to read • [Edit Online](#)

Either a client or a bot may signal the end of a Direct Line conversation by sending an **endOfConversation** activity.

Send an endOfConversation activity

An **endOfConversation** activity ends communication between bot and client. After an **endOfConversation** activity has been sent, the client may still [retrieve messages](#) using `HTTP GET`, but neither the client nor the bot can send any additional messages to the conversation.

To end a conversation, simply issue a POST request to send an **endOfConversation** activity.

Request

```
POST https://directline.botframework.com/v3/directline/conversations/abc123/activities
Authorization: Bearer RCurR_XV9ZA.cwA.BKA.iaJrC8xpy8qbOF5xnR2vtCX7CZj0LdjAPGfiCpg4Fv0
[other headers]
```

```
{
  "type": "endOfConversation",
  "from": {
    "id": "user1"
  }
}
```

Response

If the request is successful, the response will contain an ID for the activity that was sent.

```
HTTP/1.1 200 OK
[other headers]
```

```
{
  "id": "0004"
}
```

Additional resources

- [Key concepts](#)
- [Authentication](#)
- [Send an activity to the bot](#)

API reference - Direct Line API 3.0

7/26/2017 • 5 min to read • [Edit Online](#)

You can enable your client application to communicate with your bot by using Direct Line API 3.0. Direct Line API 3.0 uses industry-standard REST and JSON over HTTPS.

Base URI

To access Direct Line API 3.0, use this base URI for all API requests:

`https://directline.botframework.com`

Headers

In addition to the standard HTTP request headers, a Direct Line API request must include an `Authorization` header that specifies a secret or token to authenticate the client that is issuing the request. Specify the `Authorization` header using this format:

```
Authorization: Bearer SECRET_OR_TOKEN
```

For details about how to obtain a secret or token that your client can use to authenticate its Direct Line API requests, see [Authentication](#).

HTTP status codes

The [HTTP status code](#) that is returned with each response indicates the outcome of the corresponding request.

HTTP STATUS CODE	MEANING
200	The request succeeded.
201	The request succeeded.
202	The request has been accepted for processing.
204	The request succeeded but no content was returned.
400	The request was malformed or otherwise incorrect.
401	The client is not authorized to make the request. Often this status code occurs because the <code>Authorization</code> header is missing or malformed.
403	The client is not allowed to perform the requested operation. If the request specified a token that was previously valid but has expired, the <code>code</code> property of the Error that is returned within the ErrorResponse object is set to <code>TokenExpired</code> .
404	The requested resource was not found. Typically this status code indicates an invalid request URI.

HTTP STATUS CODE	MEANING
500	An internal server error occurred within the Direct Line service.
502	The bot is unavailable or returned an error. This is a common error code.

NOTE

HTTP status code 101 is used in the WebSocket connection path, although this is likely handled by your WebSocket client.

Errors

Any response that specifies an HTTP status code in the 4xx range or 5xx range will include an [ErrorResponse](#) object in the body of the response that provides information about the error. If you receive an error response in the 4xx range, inspect the [ErrorResponse](#) object to identify the cause of the error and resolve your issue prior to resubmitting the request.

NOTE

HTTP status codes and values specified in the `code` property inside the [ErrorResponse](#) object are stable. Values specified in the `message` property inside the [ErrorResponse](#) object may change over time.

The following snippets show an example request and the resulting error response.

Request

```
POST https://directline.botframework.com/v3/directline/conversations/abc123/activities
[detail omitted]
```

Response

```
HTTP/1.1 502 Bad Gateway
[other headers]
```

```
{
  "error": {
    "code": "BotRejectedActivity",
    "message": "Failed to send activity: bot returned an error"
  }
}
```

Token operations

Use these operations to create or refresh a token that a client can use to access a single conversation.

OPERATION	DESCRIPTION
Generate Token	Generate a token for a new conversation.
Refresh Token	Refresh a token.

Generate Token

Generates a token that is valid for one conversation.

POST /v3/directline/tokens/generate	
-------------------------------------	--

Request body	n/a
Returns	A Conversation object

Refresh Token

Refreshes the token.

POST /v3/directline/tokens/refresh	
------------------------------------	--

Request body	n/a
Returns	A Conversation object

Conversation operations

Use these operations to open a conversation with your bot and exchange activities between client and bot.

OPERATION	DESCRIPTION
Start Conversation	Opens a new conversation with the bot.
Get Conversation Information	Gets information about an existing conversation. This operation generates a new WebSocket stream URL that a client may use to reconnect to a conversation.
Get Activities	Retrieves activities from the bot.
Send an Activity	Sends an activity to the bot.
Upload and Send File(s)	Uploads and sends file(s) as attachment(s).

Start Conversation

Opens a new conversation with the bot.

POST /v3/directline/conversations	
-----------------------------------	--

Request body	n/a
Returns	A Conversation object

Get Conversation Information

Gets information about an existing conversation and also generates a new WebSocket stream URL that a client may

use to [reconnect](#) to a conversation. You may optionally supply the `watermark` parameter in the request URI to indicate the most recent message seen by the client.

```
GET /v3/directline/conversations/{conversationId}?watermark={watermark_value}
```

Request body	n/a
Returns	A Conversation object

Get Activities

Retrieves activities from the bot for the specified conversation. You may optionally supply the `watermark` parameter in the request URI to indicate the most recent message seen by the client.

```
GET /v3/directline/conversations/{conversationId}/activities?watermark={watermark_value}
```

Request body	n/a
Returns	An ActivitySet object. The response contains <code>watermark</code> as a property of the ActivitySet object. Clients should page through the available activities by advancing the <code>watermark</code> value until no activities are returned.

Send an Activity

Sends an activity to the bot.

```
POST /v3/directline/conversations/{conversationId}/activities
```

Request body	An Activity object
Returns	A ResourceResponse that contains an <code>id</code> property which specifies the ID of the Activity that was sent to the bot.

Upload and Send File(s)

Uploads and sends file(s) as attachment(s). Set the `userId` parameter in the request URI to specify the ID of the user that is sending the attachment(s).

```
POST /v3/directline/conversations/{conversationId}/upload?userId={userId}
```

Request body	For a single attachment, populate the request body with the file contents. For multiple attachments, create a multipart request body that contains one part for each attachment, and also (optionally) one part for the Activity object that should serve as the container for the specified attachment(s). For more information, see Send an activity to the bot .
---------------------	-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

Returns	A ResourceResponse that contains an <code>id</code> property which specifies the ID of the Activity that was sent to the bot.
----------------	-----------------------------------------------------------------------------------------------------------------------------------------------

Schema

Direct Line 3.0 schema includes all of the objects that are defined by the [Bot Framework v3 schema](#) as well as the `ActivitySet` object and the `Conversation` object.

ActivitySet object

Defines a set of activities.

PROPERTY	TYPE	DESCRIPTION
activities	Activity[]	Array of Activity objects.
watermark	string	Maximum watermark of activities within the set. A client may use the <code>watermark</code> value to indicate the most recent message it has seen either when retrieving activities from the bot or when generating a new WebSocket stream URL .

Conversation object

Defines a Direct Line conversation.

PROPERTY	TYPE	DESCRIPTION
conversationId	string	ID that uniquely identifies the conversation for which the specified token is valid.
expires_in	number	Number of seconds until the token expires.
streamUrl	string	URL for the conversation's message stream.
token	string	Token that is valid for the specified conversation.

Activities

For each [Activity](#) that a client receives from a bot via Direct Line:

- Attachment cards are preserved.
- URLs for uploaded attachments are hidden with a private link.
- The `channelData` property is preserved without modification.

Clients may [receive](#) multiple activities from the bot as part of an [ActivitySet](#).

When a client sends an [Activity](#) to a bot via Direct Line:

- The `type` property specifies the type activity it is sending (typically **message**).
- The `from` property must be populated with a user ID, chosen by the client.
- Attachments may contain URLs to existing resources or URLs uploaded through the Direct Line attachment endpoint.
- The `channelData` property is preserved without modification.

Clients may [send](#) a single activity per request.

Key concepts in Direct Line API 1.1

7/26/2017 • 1 min to read • [Edit Online](#)

You can enable communication between your bot and your own client application by using the Direct Line API.

IMPORTANT

This article introduces key concepts in Direct Line API 1.1 and provides information about relevant developer resources. If you are creating a new connection between your client application and bot, use [Direct Line API 3.0](#) instead.

Authentication

Direct Line API 1.1 requests can be authenticated either by using a **secret** that you obtain from the Direct Line channel configuration page in the [Bot Framework Portal](#) or by using a **token** that you obtain at runtime. For more information, see [Authentication](#).

Starting a conversation

Direct Line conversations are explicitly opened by clients and may run as long as the bot and client participate and have valid credentials. For more information, see [Start a conversation](#).

Sending messages

Using Direct Line API 1.1, a client can send messages to your bot by issuing `HTTP POST` requests. A client may send a single message per request. For more information, see [Send a message to the bot](#).

Receiving messages

Using Direct Line API 1.1, a client can receive messages by polling with `HTTP GET` requests. In response to each request, a client may receive multiple messages from the bot as part of a `MessageSet`. For more information, see [Receive messages from the bot](#).

Developer resources

Client library

The Bot Framework provides a client library that facilitates access to Direct Line API 1.1 via C#. To use the client library within a Visual Studio project, install the `Microsoft.Bot.Connector.DirectLine v1.x NuGet package`.

As an alternative to using the C# client library, you can generate your own client library in the language of your choice by using the [Direct Line API 1.1 Swagger file](#).

Web chat control

The Bot Framework provides a control that enables you to embed a Direct-Line-powered bot into your client application. For more information, see the [Microsoft Bot Framework WebChat control](#).

Authentication

7/26/2017 • 3 min to read • [Edit Online](#)

IMPORTANT

This article describes authentication in Direct Line API 1.1. If you are creating a new connection between your client application and bot, use [Direct Line API 3.0](#) instead.

A client can authenticate requests to Direct Line API 1.1 either by using a **secret** that you [obtain from the Direct Line channel configuration page](#) in the Bot Framework Portal or by using a **token** that you obtain at runtime.

The secret or token should be specified in the `Authorization` header of each request, using either the "Bearer" scheme or the "BotConnector" scheme.

Bearer scheme:

```
Authorization: Bearer SECRET_OR_TOKEN
```

BotConnector scheme:

```
Authorization: BotConnector SECRET_OR_TOKEN
```

Secrets and tokens

A Direct Line **secret** is a master key that can be used to access any conversation that belongs to the associated bot. A **secret** can also be used to obtain a **token**. Secrets do not expire.

A Direct Line **token** is a key that can be used to access a single conversation. A token expires but can be refreshed.

If you're creating a service-to-service application, specifying the **secret** in the `Authorization` header of Direct Line API requests may be simplest approach. If you're writing an application where the client runs in a web browser or mobile app, you may want to exchange your secret for a token (which only works for a single conversation and will expire unless refreshed) and specify the **token** in the `Authorization` header of Direct Line API requests. Choose the security model that works best for you.

NOTE

Your Direct Line client credentials are different from your bot's credentials. This enables you to revise your keys independently and lets you share client tokens without disclosing your bot's password.

Get a Direct Line secret

You can [obtain a Direct Line secret](#) via the Direct Line channel configuration page for your bot in the [Bot Framework Portal](#):

Configure Direct Line



+ Add new site KB_Client_1 Disable |

KB_Client_1

Secret keys

Hide | Regenerate

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
 Show | Regenerate

Version
Select which versions of the Direct Line protocol are enabled on this site. More information about these versions can be found in the [Direct Line reference documentation](#).

Generate a Direct Line token

To generate a Direct Line token that can be used to access a single conversation, first obtain the Direct Line secret from the Direct Line channel configuration page in the [Bot Framework Portal](#). Then issue this request to exchange your Direct Line secret for a Direct Line token:

```
POST https://directline.botframework.com/api/tokens/conversation
Authorization: Bearer SECRET
```

In the `Authorization` header of this request, replace **SECRET** with the value of your Direct Line secret.

The following snippets provide an example of the Generate Token request and response.

Request

```
POST https://directline.botframework.com/api/tokens/conversation
Authorization: Bearer RCurR_XV9ZA.cwA.BKA.iaJrC8xpy8qb0F5xnR2vtCX7CZj0LdjAPGfiCpg4Fv0
```

Response

If the request is successful, the response contains a token that is valid for one conversation. The token will expire in 30 minutes. For the token to remain useful, you must [refresh the token](#) before it expires.

```
HTTP/1.1 200 OK
[other headers]

"RCurR_XV9ZA.cwA.BKA.iaJrC8xpy8qb0F5xnR2vtCX7CZj0LdjAPGfiCpg4Fv0y8qb0F5xPGfiCpg4Fv0y8qqb0F5x8qb0F5xn"
```

Generate Token versus Start Conversation

The Generate Token operation (`POST /api/tokens/conversation`) is similar to the [Start Conversation](#) operation (`POST /api/conversations`) in that both operations return a `token` that can be used to access a single conversation. However, unlike the Start Conversation operation, the Generate Token operation does not start the conversation or contact the bot.

If you plan to distribute the token to clients and want them to initiate the conversation, use the Generate Token

operation. If you intend to start the conversation immediately, use the [Start Conversation](#) operation instead.

Refresh a Direct Line token

A Direct Line token is valid for 30 minutes from the time it is generated and can be refreshed an unlimited amount of times, as long as it has not expired. An expired token cannot be refreshed. To refresh a Direct Line token, issue this request:

```
POST https://directline.botframework.com/api/tokens/{conversationId}/renew
Authorization: Bearer TOKEN_TO_BE_REFRESHED
```

In the URI of this request, replace **{conversationId}** with the ID of the conversation for which the token is valid and in the `Authorization` header of this request, replace **TOKEN_TO_BE_REFRESHED** with the Direct Line token that you want to refresh.

The following snippets provide an example of the Refresh Token request and response.

Request

```
POST https://directline.botframework.com/api/tokens/abc123/renew
Authorization: Bearer
CurR_XV9ZA.cwA.BKA.iaJrC8xpy8qb0F5xnR2vtCX7CZj0LdjAPGfiCpg4Fv0y8qb0F5xPGfiCpg4Fv0y8qqb0F5x8qb0F5xn
```

Response

If the request is successful, the response contains a new token that is valid for the same conversation as the previous token. The new token will expire in 30 minutes. For the new token to remain useful, you must [refresh the token](#) before it expires.

```
HTTP/1.1 200 OK
[other headers]

"RCurR_XV9ZA.cwA.BKA.y8qb0F5xPGfiCpg4Fv0y8qqb0F5x8qb0F5xniaJrC8xpy8qb0F5xnR2vtCX7CZj0LdjAPGfiCpg4Fv0"
```

Additional resources

- [Key concepts](#)
- [Connect a bot to Direct Line](#)

Start a conversation

7/26/2017 • 1 min to read • [Edit Online](#)

IMPORTANT

This article describes how to start a conversation using Direct Line API 1.1. If you are creating a new connection between your client application and bot, use [Direct Line API 3.0](#) instead.

Direct Line conversations are explicitly opened by clients and may run as long as the bot and client participate and have valid credentials. While the conversation is open, both the bot and client may send messages. More than one client may connect to a given conversation and each client may participate on behalf of multiple users.

Open a new conversation

To open a new conversation with a bot, issue this request:

```
POST https://directline.botframework.com/api/conversations  
Authorization: Bearer SECRET_OR_TOKEN
```

The following snippets provide an example of the Start Conversation request and response.

Request

```
POST https://directline.botframework.com/api/conversations  
Authorization: Bearer  
RCurR_XV9ZA.cwA.BKA.iaJrC8xpy8qb0F5xnR2vtCX7CZj0LdjAPGfiCpg4Fv0y8qb0F5xPGfiCpg4Fv0y8qqb0F5x8qb0F5xn
```

Response

If the request is successful, the response will contain an ID for the conversation, a token, and a value that indicates the number of seconds until the token expires.

```
HTTP/1.1 200 OK  
[other headers]
```

```
{  
  "conversationId": "abc123",  
  "token":  
    "RCurR_XV9ZA.cwA.BKA.iaJrC8xpy8qb0F5xnR2vtCX7CZj0LdjAPGfiCpg4Fv0y8qb0F5xPGfiCpg4Fv0y8qqb0F5x8qb0F5xn",  
  "expires_in": 1800  
}
```

Start Conversation versus Generate Token

The Start Conversation operation (`POST /api/conversations`) is similar to the [Generate Token](#) operation (`POST /api/tokens/conversation`) in that both operations return a `token` that can be used to access a single conversation. However, the Start Conversation operation also starts the conversation and contacts the bot, whereas the Generate Token operation does neither of these things.

If you intend to start the conversation immediately, use the Start Conversation operation. If you plan to distribute

the token to clients and want them to initiate the conversation, use the [Generate Token](#) operation instead.

Additional resources

- [Key concepts](#)
- [Authentication](#)

Send a message to the bot

7/26/2017 • 4 min to read • [Edit Online](#)

IMPORTANT

This article describes how to send a message to the bot using Direct Line API 1.1. If you are creating a new connection between your client application and bot, use [Direct Line API 3.0](#) instead.

Using the Direct Line 1.1 protocol, clients can exchange messages with bots. These messages are converted to the schema that the bot supports (Bot Framework v1 or Bot Framework v3). A client may send a single message per request.

Send a message

To send a message to the bot, the client must create a [Message](#) object to define the message and then issue a `POST` request to `https://directline.botframework.com/api/conversations/{conversationId}/messages`, specifying the [Message](#) object in the body of the request.

The following snippets provide an example of the Send Message request and response.

Request

```
POST https://directline.botframework.com/api/conversations/abc123/messages
Authorization: Bearer RCurR_XV9ZA.cwA.BKA.iaJrC8xpy8qbOF5xnR2vtCX7CZj0LdjAPGfiCpg4Fv0
[other headers]
```

```
{
  "text": "hello",
  "from": "user1"
}
```

Response

When the message is delivered to the bot, the service responds with an HTTP status code that reflects the bot's status code. If the bot generates an error, an HTTP 500 response ("Internal Server Error") is returned to the client in response to its Send Message request. If the POST is successful, the service returns an HTTP 204 status code. No data is returned in body of the response. The client's message and any messages from the bot can be obtained via [polling](#).

```
HTTP/1.1 204 No Content
[other headers]
```

Total time for the Send Message request/response

The total time to POST a message to a Direct Line conversation is the sum of the following:

- Transit time for the HTTP request to travel from the client to the Direct Line service
- Internal processing time within Direct Line (typically less than 120ms)
- Transit time from the Direct Line service to the bot
- Processing time within the bot

- Transit time for the HTTP response to travel back to the client

Send attachment(s) to the bot

In some situations, a client may need to send attachments to the bot such as images or documents. A client may send attachments to the bot either by [specifying the URL\(s\)](#) of the attachment(s) within the [Message](#) object that it sends using `POST /api/conversations/{conversationId}/messages` or by [uploading attachment\(s\)](#) using `POST /api/conversations/{conversationId}/upload`.

Send attachment(s) by URL

To send one or more attachments as part of the [Message](#) object using

`POST /api/conversations/{conversationId}/messages`, specify the attachment URL(s) within the message's `images` array and/or `attachments` array.

Send attachment(s) by upload

Often, a client may have image(s) or document(s) on a device that it wants to send to the bot, but no URLs corresponding to those files. In this situation, a client can issue a

`POST /api/conversations/{conversationId}/upload` request to send attachments to the bot by upload. The format and contents of the request will depend upon whether the client is [sending a single attachment](#) or [sending multiple attachments](#).

Send a single attachment by upload

To send a single attachment by upload, issue this request:

```
POST https://directline.botframework.com/api/conversations/{conversationId}/upload?userId={userId}
Authorization: Bearer SECRET_OR_TOKEN
Content-Type: TYPE_OF_ATTACHMENT
Content-Disposition: ATTACHMENT_INFO
[other headers]

[file content]
```

In this request URI, replace `{conversationId}` with the ID of the conversation and `{userId}` with the ID of the user that is sending the message. In the request headers, set `Content-Type` to specify the attachment's type and set `Content-Disposition` to specify the attachment's filename.

The following snippets provide an example of the Send (single) Attachment request and response.

Request

```
POST https://directline.botframework.com/api/conversations/abc123/upload?userId=user1
Authorization: Bearer RCurR_XV9ZA.cwA.BKA.iaJrC8xpy8qb0F5xnR2vtCX7CZj0LdjAPGfiCpg4Fv0
Content-Type: image/jpeg
Content-Disposition: name="file"; filename="badjokeeel.jpg"
[other headers]

[JPEG content]
```

Response

If the request is successful, a message is sent to the bot when the upload completes and the service returns an HTTP 204 status code.

```
HTTP/1.1 204 No Content  
[other headers]
```

Send multiple attachments by upload

To send multiple attachments by upload, `POST` a multipart request to the `/api/conversations/{conversationId}/upload` endpoint. Set the `Content-Type` header of the request to `multipart/form-data` and include the `Content-Type` header and `Content-Disposition` header for each part to specify each attachment's type and filename. In the request URI, set the `userId` parameter to the ID of the user that is sending the message.

You may include a `Message` object within the request by adding a part that specifies the `Content-Type` header value `application/vnd.microsoft.bot.message`. This allows the client to customize the message that contains the attachment(s). If the request includes a `Message`, the attachments that are specified by other parts of the payload are added as attachments to that `Message` before it is sent.

The following snippets provide an example of the Send (multiple) Attachments request and response. In this example, the request sends a message that contains some text and a single image attachment. Additional parts could be added to the request to include multiple attachments in this message.

Request

```
POST https://directline.botframework.com/api/conversations/abc123/upload?userId=user1  
Authorization: Bearer RCurR_XV9ZA.cwA.BKA.iaJrC8xpy8qb0F5xnR2vtCX7CZj0LdjAPGfiCpg4Fv0  
Content-Type: multipart/form-data; boundary=----DD4E5147-E865-4652-B662-F223701A8A89  
[other headers]  
  
----DD4E5147-E865-4652-B662-F223701A8A89  
Content-Type: image/jpeg  
Content-Disposition: form-data; name="file"; filename="badjokeeel.jpg"  
[other headers]  
  
[JPEG content]  
  
----DD4E5147-E865-4652-B662-F223701A8A89  
Content-Type: application/vnd.microsoft.bot.message  
[other headers]  
  
{  
  "text": "Hey I just IM'd you\n\nand this is crazy\n\nbut here's my webhook\n\nso POST me maybe",  
  "from": "user1"  
}  
----DD4E5147-E865-4652-B662-F223701A8A89
```

Response

If the request is successful, a message is sent to the bot when the upload completes and the service returns an HTTP 204 status code.

```
HTTP/1.1 204 No Content  
[other headers]
```

Additional resources

- [Key concepts](#)
- [Authentication](#)
- [Start a conversation](#)

- Receive messages from the bot

Receive messages from the bot

7/26/2017 • 1 min to read • [Edit Online](#)

IMPORTANT

This article describes how to receive messages from the bot using Direct Line API 1.1. If you are creating a new connection between your client application and bot, use [Direct Line API 3.0](#) instead.

Using the Direct Line 1.1 protocol, clients must poll an `HTTP GET` interface to receive messages.

Retrieve messages with HTTP GET

To retrieve messages for a specific conversation, issue a `GET` request to the

`api/conversations/{conversationId}/messages` endpoint, optionally specifying the `watermark` parameter to indicate the most recent message seen by the client. An updated `watermark` value will be returned in the JSON response, even if no messages are included.

The following snippets provide an example of the Get Messages request and response. The Get Messages response contains `watermark` as a property of the `MessageSet`. Clients should page through the available messages by advancing the `watermark` value until no messages are returned.

Request

```
GET https://directline.botframework.com/api/conversations/abc123/messages?watermark=0001a-94
Authorization: Bearer RCurR_XV9ZA.cwA.BKA.iaJrC8xpy8qb0F5xnR2vtCX7CZj0LdjAPGfiCpg4Fv0
```

Response

```
HTTP/1.1 200 OK
[other headers]
```

```
{
  "messages": [
    {
      "conversation": "abc123",
      "id": "abc123|0000",
      "text": "hello",
      "from": "user1"
    },
    {
      "conversation": "abc123",
      "id": "abc123|0001",
      "text": "Nice to see you, user1!",
      "from": "bot1"
    }
  ],
  "watermark": "0001a-95"
}
```

Timing considerations

Even though Direct Line is a multi-part protocol with potential timing gaps, the protocol and service is designed to make it easy to build a reliable client. The `watermark` property that is sent in the Get Messages response is reliable. A client will not miss any messages as long as it replays the watermark verbatim.

Clients should choose a polling interval that matches their intended use.

- Service-to-service applications often use a polling interval of 5s or 10s.
- Client-facing applications often use a polling interval of 1s, and issue an additional request ~300ms after every message that the client sends (to rapidly retrieve a bot's response). This 300ms delay should be adjusted based on the bot's speed and transit time.

Additional resources

- [Key concepts](#)
- [Authentication](#)
- [Start a conversation](#)
- [Send a message to the bot](#)

API reference - Direct Line API 1.1

7/26/2017 • 6 min to read • [Edit Online](#)

IMPORTANT

This article contains reference information for Direct Line API 1.1. If you are creating a new connection between your client application and bot, use [Direct Line API 3.0](#) instead.

You can enable your client application to communicate with your bot by using Direct Line API 1.1. Direct Line API 1.1 uses industry-standard REST and JSON over HTTPS.

Base URI

To access Direct Line API 1.1, use this base URI for all API requests:

```
https://directline.botframework.com
```

Headers

In addition to the standard HTTP request headers, a Direct Line API request must include an `Authorization` header that specifies a secret or token to authenticate the client that is issuing the request. You can specify the `Authorization` header using either the "Bearer" scheme or the "BotConnector" scheme.

Bearer scheme:

```
Authorization: Bearer SECRET_OR_TOKEN
```

BotConnector scheme:

```
Authorization: BotConnector SECRET_OR_TOKEN
```

For details about how to obtain a secret or token that your client can use to authenticate its Direct Line API requests, see [Authentication](#).

HTTP status codes

The [HTTP status code](#) that is returned with each response indicates the outcome of the corresponding request.

HTTP STATUS CODE	MEANING
200	The request succeeded.
204	The request succeeded but no content was returned.
400	The request was malformed or otherwise incorrect.
401	The client is not authorized to make the request. Often this status code occurs because the <code>Authorization</code> header is missing or malformed.

HTTP STATUS CODE	MEANING
403	The client is not allowed to perform the requested operation. Often this status code occurs because the <code>Authorization</code> header specifies an invalid token or secret.
404	The requested resource was not found. Typically this status code indicates an invalid request URI.
500	Either an internal server error occurred within the Direct Line service or a failure occurred within the bot. If you receive a 500 error when POSTing a message to a bot, it is possible that the error was triggered by a failure in the bot. This is a common error code.

Token operations

Use these operations to create or refresh a token that a client can use to access a single conversation.

OPERATION	DESCRIPTION
Generate Token	Generate a token for a new conversation.
Refresh Token	Refresh a token.

Generate Token

Generates a token that is valid for one conversation.

POST /api/tokens/conversation	
Request body	n/a
Returns	A string that represents the token

Refresh Token

Refreshes the token.

GET /api/tokens/{conversationId}/renew	
Request body	n/a
Returns	A string that represents the new token

Conversation operations

Use these operations to open a conversation with your bot and exchange messages between client and bot.

OPERATION	DESCRIPTION
Start Conversation	Opens a new conversation with the bot.
Get Messages	Retrieves messages from the bot.
Send a Message	Sends a message to the bot.
Upload and Send File(s)	Uploads and sends file(s) as attachment(s).

Start Conversation

Opens a new conversation with the bot.

POST /api/conversations	
Request body	n/a
Returns	A Conversation object

Get Messages

Retrieves messages from the bot for the specified conversation. Set the `watermark` parameter in the request URI to indicate the most recent message seen by the client.

GET /api/conversations/{conversationId}/messages?watermark={watermark_value}	
Request body	n/a
Returns	A MessageSet object. The response contains <code>watermark</code> as a property of the MessageSet object. Clients should page through the available messages by advancing the <code>watermark</code> value until no messages are returned.

Send a Message

Sends a message to the bot.

POST /api/conversations/{conversationId}/messages	
Request body	A Message object
Returns	No data is returned in body of the response. The service responds with an HTTP 204 status code if the message was sent successfully. The client may obtain its sent message (along with any messages that the bot has sent to the client) by using the Get Messages operation.

Upload and Send File(s)

Uploads and sends file(s) as attachment(s). Set the `userId` parameter in the request URI to specify the ID of the user that is sending the attachments.

```
POST /api/conversations/{conversationId}/upload?userId={userId}
```

Request body	For a single attachment, populate the request body with the file contents. For multiple attachments, create a multipart request body that contains one part for each attachment, and also (optionally) one part for the Message object that should serve as the container for the specified attachment(s). For more information, see Send a message to the bot .
Returns	No data is returned in body of the response. The service responds with an HTTP 204 status code if the message was sent successfully. The client may obtain its sent message (along with any messages that the bot has sent to the client) by using the Get Messages operation.

Schema

Direct Line 1.1 schema is a simplified copy of the Bot Framework v1 schema that includes the following objects.

Message object

Defines a message that a client sends to a bot or receives from a bot.

PROPERTY	TYPE	DESCRIPTION
id	string	ID that uniquely identifies the message (assigned by Direct Line).
conversationId	string	ID that identifies the conversation.
created	string	Date and time that the message was created, expressed in ISO-8601 format.
from	string	ID that identifies the user that is the sender of the message. When creating a message, clients should set this property to a stable user ID. Although Direct Line will assign a user ID if none is supplied, this typically results in unexpected behavior.
text	string	Text of the message that is sent from user to bot or bot to user.

PROPERTY	TYPE	DESCRIPTION
channelData	object	An object that contains channel-specific content. Some channels provide features that require additional information that cannot be represented using the attachment schema. For those cases, set this property to the channel-specific content as defined in the channel's documentation. This data is sent unmodified between client and bot. This property must either be set to a complex object or left empty. Do not set it to a string, number, or other simple type.
images	string[]	Array of strings that contains the URL(s) for the image(s) that the message contains. In some cases, strings in this array may be relative URLs. If any string in this array does not begin with either "http" or "https", prepend https://directline.botframework.com to the string to form the complete URL.
attachments	Attachment[]	Array of Attachment objects that represent the non-image attachments that the message contains. Each object in the array contains a <code>url</code> property and a <code>contentType</code> property. In messages that a client receives from a bot, the <code>url</code> property may sometimes specify a relative URL. For any <code>url</code> property value that does not begin with either "http" or "https", prepend https://directline.botframework.com to the string to form the complete URL.

The following example shows a Message object that contains all possible properties. In most cases when creating a message, the client only needs to supply the `from` property and at least one content property (e.g., `text`, `images`, `attachments`, or `channelData`).

```
{
  "id": "CuvLPID4kDb|00000000000000000004",
  "conversationId": "CuvLPID4kDb",
  "created": "2016-10-28T21:19:51.0357965Z",
  "from": "examplebot",
  "text": "Hello!",
  "channelData": {
    "examplefield": "abc123"
  },
  "images": [
    "/attachments/CuvLPID4kDb/0.jpg?..."
  ],
  "attachments": [
    {
      "url": "https://example.com/example.docx",
      "contentType": "application/vnd.openxmlformats-officedocument.wordprocessingml.document"
    },
    {
      "url": "https://example.com/example.doc",
      "contentType": "application/msword"
    }
  ]
}
```

MessageSet object

Defines a set of messages.

PROPERTY	TYPE	DESCRIPTION
messages	Message[]	Array of Message objects.
watermark	string	Maximum watermark of messages within the set. A client may use the watermark value to indicate the most recent message it has seen when retrieving messages from the bot .

Attachment object

Defines a non-image attachment.

PROPERTY	TYPE	DESCRIPTION
contentType	string	The media type of the content in the attachment.
url	string	URL for the content of the attachment.

Conversation object

Defines a Direct Line conversation.

PROPERTY	TYPE	DESCRIPTION

PROPERTY	TYPE	DESCRIPTION
conversationId	string	ID that uniquely identifies the conversation for which the specified token is valid.
token	string	Token that is valid for the specified conversation.
expires_in	number	Number of seconds until the token expires.

Error object

Defines an error.

PROPERTY	TYPE	DESCRIPTION
code	string	Error code. One of these values: MissingProperty , MalformedData , NotFound , ServiceError , Internal , InvalidRange , NotSupported , NotAllowed , BadCertificate .
message	string	A description of the error.
statusCode	number	Status code.

ErrorMessage object

A standardized message error payload.

PROPERTY	TYPE	DESCRIPTION
error	Error	An Error object that contains information about the error.

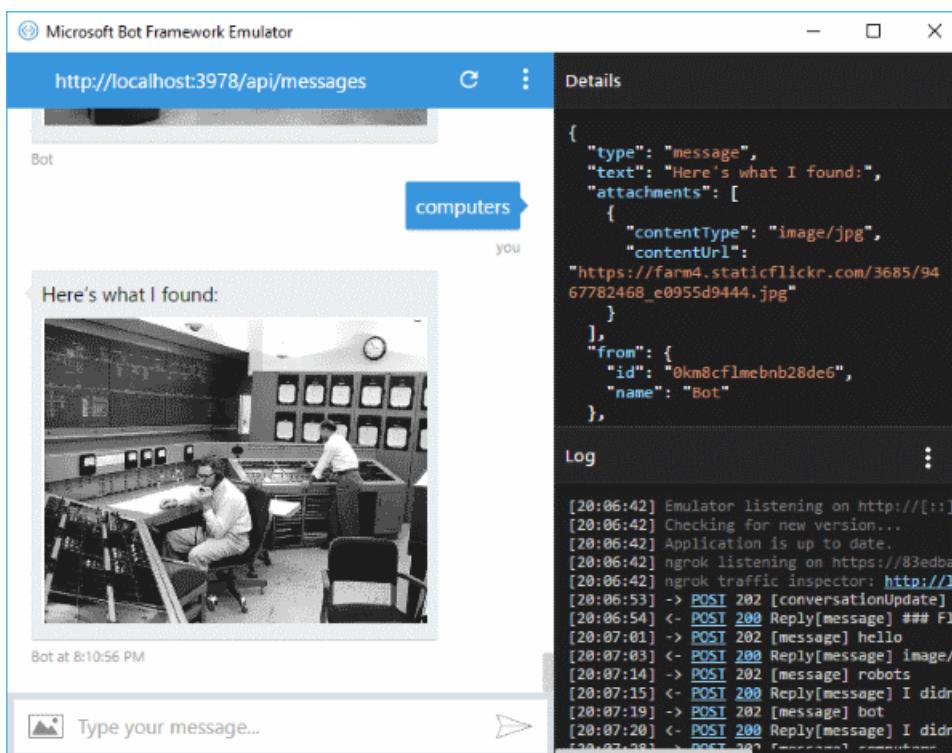
Debug bots with the Bot Framework Emulator

9/13/2017 • 4 min to read • [Edit Online](#)

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 that your bot 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.

TIP

Before you deploy your bot to the cloud, run it locally and test it using the emulator. You can test your bot using the emulator even if you have not yet registered it with the Bot Framework or configured it to run on any channels.



Prerequisites

Download the Bot Framework Emulator

Download packages for Mac, Windows, and Linux are available via the [GitHub releases page](#). The latest Windows installer is available via the [emulator download page](#) (download starts immediately).

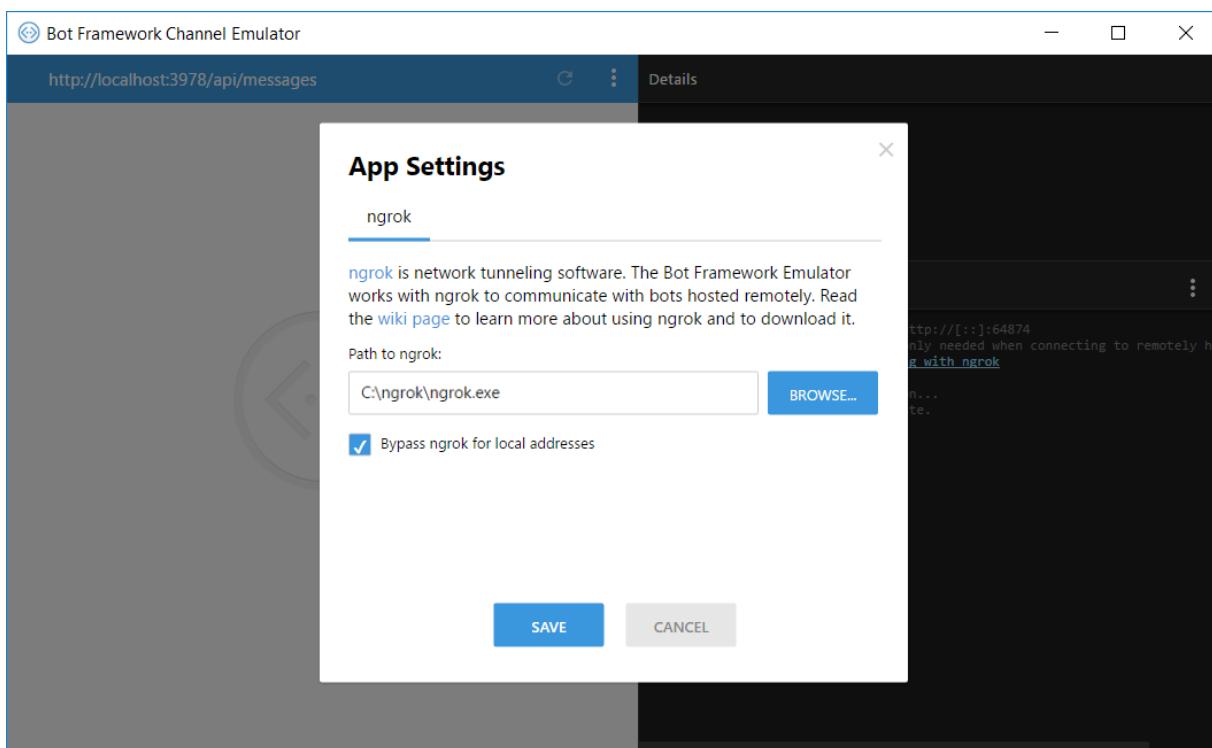
Install and configure ngrok

If you are running the Bot Framework Emulator behind a firewall or other network boundary and want to connect to a bot that is hosted remotely, you must install and configure **ngrok** tunneling software.

Computers running behind firewalls and home routers are not able to accept ad-hoc incoming requests from the outside world, but tunneling software provides a way around this by creating a bridge from outside the firewall to your local machine. The Bot Framework Emulator integrates tightly with **ngrok** tunnelling software (developed by [inconschreivable](#)), and can launch it automatically when it is needed.

To install **ngrok** and configure the emulator to use it, complete these steps:

1. Download the [ngrok](#) executable to your local machine.
2. Open the emulator's App Settings dialog, enter the path to ngrok, select whether or not to bypass ngrok for local addresses, and click **Save**.



When the emulator connects to a bot that is hosted remotely, it displays messages in its log that indicate that ngrok has automatically been launched. If you've followed these steps but the emulator's log indicates that it is not able to launch ngrok, ensure that you have installed ngrok version [2.1.18](#) or later. (Earlier versions have been known to be incompatible.) To check ngrok's version, run this command from the command line:

```
ngrok -v
```

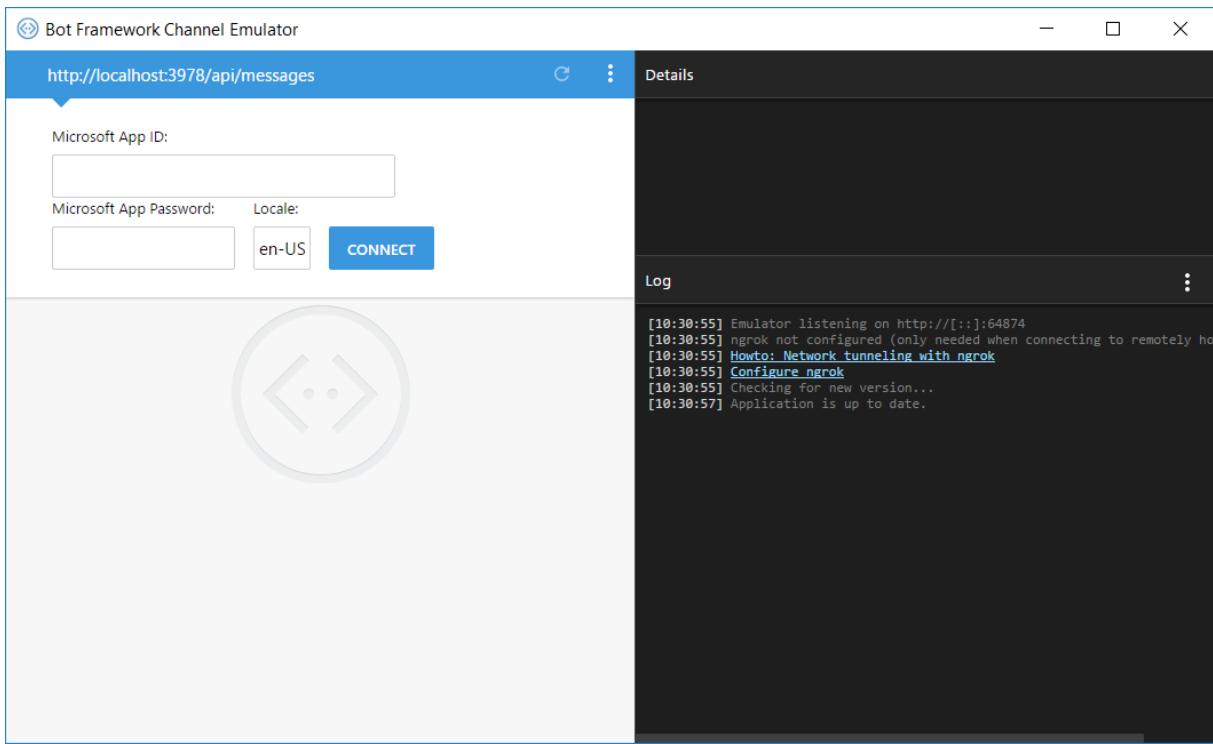
Connect to a bot that is running on localhost

Launch the Bot Framework Emulator and enter your bot's endpoint into the emulator's address bar.

TIP

If your bot was built using the Bot Builder SDK, the default endpoint for local debugging is <http://localhost:3978/api/messages>. This is where the bot will be listening for messages when hosted locally. Next, if your bot is running with Microsoft Account (MSA) credentials, enter those values into the **Microsoft App ID** and **Microsoft App Password** fields. For localhost debugging, you will not typically need to populate these fields, although doing so is supported if your bot requires it.

Finally, click **Connect** to connect the emulator to your bot. After the emulator has connected to your bot, you can send and receive messages using the embedded chat control.



Connect to a bot that is hosted remotely

Launch the Bot Framework Emulator and enter your bot's endpoint into the emulator's address bar.

Next, populate the **Microsoft App ID** and **Microsoft App Password** fields with your bot's Microsoft Account (MSA) credentials. If you have registered your bot with the Bot Framework, you can retrieve its endpoint and MSA App ID from the bot's registration page in the [Bot Framework Portal](#). If you have registered your bot with the Bot Framework but do not know its MSA Password, you can [generate a new password](#).

Ensure that [ngrok](#) is installed and that the emulator's App Settings specify the path to the **ngrok** executable. **ngrok** enables the emulator to communicate with your remotely-hosted bot.

Finally, click **Connect** to connect the emulator to your bot. After the emulator has connected to your bot, you can send and receive messages using the embedded chat control.

TIP

When you create a bot using the Azure Bot Service, it is automatically registered with the Bot Framework. You can retrieve its endpoint and MSA App ID from the bot's registration page in the [Bot Framework Portal](#). If you do not know your bot's MSA Password, you can [generate a new password](#).

Enable speech recognition

The Bot Framework Emulator supports speech recognition via the [Cognitive Services Speech API](#). This allows you to exercise your speech-enabled bot, or Cortana skill, via speech in the emulator during development. The Bot Framework Emulator provides speech recognition free of charge for up to three hours per bot per day.

Send system activities

You can use the Bot Framework Emulator to emulate specific activities within the context of a conversation, by selecting from the available options under **Conversation > Send System Activity** in the emulator's settings menu:

- conversationUpdate (user added)
- conversationUpdate (user removed)
- contactRelationUpdate (bot added)
- contactRelationUpdate (bot removed)
- typing
- ping
- deleteUserData

Emulate payment processing

You can use the Bot Framework Emulator to emulate payment processing. In emulation mode, no real payment will be processed. Instead, the process payment logic simply returns a successful payment record. For payment processing, the emulator remembers your payment methods, shipping addresses, and it supports form field validation.

Additional resources

To build a simple bot that you can test using the emulator, see:

- [Create a bot with the Bot Builder SDK for Node.js](#)
- [Create a bot with the Bot Builder SDK for .NET](#)

The Bot Framework Emulator is open source. To contribute or file bugs and suggestions, see:

- [How to Contribute](#)
- [Submitting Bugs and Suggestions](#)

For information about troubleshooting issues with your bot, see:

- [Bot Framework troubleshooting guide](#)
- [Troubleshooting Bot Framework authentication](#)

To preview if a feature is supported in a targeted channel, see [Channel Inspector](#)

Debug a bot with Visual Studio Code

8/7/2017 • 2 min to read • [Edit Online](#)

While the [Bot Framework Emulator](#) provides useful features for debugging your bots, you can also use the debugger included in [Visual Studio Code](#) to debug your bot's source code.

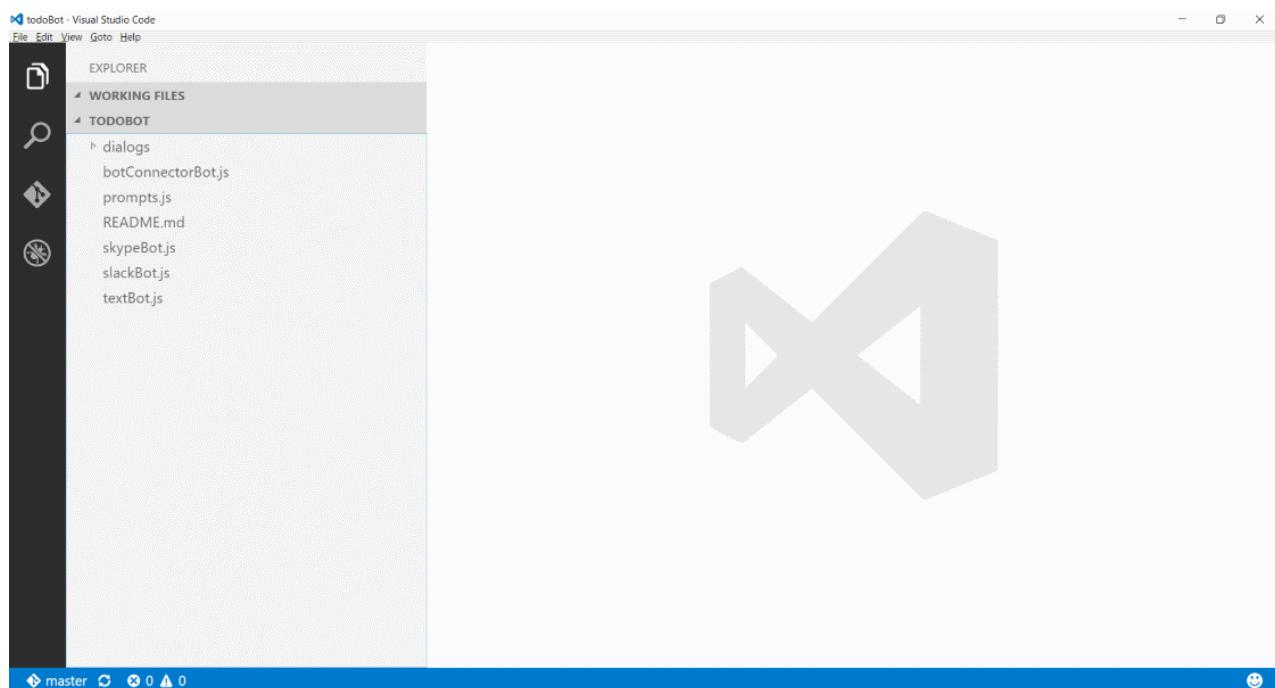
NOTE

Visual Studio Code has built-in debugging support for the Node.js runtime. This article guides you through debugging a bot built with Node.js. For debugging a bot built with C#, you will need the C# extension. You can find more information in the [VS Code documentation](#).

The Bot Builder SDK for Node.js provides the [ConsoleConnector](#) class to help you debug your bot by running it in a console window. This guide will walk you through that process.

Launch Visual Studio Code

For purposes of this walkthrough we'll use the [ConsoleConnector](#) example from the Bot Builder SDK for Node.js. After you install VS Code on your machine, open your bot's project in VS Code by using **Open Folder** in the File menu.



Launch your bot

The ConsoleConnector example illustrates running a bot on multiple platforms which is the key to debugging your bot locally. To debug locally you need a version of your bot that can run from a console window using the [ConsoleConnector](#) class. You can run the ConsoleConnector example locally by launching the **app.js** class. To debug this class using VS Code, launch Node.js with the `--debug-brk` flag, which causes it to immediately break. From a console window, type `node --debug-brk textBot.js`.

A screenshot of Visual Studio Code showing a terminal window. The terminal output is as follows:

```
C:\Source\BotBuilder\Node\examples\todoBot>node --debug-brk textBot.js
Debugger listening on port 5858
```

Configure Visual Studio Code

Before you can debug your paused bot, you'll need to configure the VS Code Node.js debugger. VS Code knows your project is using Node.js, but there are multiple configurations for launching Node.js, so you'll go through a one-time setup for each project (that is, each folder).

To set up the debugger:

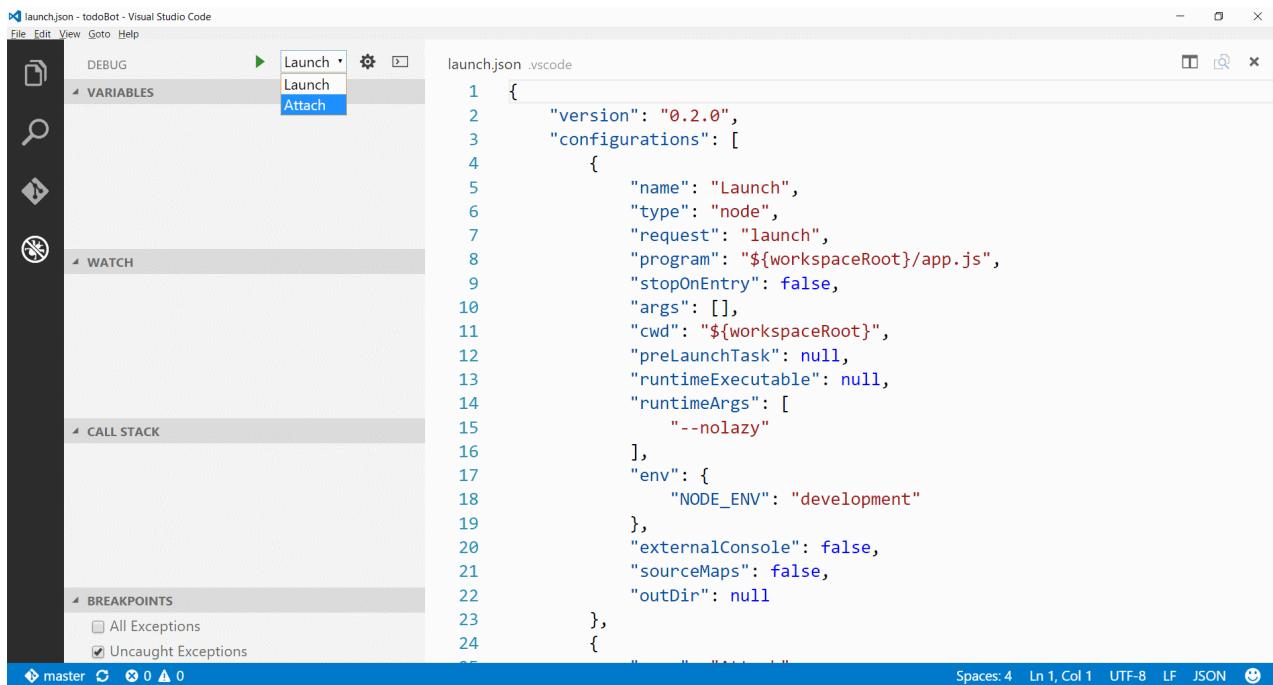
1. Select the debug tab on the lower left and click the run button.
2. VS Code prompts you to pick your debug environment. Select **Node.js**. You don't need to change the default settings.
3. You should see a **.vscode** folder added to your project. If you don't want this folder to be checked into your Git repository, add a `/**/.vscode` entry to your `.gitignore` file.

A screenshot of Visual Studio Code showing the DEBUG tab selected. A dropdown menu is open, showing two options: "Node.js" and "Extension Development".

Attach the debugger

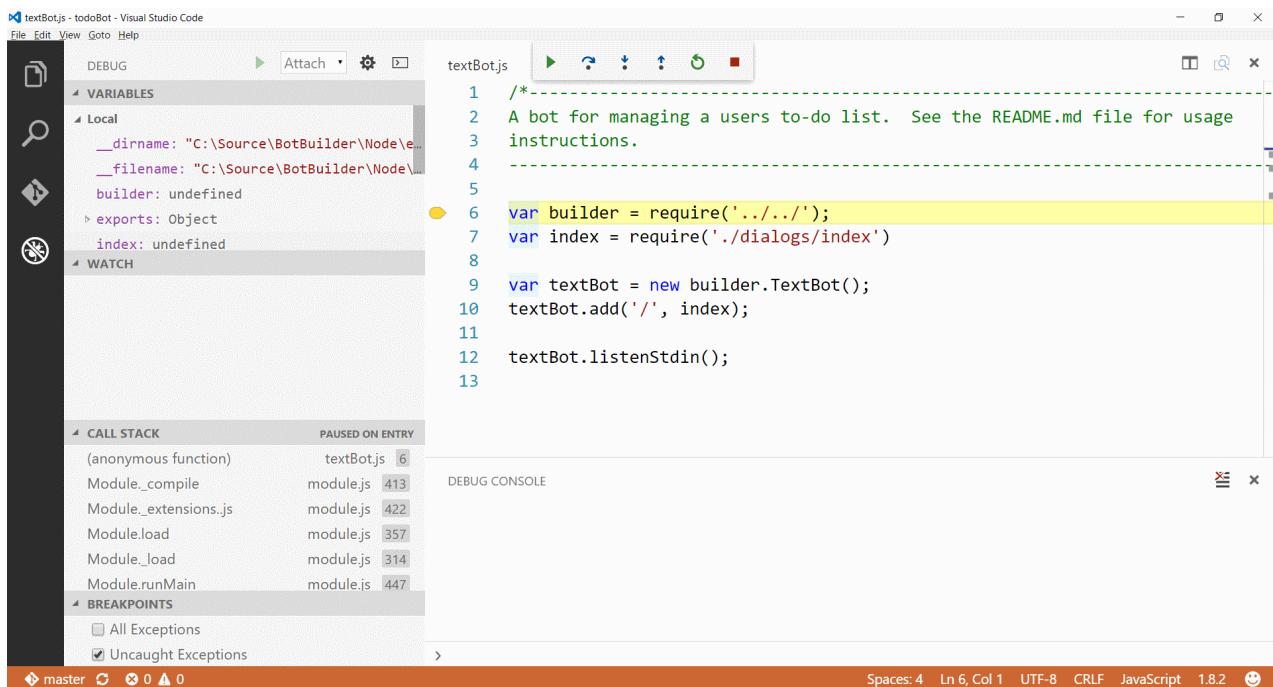
Configuring the debugger added two debug modes: **Launch** and **Attach**. Since our bot is paused in a console

window, select the **Attach** mode and click the run button again.



Debug your bot

When VS Code attaches to your bot, it pauses on the first line of code. Now you're ready to set breakpoints and debug your bot! You can communicate with your bot from the console window. Try switching back to the console window that your bot is running in and say "hello".



Additional resources

- [Bot Framework Emulator](#)
- [Debugging in Visual Studio Code](#)
- [Visual Studio Code](#)

Debug an Azure Bot Service bot

10/12/2017 • 8 min to read • [Edit Online](#)

Azure Bot Service bots are built as Azure App Service web apps, or as code running in a Consumption plan upon the Azure Functions serverless architecture. This article describes how to debug your bot after you have set up a publishing process. By downloading a zip file that contains your bot source, you can develop and debug using an integrated development environment (IDE) such as Visual Studio or Visual Studio Code. From your computer, you can update your bot on Azure Bot Service by publishing from Visual Studio, or by publishing with every check-in to your source control service through continuous deployment.

Publish any bot source using continuous deployment

You can publish bot source to Azure using continuous deployment. To set up continuous deployment, [follow these steps](#) before proceeding.

Debug a Node.js bot

Follow steps in this section to debug a bot written in Node.js.

Prerequisites

Before you can debug your Node.js bot, you must complete these tasks.

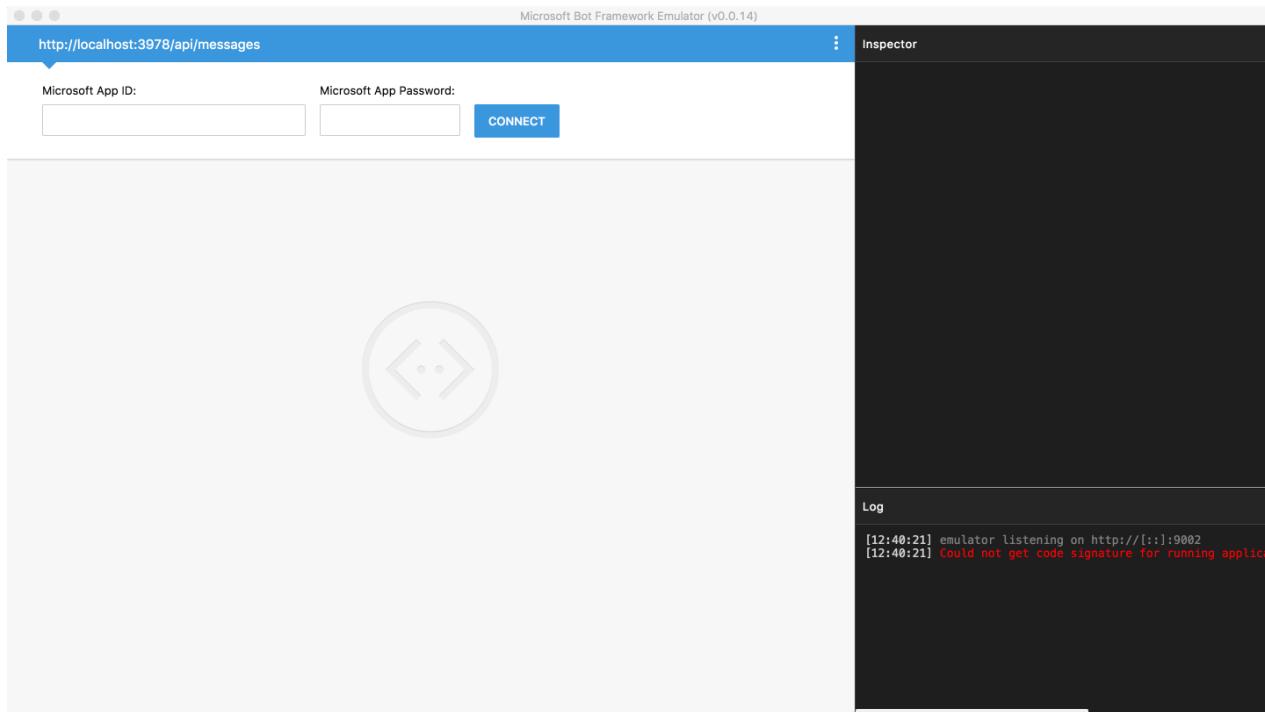
- Download the source code for your bot (from Azure), as described in [Set up continuous deployment](#).
- Download and install the [Bot Framework Emulator](#).
- Download and install a code editor such as [Visual Studio Code](#).

Debug a Node.js bot using the Bot Framework Emulator

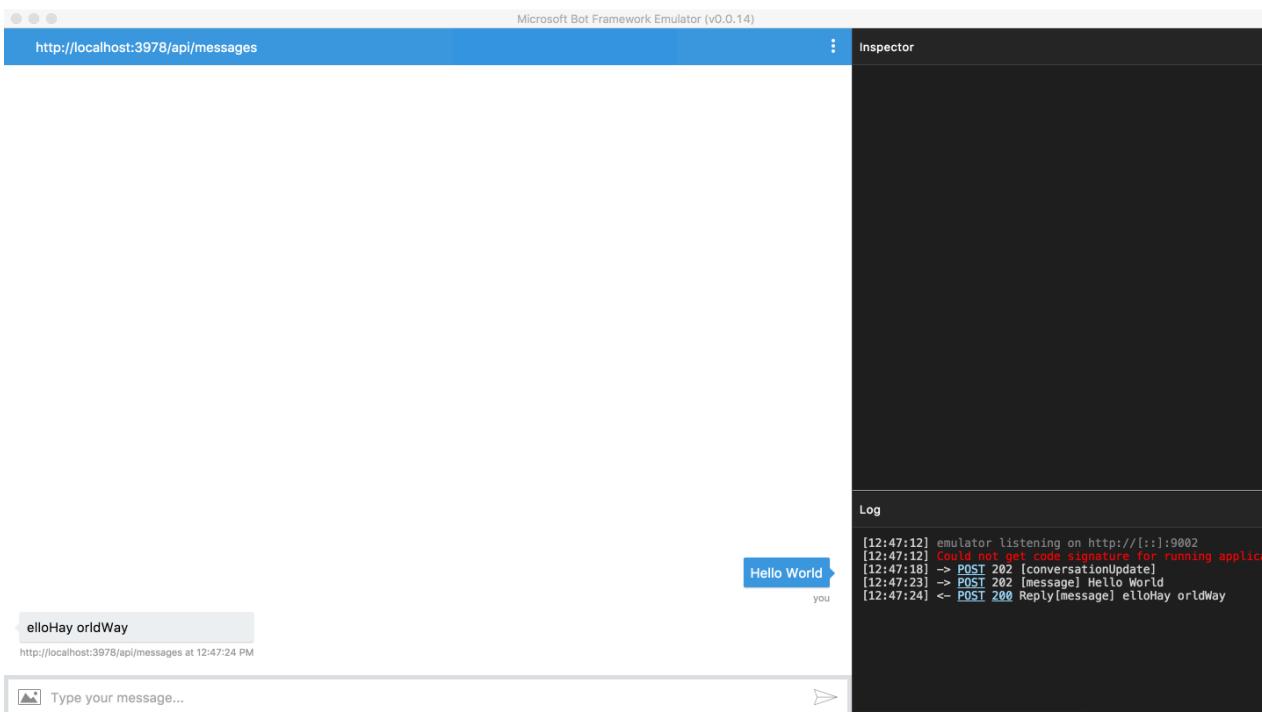
The simplest way to debug your bot locally is to start the bot in Node and then connect to it from Bot Framework Emulator. First, you must set the `NODE_ENV` environment variable. This screenshot shows how to set the `NODE_ENV` environment variable and start the bot.

```
jameslew at Jamess-MacBook-Pro in ~
$ cd repos/piglatin-node-function/EmptyBot
jameslew at Jamess-MacBook-Pro in ~/repos/piglatin-node-function/EmptyBot on master
$ export NODE_ENV="development"
jameslew at Jamess-MacBook-Pro in ~/repos/piglatin-node-function/EmptyBot on master
$ node index.js
test bot endpoint at http://localhost:3978/api/messages
```

At this point, the bot is running locally. Copy the bot's endpoint from the terminal window (in this example, `http://localhost:3978/api/messages`), start the Bot Framework Emulator, and paste the endpoint into the address bar of the emulator. Since you do not need security for local debugging, you can leave the **Microsoft App ID** and **Microsoft App Password** fields blank. Click **Connect** to establish a connection to your bot using the specified endpoint.



After you have connected the emulator to your bot, send a message to your bot by typing some text into the textbox that is located at the bottom of the emulator window (i.e., where **Type your message...** appears in the lower-left corner). By using the **Log** and **Inspector** panels on the right side of the emulator window, you can view the requests and responses as messages are exchanged between the emulator and the bot.



Additionally, you can view log details from the Node runtime in the terminal window.

A screenshot of a terminal window titled "1. node". It shows a command-line session where a user named "jameslew" runs "node index.js" in a directory "EmptyBot" under "piglatin-node-function". The logs output the bot's logic for handling messages, including waterfall steps for sending and receiving messages, and log entries for "ChatConnector" receiving messages and "session.beginDialog()".

```
jameslew at Jamess-MacBook-Pro in ~
$ cd repos/piglatin-node-function/EmptyBot
jameslew at Jamess-MacBook-Pro in ~/repos/piglatin-node-function/EmptyBot on master*
$ export NODE_ENV="development"
jameslew at Jamess-MacBook-Pro in ~/repos/piglatin-node-function/EmptyBot on master*
$ node index.js
test bot endpoint at http://localhost:3978/api/messages
WARN: ChatConnector: receive - emulator running without security enabled.
ChatConnector: message received.
WARN: ChatConnector: receive - emulator running without security enabled.
ChatConnector: message received.
session.beginDialog()
/ - waterfall() step 1 of 1
/ - session.send()
/ - session.sendBatch() sending 1 messages
WARN: ChatConnector: receive - emulator running without security enabled.
ChatConnector: message received.
/ - waterfall() step 1 of 1
/ - session.send()
/ - session.sendBatch() sending 1 messages
[]
```

Debug a Node.js bot using breakpoints in Visual Studio Code

If you need more than logs and request / response traces to debug your bot, you can use a local IDE such as Visual Studio Code to debug your code using breakpoints. Using VS Code to debug your bot can be beneficial because you can make changes locally within the editor while you are debugging, and when you push changes to the remote repository, those changes will automatically be applied to your bot in the cloud (since you have enabled continuous deployment).

First, launch VS Code and open the local folder where the source code for your bot is located.

```
index.js - EmptyBot
1 "use strict";
2 var builder = require("botbuilder");
3 var botbuilder_azure = require("botbuilder-azure");
4 var unicodelib = require("unicode-properties");
5
6 var useEmulator = (process.env.NODE_ENV == 'development');
7
8 var connector = useEmulator ? new builder.ChatConnector() : new botbuilder_azure.BotServiceConnector({
9   appId: process.env['MicrosoftAppId'],
10  appPassword: process.env['MicrosoftAppPassword'],
11  stateEndpoint: process.env['BotStateEndpoint'],
12  openIdMetadata: process.env['BotOpenIdMetadata']
13 });
14
15 var bot = new builder.UniversalBot(connector);
16
17 bot.dialog('/', function (session) {
18
19   var replyMessage = new builder.Message(session);
20   replyMessage.textFormat("plain");
21
22   if (session.message.text === "MessageTypesTest") {
23     var mtReply = session.send(messageTypesTest(session));
24     return;
25   }
26   else if (session.message.text === "DataTypesTest") {
27     //Activity dtResult = await dataTypeTest(message, connector);
28     //var dtReply = session.send(replyMessage);
29     //return;
30   }
31   //else if (session.message.text === "CardTypesTest") {
32   //  var ctResult = await cardTypesTest(message, connector);
33   //  var reply = session.send(replyMessage);
34   //  return;
35   //}
36   else if (session.message.text === "OneOffTests") {
37     oneOffTests_facebook_quick_replies(session);
38     return;
39   }
40   else if (session.message.text === "FBHeroCard") {
41     oneOffTests_HeroCard_WithShare(session);
42   }
43 });
44
45 module.exports = bot;
```

Ln 19, Col 4 Spaces: 4 UTF-8 LF JavaScript

Switch to the debugging view, and click **run**. If you are prompted to select a runtime engine to run your code, select Node.js.

Code File Edit View Go Window Help

index.js - EmptyBot

Select Environment

Node.js
VS Code Extension Development
Node.js v6.3+ (Experimental)

index.js - EmptyBot

```
index.js - EmptyBot
1 "use strict";
2 var builder = Node.js
3 var botbuilder = VS Code Extension Development
4 var unicodelib = Node.js v6.3+ (Experimental)
5
6 var useEmulator = (process.env.NODE_ENV == 'development');
7
8 var connector = useEmulator ? new builder.ChatConnector() : new botbuilder_azure.BotServiceConnector({
9   appId: process.env['MicrosoftAppId'],
10  appPassword: process.env['MicrosoftAppPassword'],
11  stateEndpoint: process.env['BotStateEndpoint'],
12  openIdMetadata: process.env['BotOpenIdMetadata']
13 });
14
15 var bot = new builder.UniversalBot(connector);
16
17 bot.dialog('/', function (session) {
18
19   var replyMessage = new builder.Message(session);
20   replyMessage.textFormat("plain");
21
22   if (session.message.text === "MessageTypesTest") {
23     var mtReply = session.send(messageTypesTest(session));
24     return;
25   }
26   else if (session.message.text === "DataTypesTest") {
27     //Activity dtResult = await dataTypeTest(message, connector);
28     //var dtReply = session.send(replyMessage);
29     //return;
30   }
31   //else if (session.message.text === "CardTypesTest") {
32   //  var ctResult = await cardTypesTest(message, connector);
33   //  var reply = session.send(replyMessage);
34   //  return;
35   //}
36   else if (session.message.text === "OneOffTests") {
37     oneOffTests_facebook_quick_replies(session);
38     return;
39   }
40   else if (session.message.text === "FBHeroCard") {
41     oneOffTests_HeroCard_WithShare(session);
42   }
43 });
44
45 module.exports = bot;
```

Ln 19, Col 4 Spaces: 4 UTF-8 LF JavaScript

Next, depending on whether you have synced the repository or modified files, you may be prompted to configure the **launch.json** file. If you are prompted, add the `env` configuration setting to the **launch.json** file (to tell the template that you are going to work with the emulator).

```
"env": {  
  "NODE\_ENV": "development"  
}
```

The screenshot shows the VS Code interface with the following details:

- File:** launch.json - EmptyBot
- Code Editor:** Content of launch.json:

```
1 // Use IntelliSense to learn about possible Node.js debug attributes.
2 // Hover to view descriptions of existing attributes.
3 // For more information, visit: https://go.microsoft.com/fwlink/?linkid=830387
4 "version": "0.2.0",
5 "configurations": [
6     {
7         "type": "node",
8         "request": "launch",
9         "name": "Launch Program",
10        "program": "${workspaceRoot}/index.js",
11        "cwd": "${workspaceRoot}",
12        "env": {
13            "NODE_ENV": "development"
14        }
15    }
16 ]
17 ]
18 ]
```
- Debug Bar:** Shows the DEBUG tab is selected, Launch Program is chosen, and the launch.json file is open.
- Sidebar:** Variables, Watch, Call Stack, Breakpoints (with Uncaught Exceptions checked).
- Bottom Status Bar:** master, Ln 18, Col 2, Spaces: 4, UTF-8, LF, JSON.

Save your changes to the **launch.json** file and click **run** again. Your bot should now be running in the VS Code environment with Node. You can open the debug console to see logging output and set breakpoints as needed.

The screenshot shows the VS Code interface with the following details:

- File:** index.js - EmptyBot
- Code Editor:** Content of index.js:

```
1 "use strict";
2 var builder = require("botbuilder");
3 var botbuilder_azure = require("botbuilder-azure");
4 var unicodelib = require("unicode-properties");
5
6 var useEmulator = (process.env.NODE_ENV === 'development');
7
8 var connector = useEmulator ? new builder.ChatConnector() : new botbuilder_azure.BotServiceConnector({
9     appId: process.env['MicrosoftAppId'],
10    appPassword: process.env['MicrosoftAppPassword'],
11    stateEndpoint: process.env['BotStateEndpoint'],
12    openIdMetadata: process.env['BotOpenIdMetadata']
13 });
14
15 var bot = new builder.UniversalBot(connector);
16
17 bot.dialog('/', function (session) {
18
19     var replyMessage = new builder.Message(session);
20     replyMessage.textFormat("plain");
21
22     if (session.message.text==="MessageTypeTest") {
23         var mReply = session.send(messageTypesTest(session));
24         return;
25     }
26     else if (session.message.text==="DataTypesTest") {
27         //Activity dtResult = await dataTypeTest(message, connector);
28         //var dtReply = session.send(replyMessage);
29     }
30 })
31
32 module.exports = bot;
```
- Debug Bar:** Shows the DEBUG tab is selected, Launch Program is chosen, and the index.js file is open.
- Sidebar:** Variables, Watch, Call Stack, Breakpoints (with All Exceptions and Uncaught Exceptions checked).
- Bottom Status Bar:** master, Ln 51, Col 1, Spaces: 4, UTF-8, LF, JavaScript.
- DEBUG CONSOLE:** Output:

```
node --debug-brk=5314 --nolazy index.js
Debugger listening on port 5314
test bot endpoint at http://localhost:3978/api/messages
```

At this point, the bot is running locally. Copy the bot's endpoint from the debug console in VS Code, start the Bot Framework Emulator, and paste the endpoint into the address bar of the emulator. Since you do not need security for local debugging, you can leave the **Microsoft App ID** and **Microsoft App Password** fields blank. Click **Connect** to establish a connection to your bot using the specified endpoint.

After you have connected the emulator to your bot, send a message to your bot by typing some text into the textbox that is located at the bottom of the emulator window (i.e., where **Type your message...** appears in the lower-left corner). As messages are exchanged between the emulator and the bot, you should hit the breakpoints that you set in VS Code.

The screenshot shows the Microsoft Visual Studio Code interface with the following details:

- Left Sidebar:** Includes sections for **VARIABLES**, **WATCH**, **CALL STACK**, and **BREAKPOINTS**. The **WATCH** section is currently selected.
- Top Bar:** Shows tabs for **DEBUG**, **Launch Program**, and **index.js**. The **index.js** tab is active.
- Code Editor:** Displays the **index.js** file content. A yellow circle highlights the current line of code: `var replyMessage = new builder.Message(session);`. The code implements a bot that handles different message types based on user input.
- Bottom Panel:** Contains the **DEBUG CONSOLE** which outputs log messages from the bot's execution.

```
index.js  x  index.js - EmptyBot

1 "use strict";
2 var builder = require("botbuilder");
3 var botbuilder_azure = require("botbuilder-azure");
4 var unicodelib = require("unicode-properties");
5
6 var useEmulator = (process.env.NODE_ENV == 'development');
7
8 var connector = useEmulator ? new builder.ChatConnector() : new botbuilder_azure.BotServiceConnector({
9   appId: process.env['MicrosoftAppId'],
10  appPassword: process.env['MicrosoftAppPassword'],
11  stateEndpoint: process.env['BotStateEndpoint'],
12  openIdMetadata: process.env['BotOpenIdMetadata']
13 });
14
15 var bot = new builder.UniversalBot(connector);
16
17 bot.dialog('/', function (session) {
18
19   var replyMessage = new builder.Message(session);
20   replyMessage.textFormat("plain");
21
22   if (session.message.text=="MessageTypesTest") {
23     var mtReply = session.send(messageTypesTest(session));
24     return;
25   }
26   else if (session.message.text=="DataTypesTest") {
27     //Activity dtResult = await dataTypesTest(message, connector);
28     //var dtReply = session.send(replyMessage);
29
30   }
31
32   session.endDialog(`Sorry, I don't understand what you mean by ${session.message.text}.`);
33
34 });
35
36 module.exports = bot;
37
38
39 // - waterfallAction step 1 of 1
```

DEBUG CONSOLE

```
---> [1] Starting Emulator on port 3978...
test bot endpoint at http://localhost:3978/api/messages
WARN: ChatConnector: receive - emulator running without security enabled.
ChatConnector: message received.
WARN: ChatConnector: receive - emulator running without security enabled.
ChatConnector: message received.
ChatConnector: message received.
session.beginDialog()
/ - waterfallAction step 1 of 1
```

Debug a Consumption plan C# script bot

The Consumption plan serverless C# environment in Azure Bot Service has more in common with Node.js than a typical C# application because it requires a runtime host, much like the Node engine. In Azure, the runtime is part of the hosting environment in the cloud, but you must replicate that environment locally on your desktop.

Prerequisites

Before you can debug your Consumption plan C# bot, you must complete these tasks.

- Download the source code for your bot (from Azure), as described in [Set up continuous deployment](#).
 - Download and install the [Bot Framework Emulator](#).
 - Install the [Azure Functions CLI](#).
 - Install the [DotNet CLI](#).

If you want to be able to debug your code by using breakpoints in Visual Studio 2017, you must also complete these tasks.

- Download and install [Visual Studio 2017](#) (Community Edition or above).
 - Download and install the [Command Task Runner Visual Studio Extension](#).

NOTE

Visual Studio Code is not currently supported.

Debug a Consumption plan C# script bot using the Bot Framework Emulator

The simplest way to debug your bot locally is to start the bot and then connect to it from Bot Framework Emulator. First, open a command prompt and navigate to the folder where the **project.json** file is located in your repository. Then, run the command `dotnet restore` to restore the various packages that are referenced in your bot.

NOTE

Visual Studio 2017 changes how Visual Studio handles dependencies. While Visual Studio 2015 uses **project.json** to handle dependencies, Visual Studio 2017 uses a **.csproj** model when loading in Visual Studio. If you are using Visual Studio 2017, download this [.csproj](#) file to the **/messages** folder in your repository before you run the `dotnet restore` command.

```
Command Prompt
C:\repos\piglatinbot-dotnetfunction\EmptyBot>dotnet restore
log : Restoring packages for C:\repos\piglatinbot-dotnetfunction\EmptyBot\project.json...
log : Lock file has not changed. Skipping lock file write. Path: C:\repos\piglatinbot-dotnetfunction\EmptyBot\project.lock.json
on
log : C:\repos\piglatinbot-dotnetfunction\EmptyBot\project.json
log : Restore completed in 196ms.

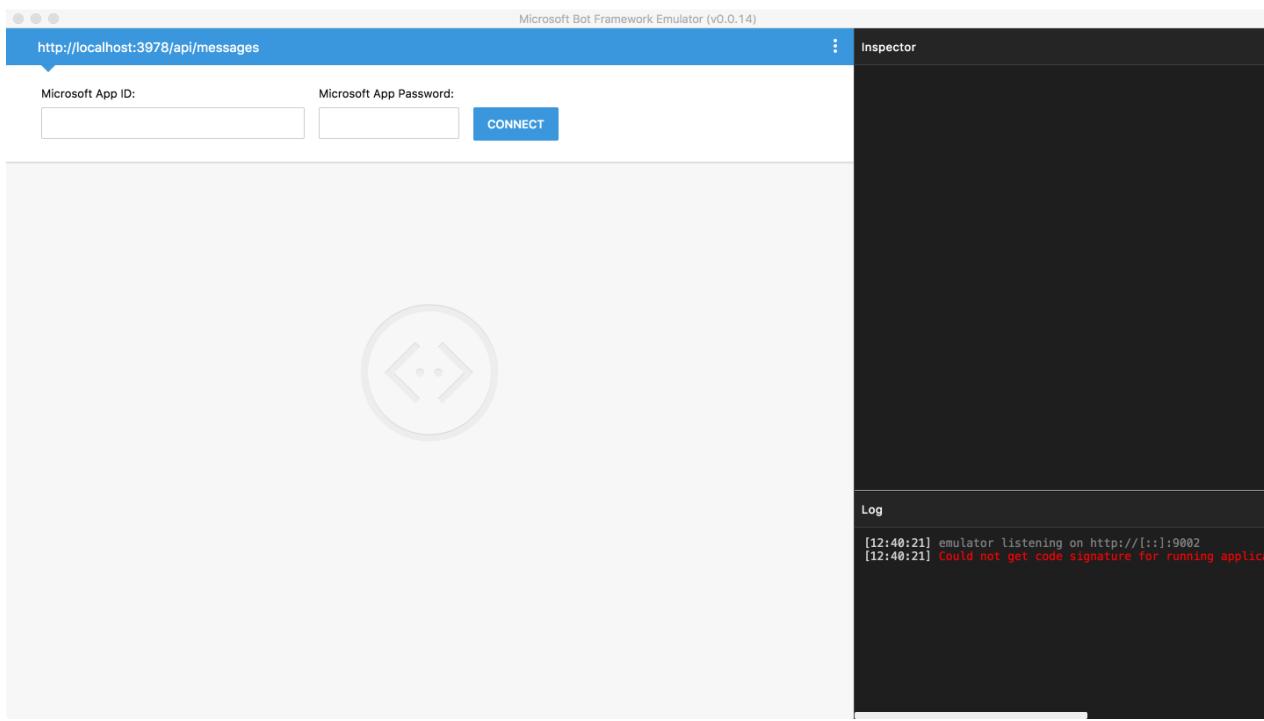
C:\repos\piglatinbot-dotnetfunction\EmptyBot>
```

Next, run `debughost.cmd` to load and start your bot.

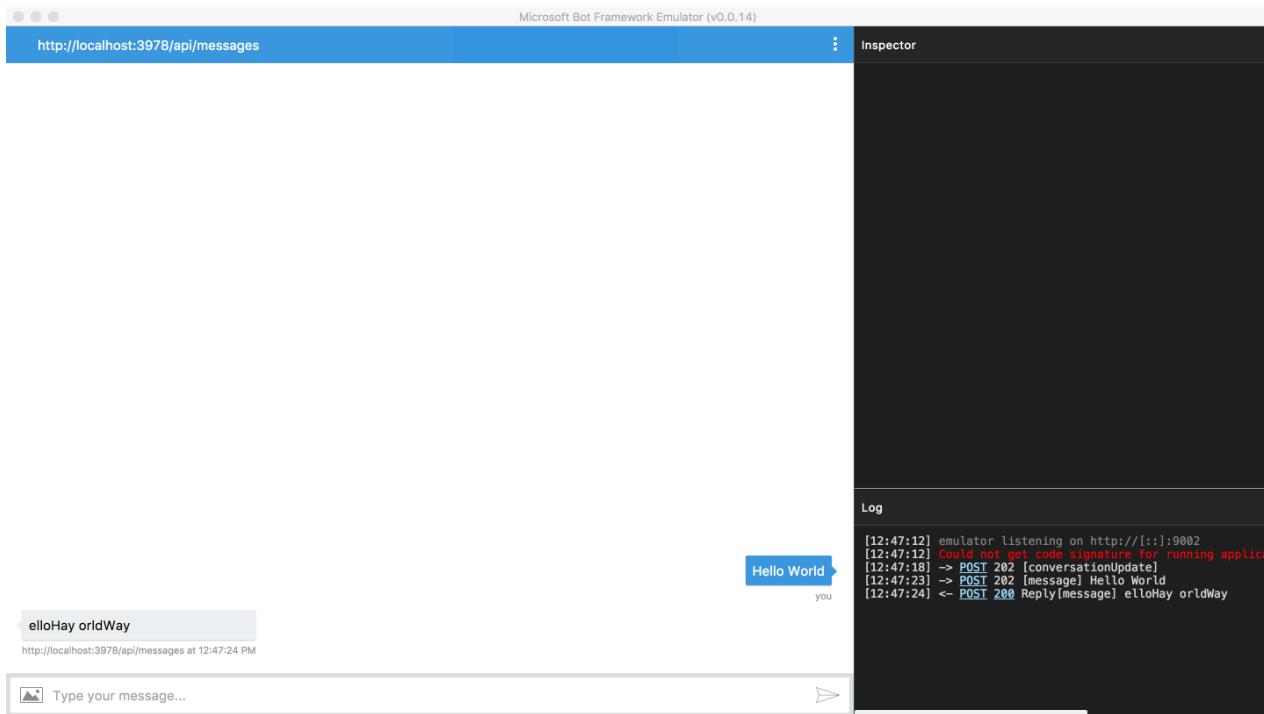
```
Command Prompt - debughost.cmd
C:\repos\piglatinbot-dotnetfunction\EmptyBot>dotnet restore
log : Restoring packages for C:\repos\piglatinbot-dotnetfunction\EmptyBot\project.json...
log : Lock file has not changed. Skipping lock file write. Path: C:\repos\piglatinbot-dotnetfunction\EmptyBot\project.lock.json
on
log : C:\repos\piglatinbot-dotnetfunction\EmptyBot\project.json
log : Restore completed in 194ms.

C:\repos\piglatinbot-dotnetfunction\EmptyBot>debughost.cmd
.gitignore already exists. Skipped!
host.json already exists. Skipped!
appsettings.json already exists. Skipped!
Directory already a git repository.
Listening on http://localhost:3978
Hit CTRL-C to exit...
```

At this point, the bot is running locally. From the console window, copy the endpoint that debughost is listening on (in this example, `http://localhost:3978`). Then, start the Bot Framework Emulator and paste the endpoint into the address bar of the emulator. For this example, you must also append `/api/messages` to the endpoint. Since you do not need security for local debugging, you can leave the **Microsoft App ID** and **Microsoft App Password** fields blank. Click **Connect** to establish a connection to your bot using the specified endpoint.



After you have connected the emulator to your bot, send a message to your bot by typing some text into the textbox that is located at the bottom of the emulator window (i.e., where **Type your message...** appears in the lower-left corner). By using the **Log** and **Inspector** panels on the right side of the emulator window, you can view the requests and responses as messages are exchanged between the emulator and the bot.



Additionally, you can view log details the console window.

```

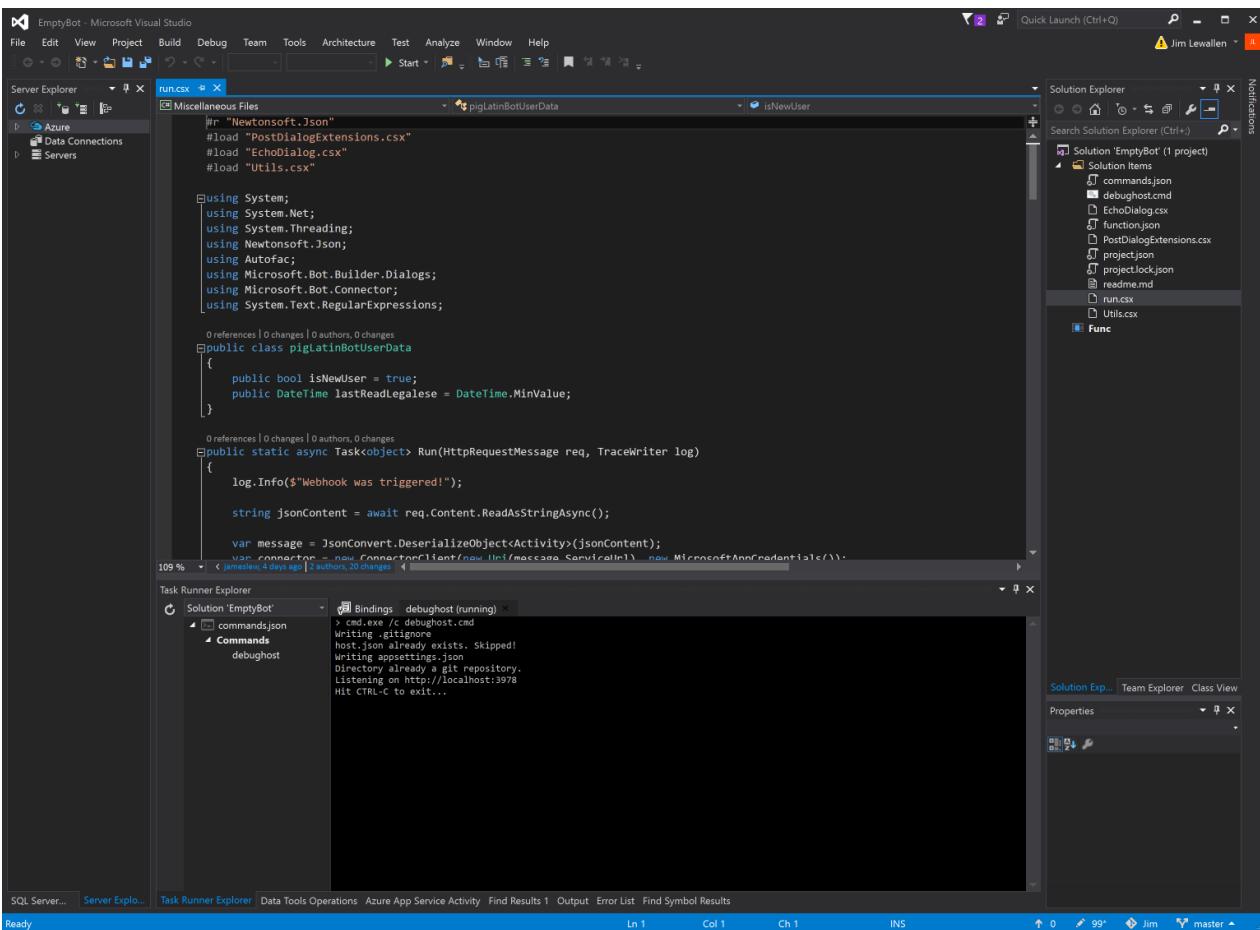
Command Prompt - debughost.cmd
C:\repos\piglatinbot-dotnetfunction\EmptyBot>dotnet restore
log : Restoring packages for C:\repos\piglatinbot-dotnetfunction\EmptyBot\project.json...
log : Lock file has not changed. Skipping lock file write. Path: C:\repos\piglatinbot-dotnetfunction\EmptyBot\project.lock.json
on
log : C:\repos\piglatinbot-dotnetfunction\EmptyBot\project.json
log : Restore completed in 194ms.

C:\repos\piglatinbot-dotnetfunction\EmptyBot>debughost.cmd
.gitignore already exists. Skipped!
host.json already exists. Skipped!
appsettings.json already exists. Skipped!
Directory already a git repository.
Listening on http://localhost:3978
Hit CTRL-C to exit...
Found the following functions:
Host.Functions.EmptyBot
Job host started
Executing: 'Functions.EmptyBot' - Reason: 'This function was programmatically called via the host APIs.'
Webhook was triggered!
Webhook was triggered!
ActivityType conversationUpdate, Activity_Text
ActivityType conversationUpdate, Activity_Text
Executed: 'Functions.EmptyBot' (Succeeded)
Executing: 'Functions.EmptyBot' - Reason: 'This function was programmatically called via the host APIs.'
Webhook was triggered!
Webhook was triggered!
ActivityType message, Activity_Text_Hello
ActivityType message, Activity_Text_Hello
Executed: 'Functions.EmptyBot' (Succeeded)

```

Debug a Consumption plan C# bot using breakpoints in Visual Studio

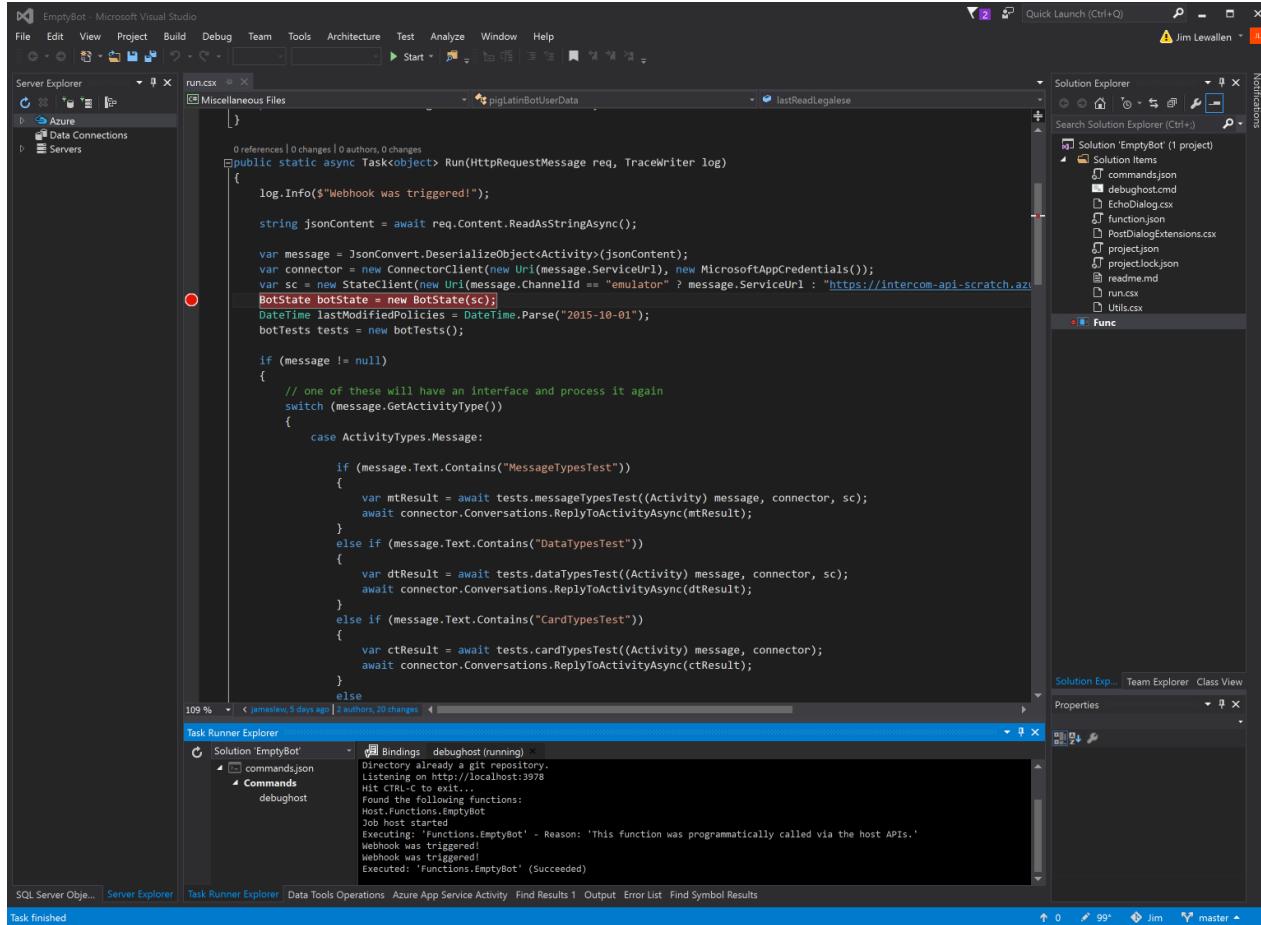
To debug your bot using breakpoints in Visual Studio 2017, stop the **DebugHost.cmd** script, and load the solution for your project (included as part of the repository) in Visual Studio. Then, click **Task Runner Explorer** at the bottom of the Visual Studio window.



You will see the bot loading in the debug host environment in the **Task Runner Explorer** window. Your bot is now running locally. Copy the bot's endpoint from the **Task Runner Explorer** window, start the Bot Framework Emulator, and paste the endpoint into the address bar of the emulator. For this example, you must also append

`/api/messages` to the endpoint. Since you do not need security for local debugging, you can leave the **Microsoft App Id** and **Microsoft App Password** fields blank. Click **Connect** to establish a connection to your bot using the specified endpoint.

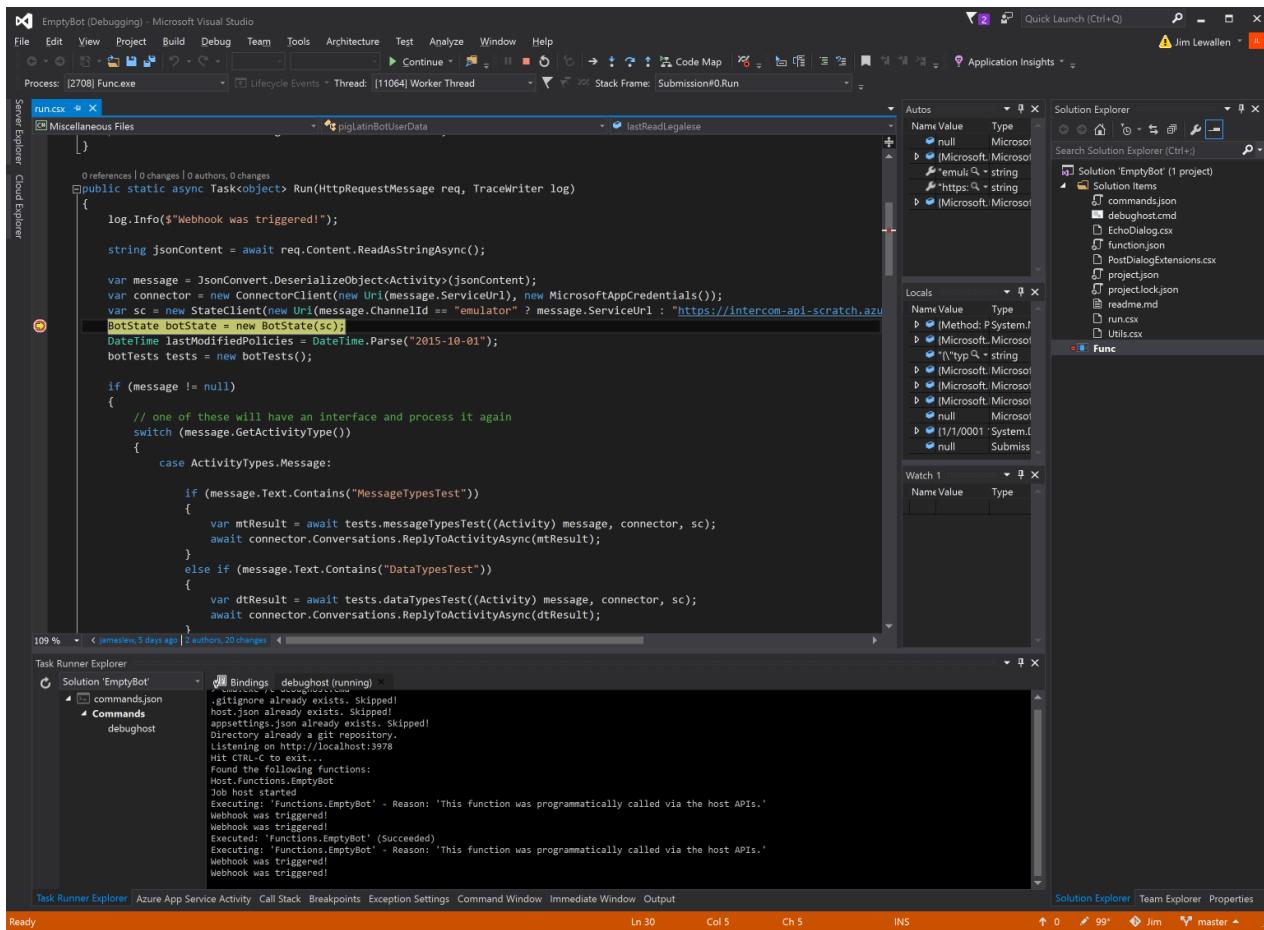
After you have connected the emulator to your bot, send a message to your bot by typing some text into the textbox that is located at the bottom of the emulator window (i.e., where **Type your message...** appears in the lower-left corner). As messages are exchanged between the emulator and the bot, you will see the responses as well as logged output within **Task Runner Explorer** in Visual Studio.



You can also set breakpoints for your bot. The breakpoints are hit only after clicking **Start** in the Visual Studio environment, which will attach to the Azure Function host (`func` command from Azure Functions CLI). Chat with your bot again using the emulator and you should hit the breakpoints that you set in Visual Studio.

TIP

If you cannot successfully set a breakpoint, a syntax error likely exists in your code. To troubleshoot, look for compile errors in the **Task Runner Explorer** window after you try to send messages to your bot.



NOTE

By following the steps that are described in this article, you will be able to debug a majority of bots that you might build using Azure Bot Service. However, if you create a bot using the [Proactive template](#), you must do some additional work to enable queue storage that is used between the trigger function and the bot function. More information on this topic will be made available soon.

Debug an Azure App Service web app C# bot

You can debug a C# bot Azure Bot Service web app locally in Visual Studio.

Prerequisites

Before you can debug your web app C# bot, you must complete these tasks.

- Download and install the [Bot Framework Channel Emulator](#).
- Download and install [Visual Studio 2017](#) (Community Edition or above).

Get and debug your source code

Follow these steps to download your C# bot source.

1. In the Azure Portal, click your Bot Service, click the **BUILD** tab, and click **Download zip file**.
2. Extract the contents of the downloaded zip file to a local folder.
3. In Explorer, double-click the .sln file.
4. Click **Debug**, and click **Start Debugging**. Your bot's `default.htm` page appears. Note the address in the location bar.
5. In the Bot Framework Channel Emulator, click the blue location bar, and enter an address similar to `http://localhost:3984/api/messages`. Your port number might be different.

You are now debugging locally. You can simulate user activity in the emulator, and set breakpoints on your code

in Visual Studio. You can then [re-publish your bot to Azure](#).

Test a Cortana skill

9/13/2017 • 5 min to read • [Edit Online](#)

If you've built a Cortana skill using the Bot Builder SDK, you can test it by invoking it from Cortana. The following instructions walk you through the steps required to try out your Cortana skill.

Register your bot

Log in to the [Bot Framework Portal](#) using a [Microsoft account](#). If you haven't [registered](#) your bot yet, you can do so now.

NOTE

To register a bot as Cortana skill, you must be signed in with a Microsoft account. Signing up for an Outlook.com mailbox automatically creates a Microsoft account. Support for work and school accounts is coming soon.

Enable speech recognition priming

If your bot uses a Language Understanding Intelligent Service (LUIS) app, make sure you register the LUIS application ID. This helps your bot recognize spoken utterances that are defined in your LUIS model.

In the **Settings** panel, under **Configuration**, enter the LUIS application ID in the **Speech recognition priming with LUIS** text box.

Speech recognition priming with LUIS

In order to use speech recognition with your LUIS-powered bot, please select the LUIS applications referenced in your bot.

Did not find any LUIS apps in your account

Enter a LUIS application ID

Add

Add the Cortana channel

Under **My bots**, select the bot you would like to connect to the Cortana channel. From the list of channels, click the button to add Cortana.

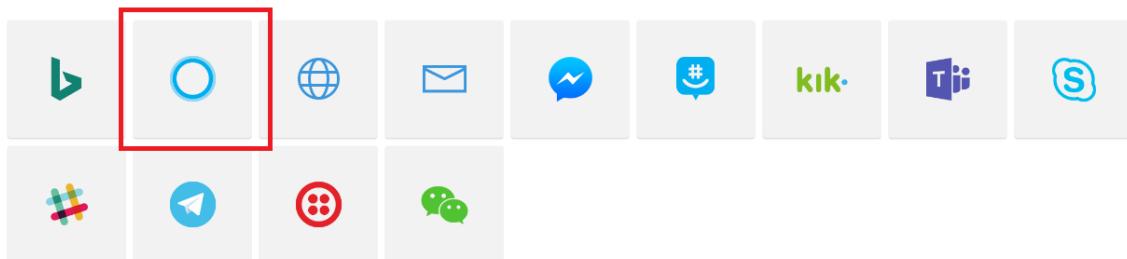
Secure your bot by configuring the **AppID** and **AppPassword** in the Bot Framework portal and in your **Web.config** file.

Connect to channels

Name	Health	Published	
 Skype	Running	--	Edit 
 Web Chat	Running	--	Edit 

[Get bot embed codes](#)

Add a channel



NOTE

You should set the messaging endpoint of your bot in the Bot Framework portal before connecting to Cortana.

If you have already deployed your bot, make sure the **Messaging endpoint** is correctly set.

If you are running the bot locally, you can get an endpoint to use for testing by running tunneling software, such as [ngrok](#). To use ngrok to get an endpoint, from a console window type:

```
ngrok.exe http 3979 -host-header="localhost:3979"
```

This configures and displays an ngrok forwarding link that forwards requests to your bot, which is running on port 3978. The URL to the forwarding link should look something like this: <https://0d6c4024.ngrok.io>. Append </api/messages> to the link, to form an endpoint URL in this format: <https://0d6c4024.ngrok.io/api/messages>. Enter this endpoint URL in the **Configuration** section for your bot in the [Bot Framework Portal](#).

Configure Cortana

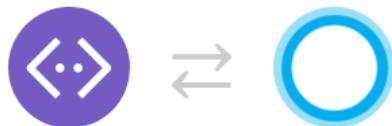
When registering your bot with the Cortana channel, some basic information about your bot will be pre-filled into the registration form. Review this information carefully. This form consists of the following fields.

FIELD	DESCRIPTION
Bot icon	An icon that is displayed in the Cortana canvas when your skill is invoked. This is also used where skills are discoverable (like the Microsoft store). The image should be a PNG that is 60 x 60 pixels and be no more than 30kb in size.
Name	The name of your Cortana skill is displayed to the user at the top of the visual UI.

FIELD	DESCRIPTION
Invocation name	This is the name users say when invoking a skill. It should be no more than three words and easy to pronounce. See the Invocation Name Guidelines for more information on how to choose this name.
Description	A description of your Cortana skill. This is used where skills are discoverable (like the Microsoft store).
Short description	A short description of your skill's functionality, used to describe the skill in Cortana's notebook.



Configure Cortana



Use this bot to power a Cortana skill

Adding the Cortana channel means that Cortana becomes aware of your bot's capabilities as a skill. This registration allows users to invoke and converse with your bot through Cortana's user interface.

To learn more about Cortana skills, please take a look at the [Cortana Skills Kit developer docs](#).



* Skill icon

[Upload custom icon](#)

32,000 byte maximum size, png only

* Display name

Dice Roller

* Invocation name

Dice Roller

* Long description

This is a bot that rolls dice.

* Short description

This is a bot that rolls dice.

Request user profile data (Optional)

Cortana provides access to several different types of user profile information, that you can use to customize the bot for the user.

NOTE

You can skip this step if you don't need to use user profile data in your bot.

To add user profile information, click the **Add a user profile request** link, then select the user profile information you want from the drop-down list. Add a friendly name to use to access this information from your bot's code. See [Cortana-specific entities](#) for more information on using these fields.

Request user profile data

With the user's consent, bots can utilize user information from Cortana's notebook to customize interactions. Request user profile information below.

Data
Friendly Name

User.SemanticLocation.Current	▼
CurrentLocation	

[Remove](#)

[Add a user profile request](#)

Manage user identity through connected services (Optional)

If your skill requires authentication, you can connect an account so that Cortana will require users to log in into your skill before they can use it. Currently, only **Auth Code Grant** authentication is supported, and **Implicit Grant** is not supported. See [Secure your skill with authentication](#) for more information.

NOTE

You can skip this step if your bot doesn't require authentication.

Connect to Cortana

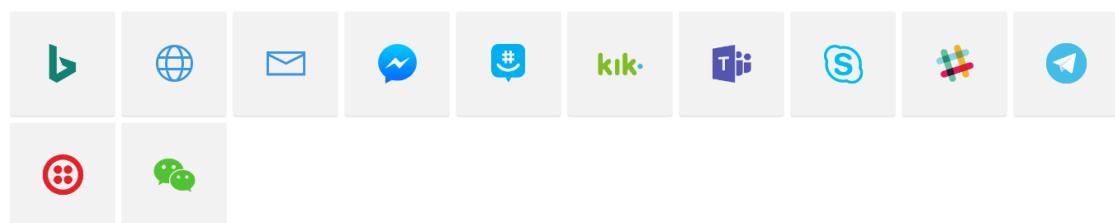
When you are done filling out the registration form for your Cortana skill, click **Save** to complete the connection. This brings you back to your bot's main page in the Bot Framework developer center and you should see that it is now connected to Cortana.

Connect to channels

Name	Health	Published	
Cortana	Running	Manage in Cortana dashboard	Edit
Skype	Running	--	Edit
Web Chat	Running	--	Edit

[Get bot embed codes](#)

Add a channel



Test your Cortana skill using the Chat control

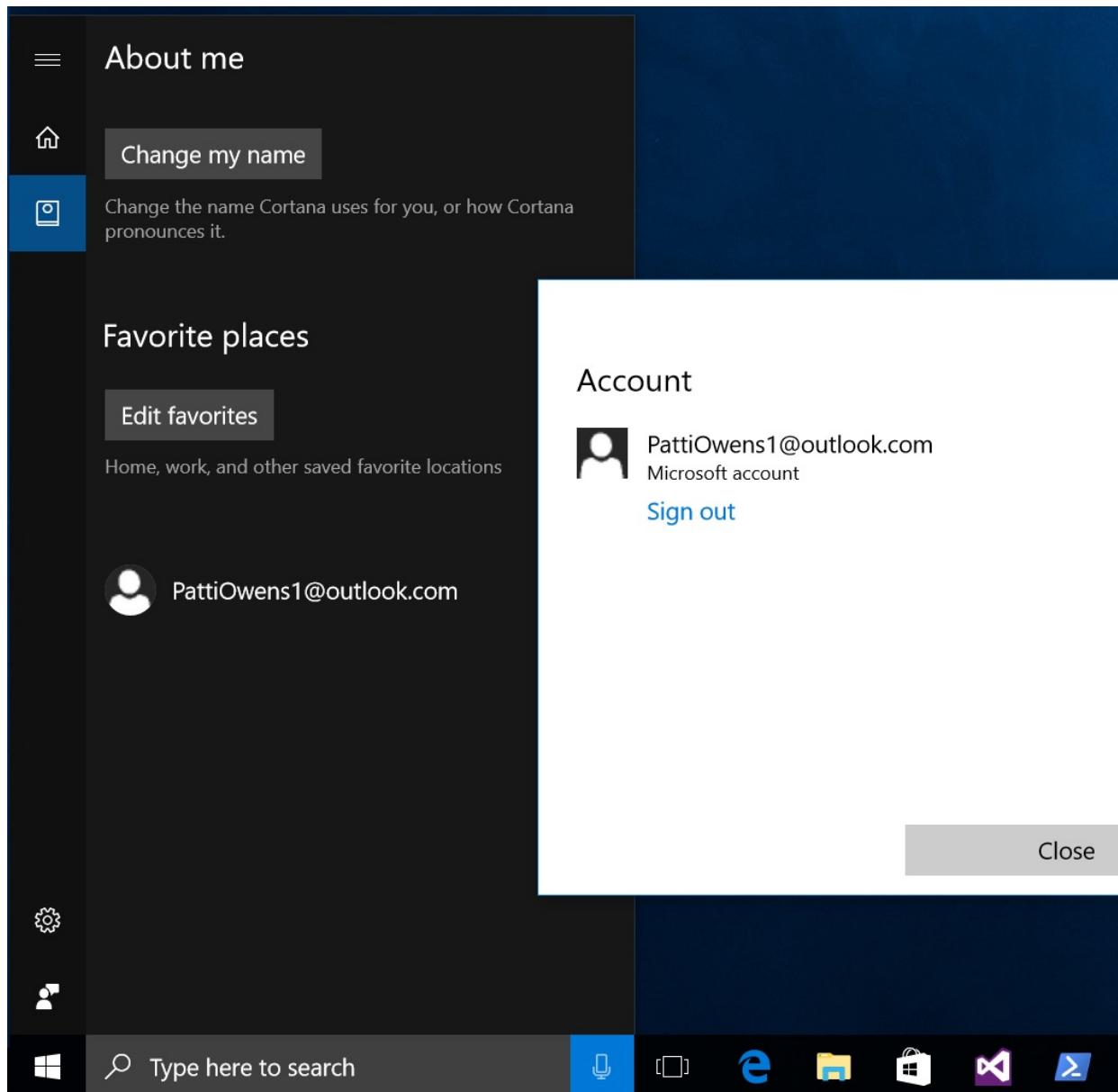
Now that your bot is available as a Cortana skill you should test it. At this point your bot has already been automatically deployed as a Cortana skill to your account.

Click **Test** to open the **Chat** window in the Bot Framework portal and type a message to verify that your bot is working.

Test your Cortana skill

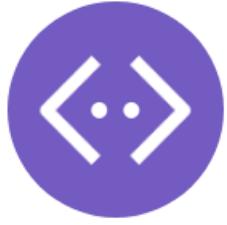
You can invoke your Cortana skill by speaking an invocation phrase to Cortana.

1. Open the Notebook within Cortana and click **About me** to see which account you're using for Cortana. Make sure you are signed in with the same Microsoft account that you used to register your bot.



2. Click on the microphone button in the Cortana app or in the "Ask me anything" search box in Windows, and say your bot's **invocation phrase**. The invocation phrase includes an *invocation name*, which uniquely identifies the skill to invoke. For example, if a skill's invocation name is "Northwind Photo", a proper invocation phrase could include "Ask Northwind Photo to..." or "Tell Northwind Photo that...".

You specify your bot's *Invocation Name* when you configure it for Cortana.



* Skill icon

[Upload custom icon](#)

32,000 byte maximum size, png only

* Display name

Dice Roller

* Invocation name

Roller

* Long description

This is a bot that rolls dice.

* Short description

Rolls dice.

3. If Cortana recognizes your invocation phrase, your bot launches in Cortana's canvas.

Troubleshoot

If your Cortana skill fails to launch, check the following:

- Make sure you are signed in to Cortana using the same Microsoft account that you used to register your bot in the Bot Framework Portal.
- Check if the bot is working by clicking **Test** to open the **Chat** window of the Bot Framework portal and typing a message to it.
- Check if your invocation name meets the [guidelines](#). If your invocation name is longer than three words, hard to pronounce, or sounds like other words, Cortana might have difficulty recognizing it.
- If your skill uses a LUIS model, make sure you [enable speech recognition priming](#).

See the [Enable Debugging of Cortana skills](#) for additional troubleshooting tips and information on how to enable debugging of your skill in the Cortana dashboard.

Next steps

Once you have tested your Cortana skill and verified that it works the way you'd like it to, you can deploy it to a group of beta testers or release it to the public. See [Publishing Cortana Skills](#) for more information.

Additional resources

- [The Cortana Skills Kit](#)
- [Cortana Dev Center](#)
- [Testing and debugging best practices](#)
- [Preview features with the Channel Inspector](#)

Deploy a bot to the cloud

10/4/2017 • 1 min to read • [Edit Online](#)

After you have built and tested your bot, you need to deploy it to the cloud.

NOTE

Bots created with the Azure Bot Service are automatically deployed to Azure as part of the creation process.

Deploy your bot to the cloud

Before others can use your bot, you must deploy it to the cloud. You can deploy it to Azure or to any other cloud service. These articles describe various techniques for deploying your bot to Azure:

- [Deploy from a local git repository](#) using continuous deployment
- [Deploy from GitHub](#) using continuous deployment
- [Deploy a .NET bot from Visual Studio](#)
- [Deploy a Node.js bot from Visual Studio](#)

Next steps

- [Register and publish a bot](#)
- [Configure a bot to run on one or more channels.](#)

Deploy a bot to Azure from a local git repository

9/7/2017 • 5 min to read • [Edit Online](#)

Azure allows continuous deployment from your Git repository to Azure. With continuous deployment, when you change and build your bot's code, the bot will automatically deploy to Azure. This tutorial shows you how to deploy a bot to Azure via continuous deployment from a local Git repository.

NOTE

If you created a bot with the Azure Bot Service, your bot deployment was part of the Azure Bot Service bot creation process.

Prerequisites

You must have a Microsoft Azure subscription before you can deploy a bot to Azure. If you do not already have a subscription, you can register for a [free trial](#). You will also need [Git](#).

Application settings and Messaging endpoint

Verify application settings

For your bot to function properly in the cloud, you must ensure that its application settings are correct. If you've already [registered](#) your bot with the Bot Framework, update the Microsoft App Id and Microsoft App Password values in your application's configuration settings as part of the deployment process. Specify the **app ID** and **password** values that were generated for your bot during registration.

TIP

If you're using the Bot Builder SDK for Node.js, set the following environment variables:

- `MICROSOFT_APP_ID`
- `MICROSOFT_APP_PASSWORD`

If you're using the Bot Builder SDK for .NET, set the following key values in the `web.config` file:

- `MicrosoftAppId`
- `MicrosoftAppPassword`

If you have not yet registered your bot with the Bot Framework (and therefore do not yet have an **app ID** and **password**), you can deploy your bot with temporary placeholder values for these settings. Then later, after you register your bot, update your deployed application's settings with the **app ID** and **password** values that were generated for your bot during registration.

Verify Messaging endpoint

Your deployed bot must have an **Messaging endpoint** that can receive messages from the Bot Framework Connector Service.

NOTE

When you deploy your bot to Azure, SSL will automatically be configured for your application, thereby enabling the **Messaging endpoint** that the Bot Framework requires. If you deploy to another cloud service, be sure to verify that your application is configured for SSL so that the bot will have a **Messaging endpoint**.

Step 1: Install the Azure CLI

Download and [install Azure CLI 2.0](#).

Step 2: Create and configure an Azure site

Open a command prompt, Powershell, or other console and log in to your Azure account using the following command:

```
az login
```

Afer you log in, run the following command to create a resource group. Specify the name of the resource group you want to create and the location, such as *westus*, where your bot will be deployed.

```
az group create --location <Location> --name <MyResourceGroup>
```

Run the following command to create a plan. Specify a name for the plan and the resource group name you specified in the previous step.

```
az appservice plan create --name <MyPlan> --resource-group <MyResourceGroup> --sku FREE
```

The next step is to create a new Azure site. Find your bot's handle in the [Bot Framework Portal](#) on your bot's **SETTINGS** page. Run the following command with your bot's handle, the name of the resource group, and the name of your plan:

```
az webapp create --name <MyBotHandle> --resource-group <MyResourceGroup> --plan <MyPlan>
```

Git requires a user name-password pair to deploy a bot. The user name and password are account-level. They are different from your Azure subscription credentials. The user name must be unique, and the password must be at least eight characters long, with two of the following three elements: letters, numbers, symbols. You create this deployment user only once and you can use it for all your Azure deployments. Record the user name and password. You need them to deploy the bot later.

Create a deployment account with the following command with a user name and a password:

```
az webapp deployment user set --user-name <UserName> --password <Password>
```

TIP

If you get a '**Conflict**'. **Details: 409 error**, change the user name. If you get a '**Bad Request**'. **Details: 400 error**, use a stronger password.

Find the URL where Git will push your source code changes by running the following command:

```
az webapp deployment source config-local-git --name <MyBotHandle> --resource-group <MyResourceGroup> --query url --output tsv
```

Your URL will look similar to <https://<UserName>@<MyBotHandle>.scm.azurewebsites.net/<MyBotHandle>.git>.

Connect your local Git repository to the bot hosted on Azure. A dialog will appear where you must provide the user name and password credentials you created earlier.

```
cd <MyGitRepo>
git remote add azure <URLResultFromLastStep>
git push azure master
```

Step 3: Commit changes to Git and push to the Azure site

To update the site with your latest changes, run the following commands to commit the changes and push those changes to the Azure site:

```
git add .
git commit -m "<your commit message>"
git push azure master
```

A dialog will appear where you must provide the user name and password credentials you created earlier.

Step 4: Test the connection to your bot

Verify the deployment of your bot by using the [Bot Framework Emulator](#).

Enter the bot's **Messaging endpoint** into the address bar of the Emulator. If you built your bot with the Bot Builder SDK, the endpoint should end with `/api/messages`.

Example: <https://<appname>.azurewebsites.net/api/messages>

NOTE

Because your bot runs on Azure, you must enter its **app ID** and **password** into the emulator to connect, and because it's running remotely, you might need to provide the emulator with a path to **ngrok** tunneling software on your computer.

Next steps

After you have deployed your bot to the cloud and verified that the deployment was successful by testing the bot using the Bot Framework Emulator, the next step in the bot publication process will depend upon whether or not you've already registered your bot with the Bot Framework.

If you have already registered your bot with the Bot Framework:

1. Return to the [Bot Framework Portal](#) and [update your bot's registration data](#) to specify the **Messaging endpoint** for the bot.
2. [Configure the bot to run on one or more channels](#).

If you have not yet registered your bot with the Bot Framework:

1. [Register your bot with the Bot Framework](#).
2. [Update the Microsoft App Id and Microsoft App Password values](#) in your deployed application's configuration settings to specify the **app ID** and **password** values that were generated for your bot during

the registration process.

3. [Configure the bot to run on one or more channels.](#)

Deploy a bot to Azure from GitHub

10/24/2017 • 4 min to read • [Edit Online](#)

Azure allows continuous deployment of your Git repository to Azure. With continuous deployment, when you change and build your bot's code, the bot will automatically deploy to Azure. This tutorial shows you how to deploy a bot to Azure via continuous deployment from GitHub.

NOTE

If you created a bot with the Azure Bot Service, your bot deployment was part of the Azure Bot Service bot creation process.

Prerequisites

You must have a Microsoft Azure subscription before you can deploy a bot to Azure. If you do not already have a subscription, you can register for a [free trial](#). Additionally, the process described by this article requires [Git](#) and a [GitHub](#) account.

Application settings and Messaging endpoint

Verify application settings

For your bot to function properly in the cloud, you must ensure that its application settings are correct. If you've already [registered](#) your bot with the Bot Framework, update the Microsoft App Id and Microsoft App Password values in your application's configuration settings as part of the deployment process. Specify the **app ID** and **password** values that were generated for your bot during registration.

TIP

If you're using the Bot Builder SDK for Node.js, set the following environment variables:

- `MICROSOFT_APP_ID`
- `MICROSOFT_APP_PASSWORD`

If you're using the Bot Builder SDK for .NET, set the following key values in the `web.config` file:

- `MicrosoftAppId`
- `MicrosoftAppPassword`

If you have not yet registered your bot with the Bot Framework (and therefore do not yet have an **app ID** and **password**), you can deploy your bot with temporary placeholder values for these settings. Then later, after you register your bot, update your deployed application's settings with the **app ID** and **password** values that were generated for your bot during registration.

Verify Messaging endpoint

Your deployed bot must have an **Messaging endpoint** that can receive messages from the Bot Framework Connector Service.

NOTE

When you deploy your bot to Azure, SSL will automatically be configured for your application, thereby enabling the **Messaging endpoint** that the Bot Framework requires. If you deploy to another cloud service, be sure to verify that your application is configured for SSL so that the bot will have a **Messaging endpoint**.

Step 1: Get a GitHub repository

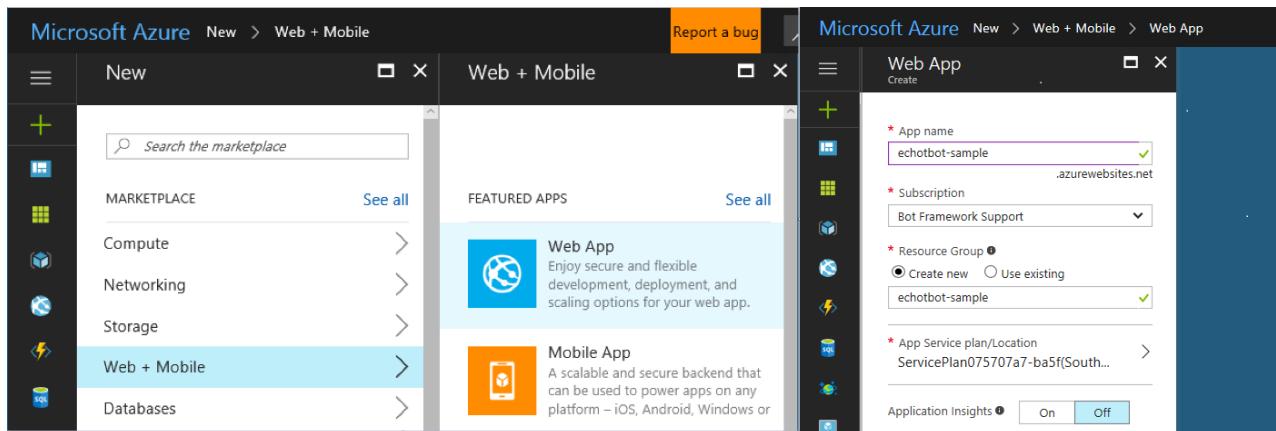
Start by [forking](#) the GitHub repository that contains the code for the bot that you want to deploy.

NOTE

This tutorial uses the [echobot](#) GitHub repository, which contains the Node.js code for creating a simple bot. Be sure to replace "echobotsample" with your bot ID in all settings and URLs that are shown in the examples.

Step 2: Create an Azure web app

Next, log in to the [Azure Portal](#) and create an Azure web app.



Step 3: Set up continuous deployment from your GitHub repository to Azure

Specify GitHub as the **Deployment Option** for your web app. When you are asked to authorize Azure access to your GitHub repo, choose the branch from which to deploy.

The screenshot shows two side-by-side windows of the Microsoft Azure portal. The left window is titled 'Microsoft Azure echotbot-sample' and shows the 'Deployment' section of the 'echotbot-sample' App Service. The right window is titled 'Microsoft Azure echotbot-sample > Deployment option > Choose source' and displays a list of deployment source options: Visual Studio Team Services, OneDrive, Local Git Repository, GitHub, Bitbucket, Dropbox, and External Repository. The 'GitHub' option is highlighted.

The deployment process may take a minute or two to complete. You can verify that the deployment has completed by visiting the web app in a browser.

NOTE

The URL of the web app will be `https://appname.azurewebsites.net`, where **appname** is the value that you specified when creating the app. In this example, the URL is `https://echobotsample.azurewebsites.net`.

The screenshot shows the 'echobotsample' Web app in the Azure portal. On the left, the 'Essentials' blade displays resource group (XXXX), status (Running), location (West US), subscription name (XXXX), and subscription ID (XXXX). On the right, a browser window titled 'EchoBot' shows the URL 'echobotsample.azurewebsites.net'. The page content reads 'Hello world! This is the EchoBot home page :)'.

Step 4: Update Application settings with bot credentials

Update **Application settings** to specify values for `MICROSOFT_APP_ID` and `MICROSOFT_APP_PASSWORD` using the values that you acquired when you [registered](#) the bot in the Bot Framework Portal.

NOTE

If you have not yet registered the bot in the Bot Framework Portal, you can populate `MICROSOFT_APP_ID` and `MICROSOFT_APP_PASSWORD` with temporary (placeholder) values for now. After you register your bot, return to the Azure Portal and update these values with the **App ID** and **App Password** values that you acquire during the registration process.

The screenshot shows the Azure portal interface for managing an App Service application named "echobot-sample". The left sidebar contains a navigation menu with icons for Overview, Activity log, Access control (IAM), Tags, Diagnose and solve problems, Deployment (Quickstart, Deployment credentials, Deployment slots, Deployment options, Continuous Delivery (Preview)), and Settings (Application settings, Authentication / Authorization). The "Application settings" item is currently selected and highlighted in blue. The main content area is titled "Application settings" and includes sections for "ARR Affinity" (set to "On"), "Auto swap" (set to "Off"), "Auto Swap Slot" (a dropdown menu), "Debugging" (Remote debugging set to "Off", Remote Visual Studio version set to "2017"), and "App settings". The "App settings" section lists environment variables: WEBSITE_NODE_DEFAULT_VERSION (value: 6.9.1, slot setting checkbox), MICROSOFT_APP_ID (value: 12345, slot setting checkbox), and MICROSOFT_APP_PASSWORD (value: XXXXX, slot setting checkbox). The MICROSOFT_APP_PASSWORD row is highlighted with a blue background, indicating it is the current focus or being edited.

Step 5: Test the connection to your bot

Verify the deployment of your bot by using the [Bot Framework Emulator](#).

Enter the bot's **Messaging endpoint** into the address bar of the Emulator. If you built your bot with the Bot Builder SDK, the endpoint should end with `/api/messages`.

Example: `https://<appname>.azurewebsites.net/api/messages`

Next steps

After you have deployed your bot to the cloud and verified that the deployment was successful by testing the bot using the Bot Framework Emulator, the next step in the bot publication process will depend upon whether or not you've already registered your bot with the Bot Framework.

If you have already registered your bot with the Bot Framework:

1. Return to the [Bot Framework Portal](#) and [update your bot's registration data](#) to specify the **Messaging endpoint** for the bot.
2. [Configure the bot to run on one or more channels](#).

If you have not yet registered your bot with the Bot Framework:

1. [Register your bot with the Bot Framework](#).
2. [Update the Microsoft App Id and Microsoft App Password values](#) in your deployed application's

configuration settings to specify the **app ID** and **password** values that were generated for your bot during the registration process.

3. [Configure the bot to run on one or more channels.](#)

Deploy a .NET bot to Azure from Visual Studio

10/4/2017 • 4 min to read • [Edit Online](#)

When you build a bot with Visual Studio, you can take advantage of its built-in publishing capability. This tutorial shows you how to deploy a .NET bot to Azure directly from Visual Studio 2017.

NOTE

If you created a bot with the Azure Bot Service, your bot deployment was part of the Azure Bot Service bot creation process. For more information, see [Publish a bot to Azure Bot Service](#).

Prerequisites

- You must have a Microsoft Azure subscription. If you do not already have a subscription, you can register for a [free trial](#).
- Install Visual Studio 2017. If you do not already have Visual Studio, you can download [Visual Studio 2017 Community](#) for free.
- Install **Azure development**. You can choose to have **Azure development** installed while installing Visual Studio or you can add it if you already have Visual Studio installed. You can add it by going to your computer's **Uninstall or change a program**, find **Microsoft Visual Studio 2017** and choose to **Modify** it. When the **Workloads** window shows up, check the box for **Azure development** and click **Modify** to install this resource.

Application settings and Messaging endpoint

Verify application settings

For your bot to function properly in the cloud, you must ensure that its application settings are correct. If you've already [registered](#) your bot with the Bot Framework, update the Microsoft App Id and Microsoft App Password values in your application's configuration settings as part of the deployment process. Specify the **app ID** and **password** values that were generated for your bot during registration.

TIP

If you're using the Bot Builder SDK for Node.js, set the following environment variables:

- `MICROSOFT_APP_ID`
- `MICROSOFT_APP_PASSWORD`

If you're using the Bot Builder SDK for .NET, set the following key values in the `web.config` file:

- `MicrosoftAppId`
- `MicrosoftAppPassword`

If you have not yet registered your bot with the Bot Framework (and therefore do not yet have an **app ID** and **password**), you can deploy your bot with temporary placeholder values for these settings. Then later, after you register your bot, update your deployed application's settings with the **app ID** and **password** values that were generated for your bot during registration.

Verify Messaging endpoint

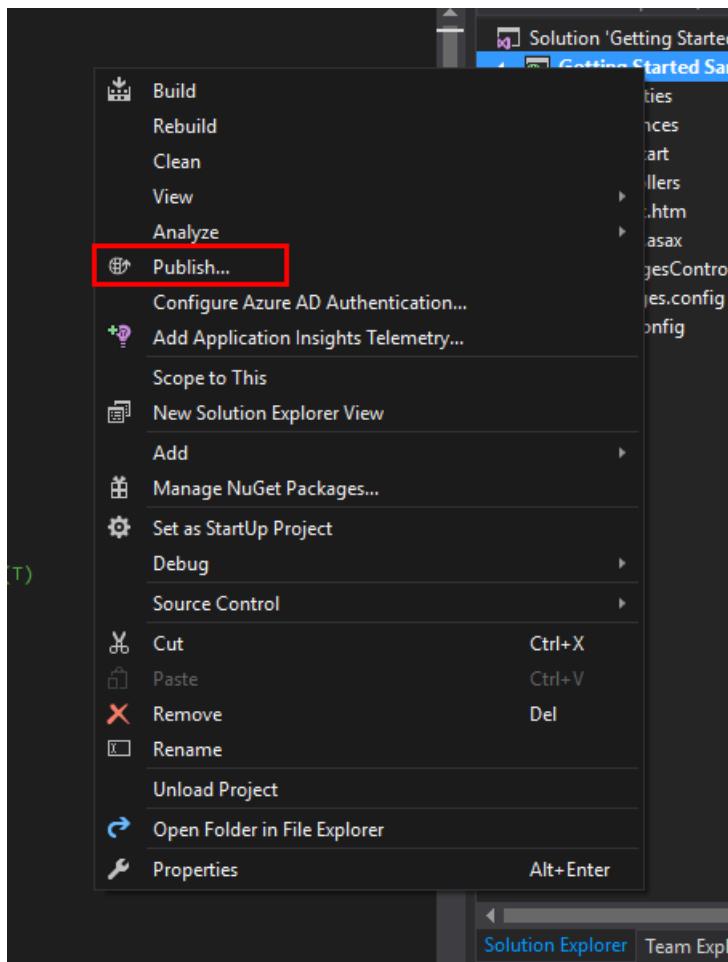
Your deployed bot must have an **Messaging endpoint** that can receive messages from the Bot Framework

NOTE

When you deploy your bot to Azure, SSL will automatically be configured for your application, thereby enabling the **Messaging endpoint** that the Bot Framework requires. If you deploy to another cloud service, be sure to verify that your application is configured for SSL so that the bot will have a **Messaging endpoint**.

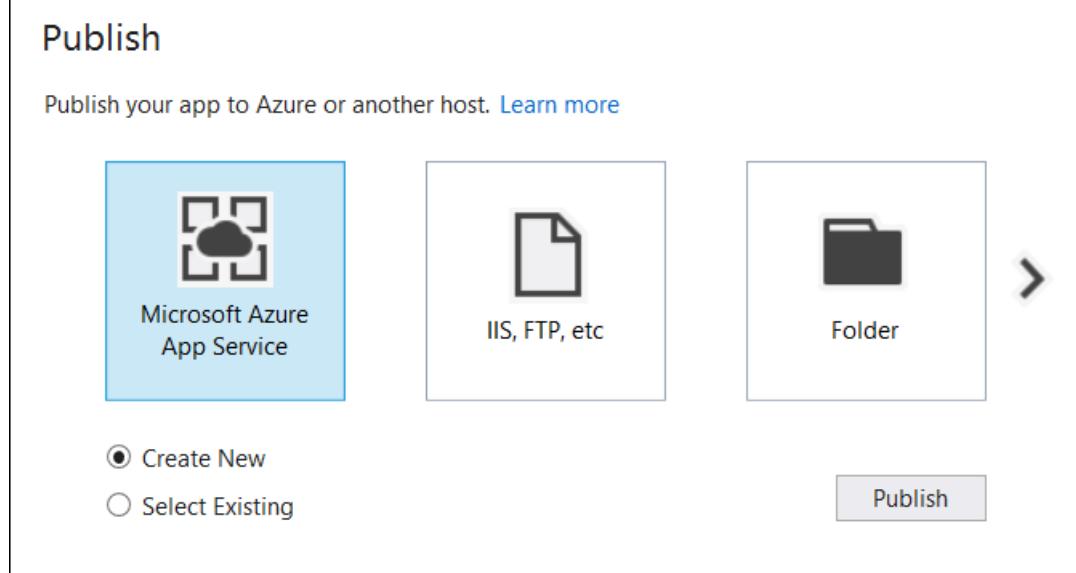
Step 1: Launch the Microsoft Azure Publishing Wizard in Visual Studio

Open your .NET bot project in Visual Studio. In Solution Explorer, right-click on the project name and select **Publish**. This starts the Microsoft Azure publishing wizard.

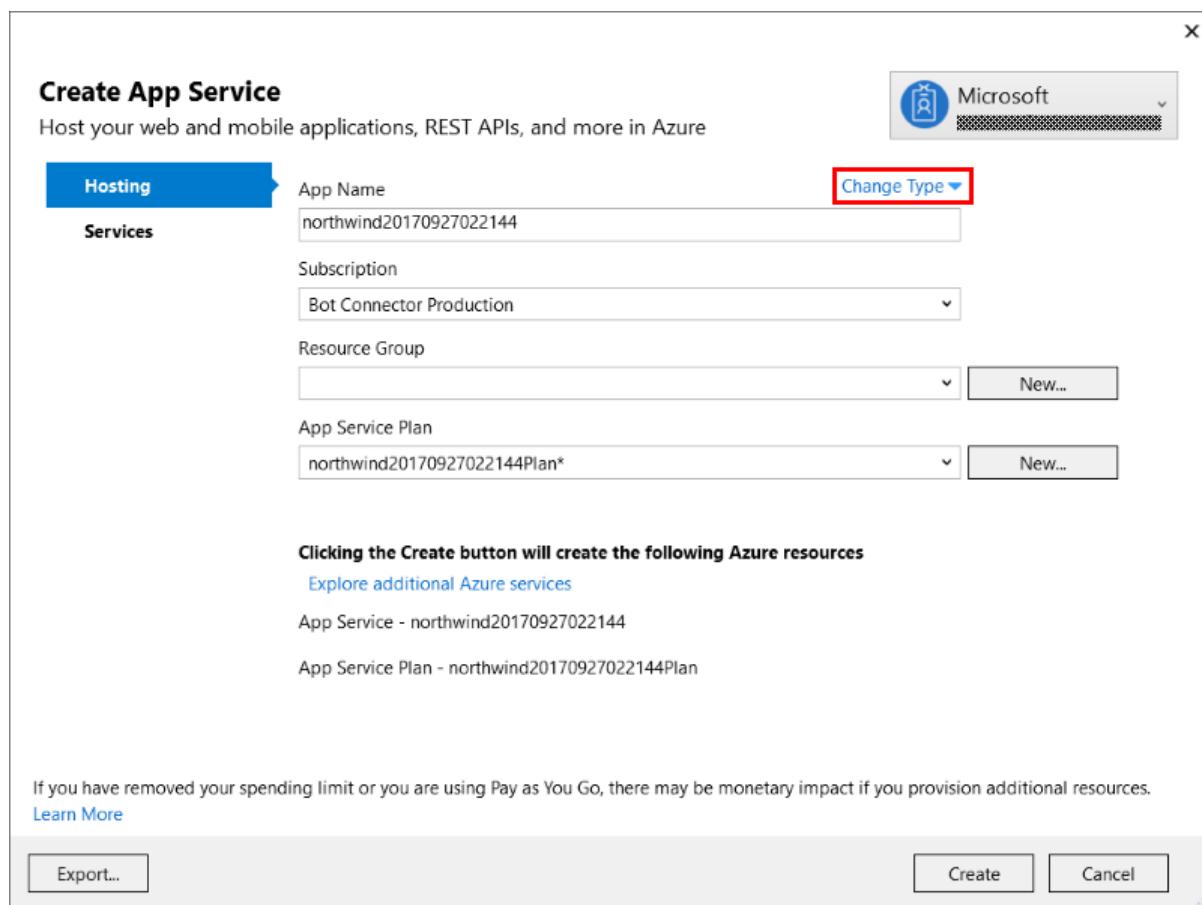


Step 2: Publish to Azure using the Azure Publishing Wizard

1. Select **Microsoft Azure App Service** as the project type and click **Publish**.



2. Click **Change Type** and change the service's type to **Web App**. Then, name your web app and fill out the rest of the information as appropriate for your implementation.



3. Click **Create** to create your app service. Once the service is created, the bot will be published to Azure and your bot's HTML page will be displayed in your default browser.

Copy the **Site URL** value to the clipboard. You will need this value later to test the connection to the bot using the Emulator.

Publish

Publish your app to Azure or another host. [Learn more](#)

The screenshot shows the 'Publish' summary page. At the top, there's a dropdown menu showing 'northwind20170927022144 - Web Deploy' with a 'Publish' button next to it. Below this is a 'Create new profile' link. The main area is titled 'Summary' and contains the following information:

Site URL	http://northwind20170927022144.azurewebsites.net <input type="button" value="Copy"/>	Settings...
Resource Group	AADv2	Preview...
Configuration	Release	Rename profile...
Username	\$northwind20170927022144	Delete profile
Password	*****	

By default, your bot will be published in a **Release** configuration. If you want to debug your bot, in **Settings**, change **Configuration** to **Debug**. Click **Save** to save your settings.

The screenshot shows the 'Publish' dialog box. At the top left is a globe icon and the word 'Publish'. On the right are standard window controls. The main area has a title bar 'northwind20170927022144 - Web Deploy'. Below this, the 'Settings' tab is selected and highlighted in blue. A red box highlights the 'Configuration: Debug' dropdown. Underneath is a 'File Publish Options' section with a collapse arrow. The 'Databases' section shows a message: 'No databases found in the project'. At the bottom are navigation buttons: '< Prev' and 'Next >' (disabled), a blue 'Save' button, and a 'Cancel' button.

Step 3: Test the connection to your bot

Verify the deployment of your bot by using the [Bot Framework Emulator](#).

Enter the bot's **Messaging endpoint** into the address bar of the Emulator. If you built your bot with the Bot Builder

SDK, the endpoint should end with `/api/messages`.

Example: `https://<appname>.azurewebsites.net/api/messages`

Next steps

After you have deployed your bot to the cloud and verified that the deployment was successful by testing the bot using the Bot Framework Emulator, the next step in the bot publication process will depend upon whether or not you've already registered your bot with the Bot Framework.

If you have already registered your bot with the Bot Framework:

1. Return to the [Bot Framework Portal](#) and [update your bot's registration data](#) to specify the **Messaging endpoint** for the bot.
2. [Configure the bot to run on one or more channels](#).

If you have not yet registered your bot with the Bot Framework:

1. [Register your bot with the Bot Framework](#).
2. [Update the Microsoft App Id and Microsoft App Password values](#) in your deployed application's configuration settings to specify the **app ID** and **password** values that were generated for your bot during the registration process.
3. [Configure the bot to run on one or more channels](#).

Deploy a Node.js bot to Azure from Visual Studio

10/4/2017 • 4 min to read • [Edit Online](#)

When you build a bot with Visual Studio, you can take advantage of its built-in publishing capability. This tutorial shows you how to deploy a Node.js bot to Azure directly from Visual Studio 2017.

Prerequisites

- You must have a Microsoft Azure subscription. If you do not already have a subscription, you can register for a [free trial](#).
- Install Visual Studio 2017. If you do not already have Visual Studio, you can download [Visual Studio 2017 Community](#) for free.
- Install **Azure development**. You can choose to have **Azure development** installed while installing Visual Studio or you can add it if you already have Visual Studio installed. You can add it by going to your computer's **Uninstall or change a program**, find **Microsoft Visual Studio 2017** and choose to **Modify** it. When the **Workloads** window shows up, check the box for **Azure development** and click **Modify** to install this resource.

Application settings and Messaging endpoint

Verify application settings

For your bot to function properly in the cloud, you must ensure that its application settings are correct. If you've already [registered](#) your bot with the Bot Framework, update the Microsoft App Id and Microsoft App Password values in your application's configuration settings as part of the deployment process. Specify the **app ID** and **password** values that were generated for your bot during registration.

TIP

If you're using the Bot Builder SDK for Node.js, set the following environment variables:

- `MICROSOFT_APP_ID`
- `MICROSOFT_APP_PASSWORD`

If you're using the Bot Builder SDK for .NET, set the following key values in the `web.config` file:

- `MicrosoftAppId`
- `MicrosoftAppPassword`

If you have not yet registered your bot with the Bot Framework (and therefore do not yet have an **app ID** and **password**), you can deploy your bot with temporary placeholder values for these settings. Then later, after you register your bot, update your deployed application's settings with the **app ID** and **password** values that were generated for your bot during registration.

Verify Messaging endpoint

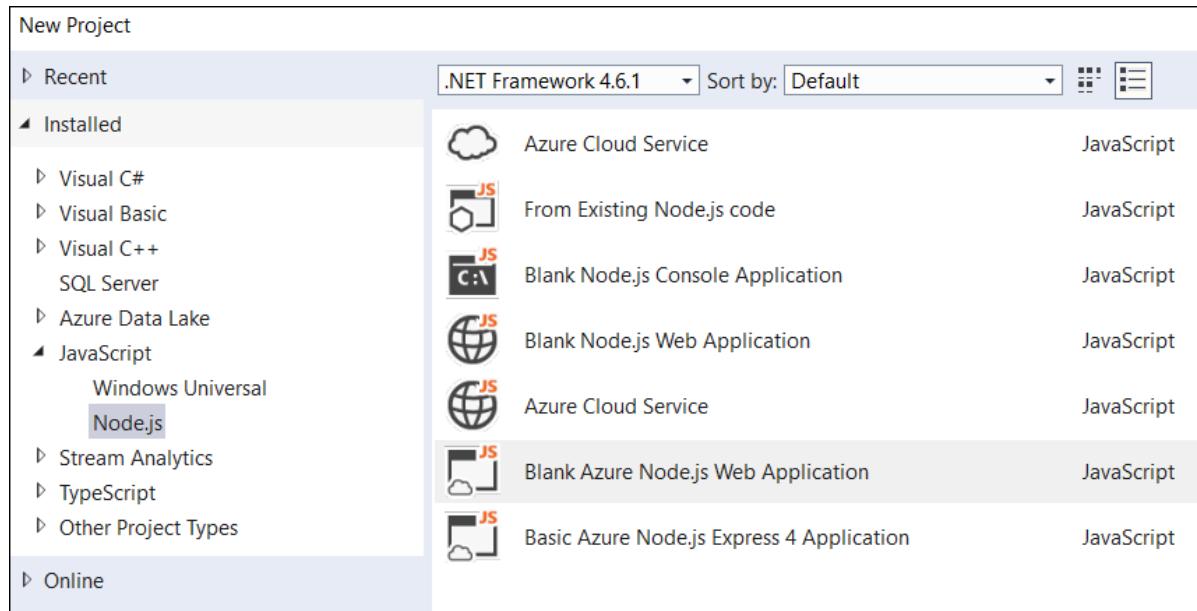
Your deployed bot must have an **Messaging endpoint** that can receive messages from the Bot Framework Connector Service.

NOTE

When you deploy your bot to Azure, SSL will automatically be configured for your application, thereby enabling the **Messaging endpoint** that the Bot Framework requires. If you deploy to another cloud service, be sure to verify that your application is configured for SSL so that the bot will have a **Messaging endpoint**.

Step 1: Launch the Microsoft Azure Publishing Wizard in Visual Studio

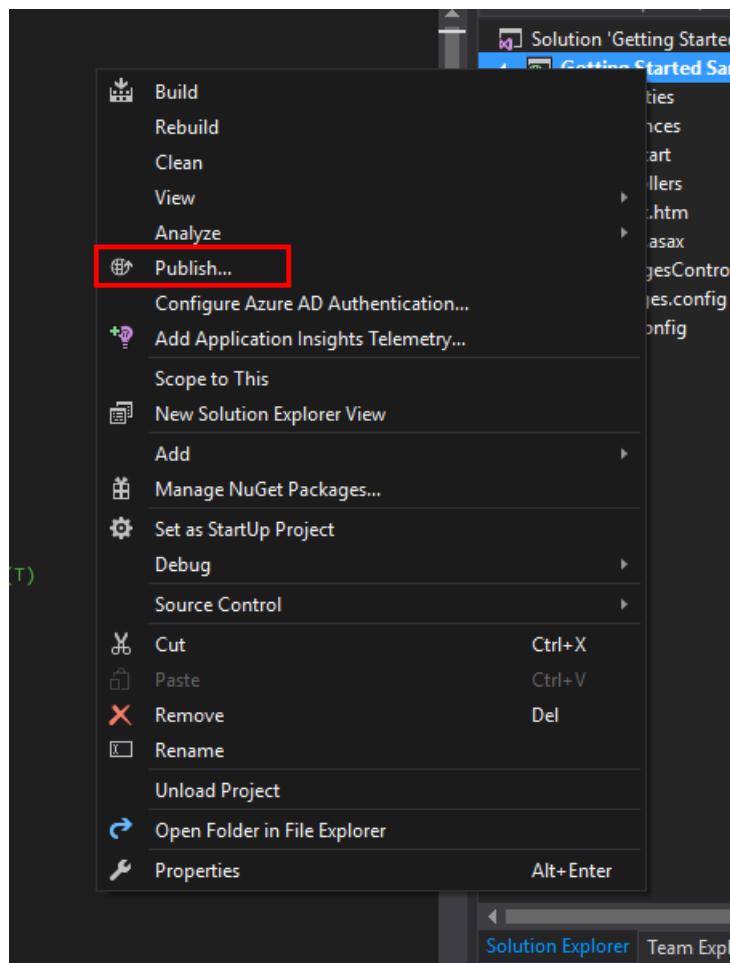
1. Create a new **Blank Azure Node.js Web Application** project in Visual Studio. This process only works with this project type.



2. Update your **package.json** file to add these two dependencies to the project.

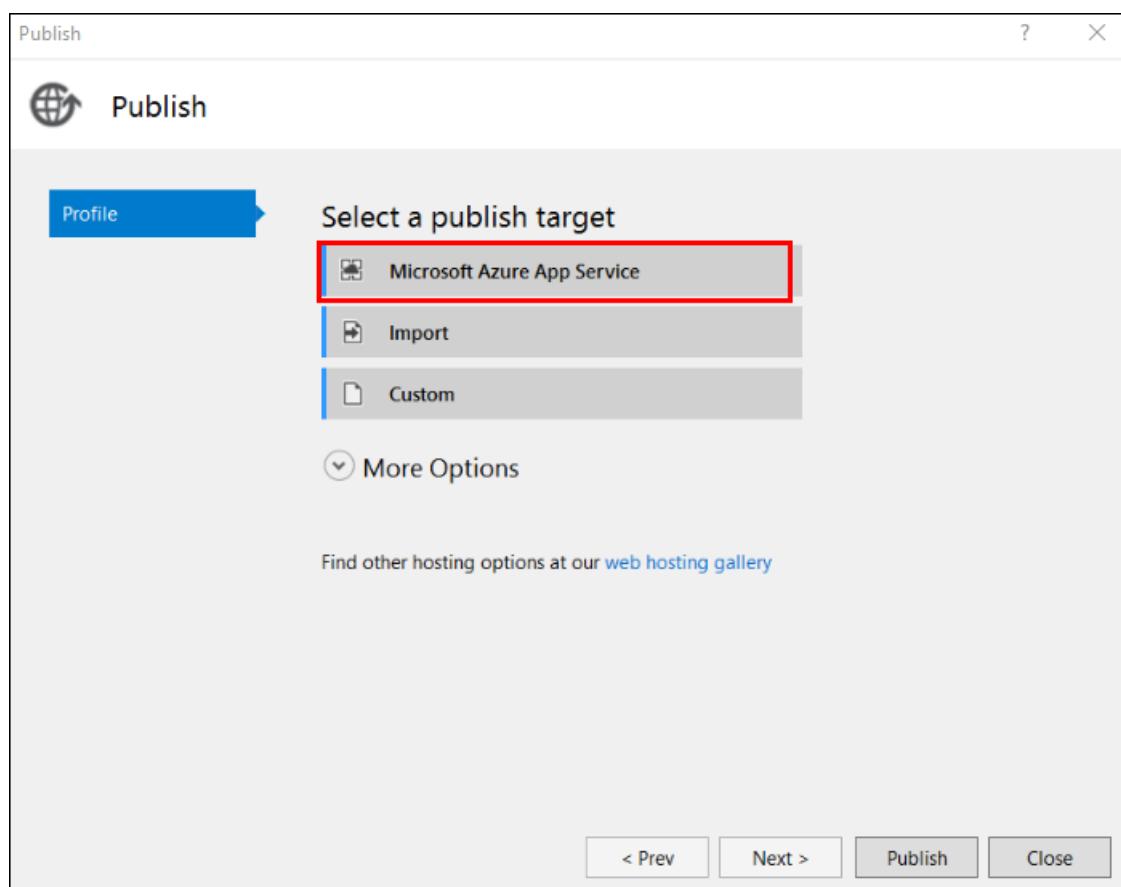
```
"dependencies": {  
  "botbuilder": "^3.7.0",  
  "restify": "^4.3.0"  
}
```

3. Build the project. Then, in Solution Explorer, right-click **npm** and click **Install Missing npm Packages**. This will install the two dependencies you added in the previous step.
4. Add your bot code to the project as necessary.
5. In Solution Explorer, right-click on the project name and select **Publish**. This starts the Microsoft Azure publishing wizard.

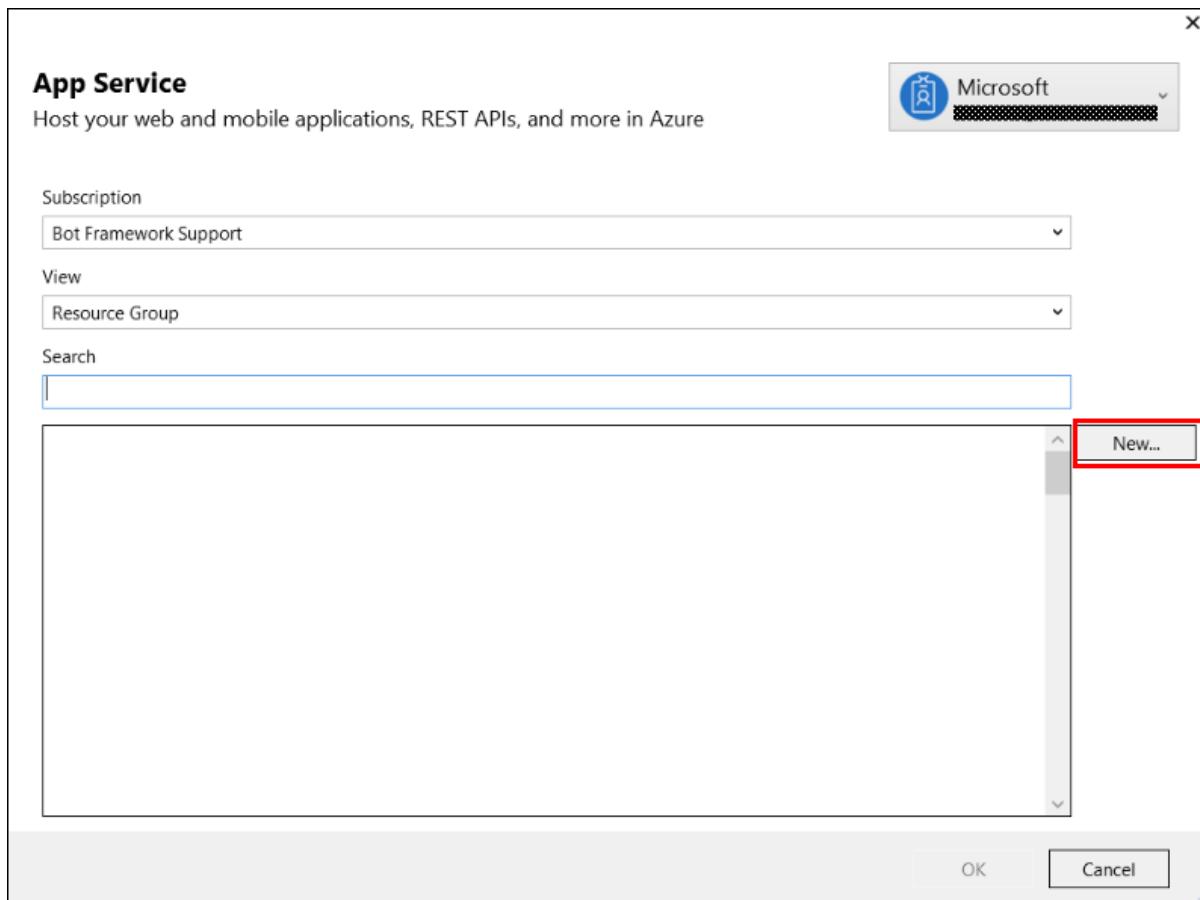


Step 2: Publish to Azure using the Azure Publishing Wizard

1. Select **Microsoft Azure App Service** as the project type. Choosing this option will open the **App Service** window.



2. Click **New** to create your **App Service**.



3. Fill in the form as appropriate for your implementation.

- Click **New** to create a new **Resource Group** or **App Service Plan** if necessary.
- Click **Change Type** and change the service's type to **Web App**.
- Click **Create** to create your app service.

Create App Service
Host your web and mobile applications, REST APIs, and more in Azure

Hosting App Name **Change Type**

Services

Subscription: Bot Framework Support
Resource Group: nodejs (westus) **New...**
App Service Plan: NodejsWebApp3345245451Plan (S1, West US) **New...**

Clicking the Create button will create the following Azure resources
[Explore additional Azure services](#)
App Service - NorthwindCafe20170929104930

If you have removed your spending limit or you are using Pay as You Go, there may be monetary impact if you provision additional resources.
[Learn More](#)

Export... **Create** **Cancel**

- Validate connection and copy the **Destination URL** value to the clipboard. You will need this value later to test the connection to the bot using the Emulator.

Publish

Profile **NorthwindCafe20170929104930 - Web Deploy**

Connection

Profile: **NorthwindCafe20170929104930 - Web Deploy**

Publish method: **Web Deploy**

Server: northwindcafe20170929104930.scm.azurewebsites.net:443

Site name: NorthwindCafe20170929104930

User name: \$NorthwindCafe20170929104930

Password: **XXXXXXXXXXXXXX**

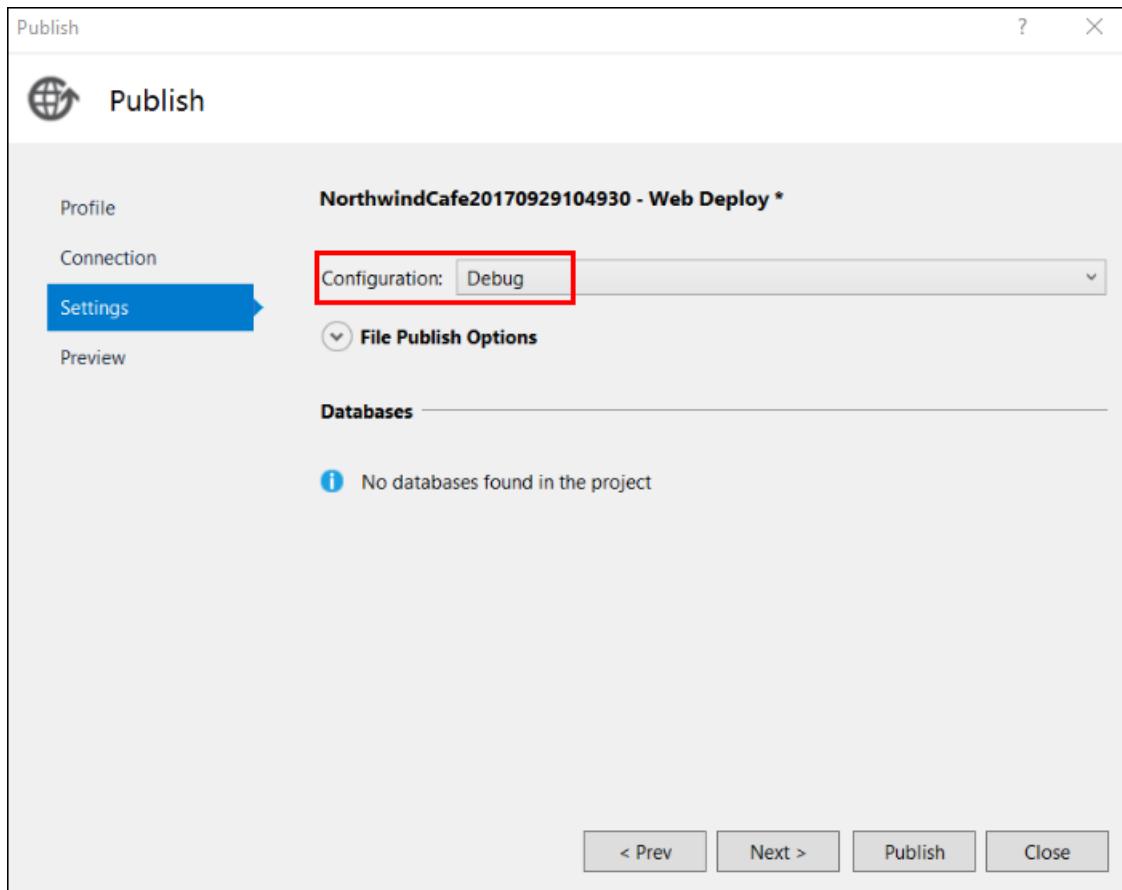
Save password

Destination URL: **http://northwindcafe20170929104930.azurewebsites.net**

Validate Connection

< Prev Next > **Publish** Close

- Click **Settings**. By default, your bot will be published in a **Release** configuration. If you want to debug your bot, in **Settings**, change **Configuration** to **Debug**.



6. Click **Publish** to deploy your bot to Azure.

Step 3: Test the connection to your bot

Verify the deployment of your bot by using the [Bot Framework Emulator](#).

Enter the bot's **Messaging endpoint** into the address bar of the Emulator. If you built your bot with the Bot Builder SDK, the endpoint should end with `/api/messages`.

Example: `https://<appname>.azurewebsites.net/api/messages`

Next steps

After you have deployed your bot to the cloud and verified that the deployment was successful by testing the bot using the Bot Framework Emulator, the next step in the bot publication process will depend upon whether or not you've already registered your bot with the Bot Framework.

If you have already registered your bot with the Bot Framework:

1. Return to the [Bot Framework Portal](#) and update your bot's registration data to specify the **Messaging endpoint** for the bot.
2. [Configure the bot to run on one or more channels.](#)

If you have not yet registered your bot with the Bot Framework:

1. [Register your bot with the Bot Framework.](#)
2. [Update the Microsoft App Id and Microsoft App Password values](#) in your deployed application's configuration settings to specify the **app ID** and **password** values that were generated for your bot during the registration process.
3. [Configure the bot to run on one or more channels.](#)

Register a bot with the Bot Framework

10/12/2017 • 3 min to read • [Edit Online](#)

Before others can use your bot, you must register it with the Bot Framework. Registration is a simple process. You are prompted to provide some information about your bot and then the portal generates the app ID and password that your bot will use to authenticate with the Bot Framework.

NOTE

Bots created with the Azure Bot Service are automatically registered as part of the creation process.

Register your bot

To register your bot, you must sign in to the [Bot Framework Portal](#). After you sign in, click **My bots**, then click **Create a bot**, then select **Register an existing bot built using Bot Builder SDK.**, and finally, click **Ok**. Then complete the following steps.

Complete the Bot profile section of the form.

1. Upload an icon that will represent your bot in the conversation.
2. Provide your bot's **Display Name**. When users search for this bot, this is the name that will appear in the search results.
3. Provide your bot's **Handle**. This value will be used in the URL for your bot and cannot be changed after registration.
4. Provide a **Description** of your bot. This is the description that will appear in search results, so it should accurately describe what the bot does.

Complete the Configuration section of the form.

1. Provide your bot's **HTTPS** messaging endpoint. This is the endpoint where your bot will receive HTTP POST messages from Bot Connector. If you built your bot by using the Bot Builder SDK, the endpoint should end with `/api/messages`.
 - If you have already deployed your bot to the cloud, specify the endpoint generated from that deployment.
 - If you have not yet deployed your bot to the cloud, leave the endpoint blank for now. You will return to the Bot Framework Portal later and specify the endpoint after you've deployed your bot.
2. Click **Create Microsoft App ID and password**.
 - On the next page, click **Generate an app password to continue**.
 - Copy and securely store the password that is shown, and then click **Ok**.
 - Click **Finish and go back to Bot Framework**.
 - Back in the Bot Framework Portal, the **App ID** field is now populated.

Complete the Analytics section of the form.

To enable [analytics](#) for your bot, provide the **AppInsights Instrumentation key**, **AppInsights API key**, and **AppInsights Application ID** from the corresponding [Azure Application Insights](#) resource that you've created to monitor your bot.

NOTE

If you have not yet created an Azure Application Insights resource to monitor your bot, leave these fields blank. To enable [analytics](#) for your bot in the future, return to the Bot Framework Portal and populate these fields at that time.

Complete the Admin section of the form.

1. Specify the email address(es) for the **Owner(s)** of the bot.
2. Check to indicate that you have read and agree to the [Terms of Use](#), [Privacy Statement](#), and [Code of Conduct](#).
3. Click **Register** to complete the registration process.

Update application configuration settings

After you've registered your bot, update the Microsoft App Id and Microsoft App Password values in your application's configuration settings to specify the **app ID** and **password** values that were generated for your bot during the registration process.

TIP

If you're using the Bot Builder SDK for Node.js, set the following environment variables:

- MICROSOFT_APP_ID
- MICROSOFT_APP_PASSWORD

If you're using the Bot Builder SDK for .NET, set the following key values in the web.config file:

- MicrosoftAppId
- MicrosoftAppPassword

Update or delete registration

To update or delete the bot's registration data:

1. Sign in to the [Bot Framework Portal](#).
2. Click **My Bots**.
3. Select the bot that you want to configure and click **Settings**.
 - To generate a new password, scroll down to the **Configuration** section and click **Manage Microsoft App ID and password**.
 - To update a bot's settings, change field value(s) as desired, then scroll down to the **Admin** section and click **Save changes**.
 - To delete a bot, scroll down to the **Admin** section and click **Delete bot**.

Next steps

After you have registered your bot with the Bot Framework, the next step in the bot publication process will depend upon whether or not you've already deployed your bot to the cloud.

If you have not yet deployed your bot to the cloud:

1. Deploy your bot to the cloud by following the instructions found in [Deploy a bot to the cloud](#).
2. Return to the [Bot Framework Portal](#) and [update your bot's registration data](#) to specify the **HTTPS** endpoint for the bot.

3. [Configure the bot to run on one or more channels.](#)

If you have already deployed your bot to the cloud:

1. Update the Microsoft App Id and Microsoft App Password values in your deployed application's configuration settings to specify the **app ID** and **password** values that were generated for your bot during the registration process, as described [here](#).

2. [Configure the bot to run on one or more channels.](#)

Bot analytics

8/7/2017 • 2 min to read • [Edit Online](#)

Analytics is an extension of [Application Insights](#). Application Insights provides **service-level** and instrumentation data like traffic, latency, and integrations. Analytics provides **conversation-level** reporting on user, message, and channel data.

View analytics for a bot

To access Analytics, open the bot in the developer portal and click **Analytics**.

Too much data? [Enable and configure sampling](#) to reduce telemetry traffic and storage while maintaining statistically correct analysis.

Specify channel

Choose which channels appear in the graphs below. Note that if a bot is not enabled on a channel, there will be no data from that channel.

The screenshot shows a list of channels with checkboxes. The checked channels are Bing, Facebook Messenger, Skype, Twilio (SMS), Cortana, GroupMe, Skype for Business, Web Chat, Direct Line, Kik, Slack, WeChat, Email, Microsoft Teams, and Telegram. A date range selector at the top right shows "4/3/2017 - 5/4/2017".

Bing	Cortana	Direct Line	Email
<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
Facebook Messenger	GroupMe	Kik	Microsoft Teams
<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
Skype	Skype for Business	Slack	Telegram
<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
Twilio (SMS)	Web Chat	WeChat	

- Check the check box to include a channel in the chart.
- Clear the check box to remove a channel from the chart.

Specify time period

Analysis is available for the past 90 days only. Data collection began when Application Insights was enabled.

The screenshot shows a dropdown menu with the date range "4/3/2017 - 5/4/2017" and a list of time increments: Last Hour, Last Day, Last Week, Last Month (selected), and Last 90 days.

4/3/2017 - 5/4/2017

Last Hour Last Day Last Week Last Month Last 90 days

Click the drop-down menu and then click the amount of time the graphs should display. Note that changing the overall time frame will cause the time increment (X-axis) on the graphs to change accordingly.

Grand totals

The total number of active users and messages sent and received during the specified time frame. Dashes -- indicate no activity.

Retention

Retention tracks how many users who sent one message came back later and sent another one. The chart is a rolling 10-day window; the results are not affected by changing the time frame.

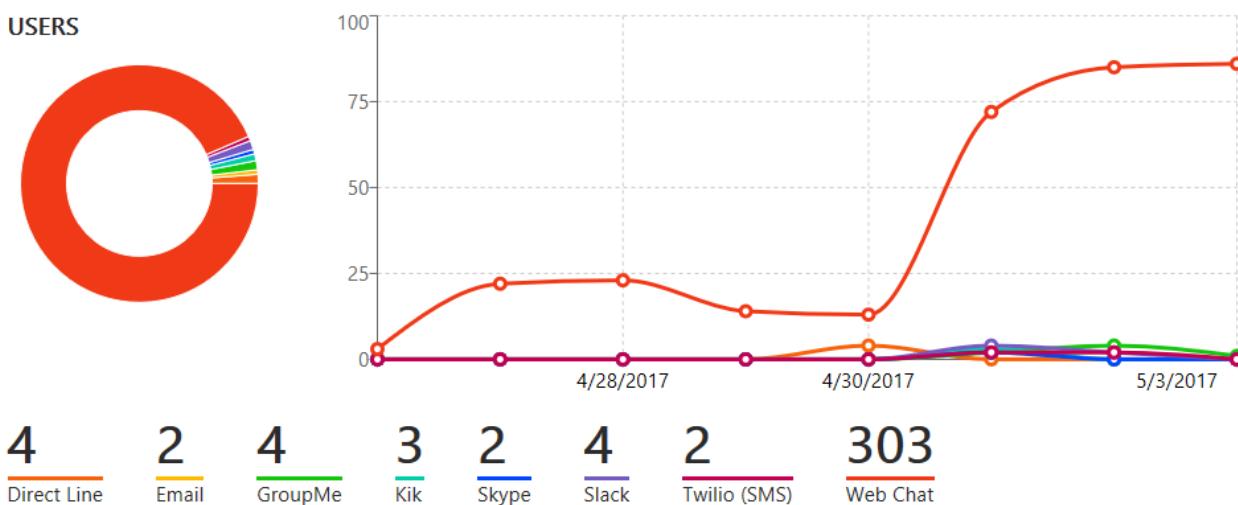
RETENTION - % USERS WHO MESSED AGAIN (LAST 10 DAYS)

Date	Users	Days later									
		1	2	3	4	5	6	7	8	9	10
4/23/2017	0	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
4/24/2017	13	31%	8%	8%	8%	8%	8%	8%	15%	8%	
4/25/2017	43	2%	2%	2%	5%	5%	2%	5%	2%		
4/26/2017	27	4%	4%	4%	4%	4%	4%	4%			
4/27/2017	37	3%	3%	3%	3%	3%	3%				
4/28/2017	22	5%	5%	5%	5%	5%					
4/29/2017	23	9%	9%	9%	9%						
4/30/2017	14	7%	7%	7%							
5/1/2017	17	18%	12%								
5/2/2017	87	9%									

Note that the most recent possible date is two days ago; a user sent messages the day before yesterday and then *returned* yesterday.

User

The Users graph tracks how many users accessed the bot using each channel during the specified time frame.



- The percentage chart shows what percentage of users used each channel.
- The line graph indicates how many users were accessing the bot at a certain time.
- The legend for the line graph indicates which color represents which channel and the includes the total number of users during the specified time period.

Messages

The Message graph tracks how many messages were sent and received using which channel during the specified time frame.



124 Direct Line **4** Email **50** GroupMe **24** Kik **2** Skype **19** Slack **14** Twilio (SMS) **1.3k** Web Chat

- The percentage chart shows what percentage of messages were communicated over each channel.
- The line graph indicates how many messages were sent and received over the specified time frame.
- The legend for the line graph indicates which line color represents each channel and the total number of messages sent and received on that channel during the specified time period.

Enable analytics

Analytics are not available until Application Insights has been enabled and configured. Application Insights will begin collecting data as soon as it is enabled. For example, if Application Insights was enabled a week ago for a six-month-old bot, it will have collected one week of data.

NOTE

Analytics requires both an Azure subscription and Application Insights [resource](#). To access Application Insights, open the bot in the [Bot Framework Portal](#) and click **Settings**.

- Create an Application Insights [resource](#).
- Open the bot in the dashboard. Click **Settings** and scroll down to the **Analytics** section.
- Enter the information to connect the bot to Application Insights. All fields are required.

Analytics

Enable Analytics for your bot via Azure Application Insights.

AppInsights Instrumentation key [?](#)

AppInsights API key [?](#)

AppInsights Application ID [?](#)

AppInsights Instrumentation Key

To find this value, open Application Insights and navigate to **Configure > Properties**.

AppInsights API key

Provide an Azure App Insights API key. Learn how to [generate a new API key](#). Only **Read** permission is required.

AppInsights Application ID

To find this value, open Application Insights and navigate to **Configure > API Access**.

For more information on how to locate these values, see [Application Insights keys](#).

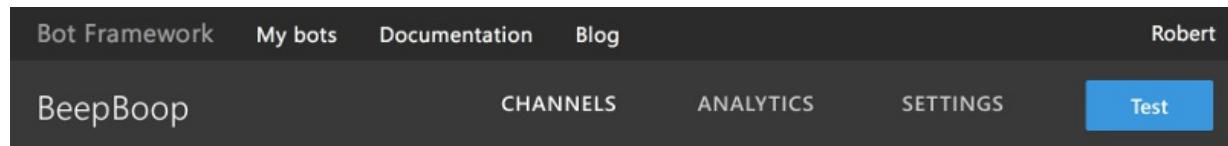
Additional resources

- [Application Insights keys](#)

Connect a bot to channels

8/7/2017 • 2 min to read • [Edit Online](#)

A channel is the connection between the Bot Framework and communication apps. You configure a bot to connect to the channels you want it to be available on. For example, a business bot connected to the Bing channel will show up in Bing search results and people can interact with it on Bing. Connecting to channels is quick and easy in the [Bot Framework Portal](#).

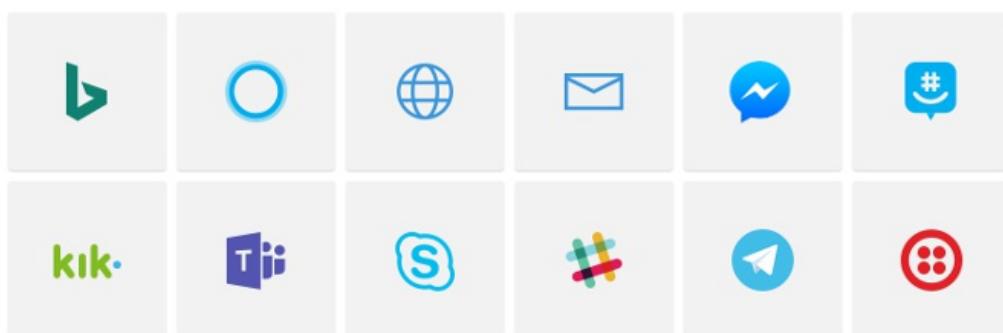


Connect to channels

Name	Health	Published	
 Skype	Running	--	Edit 
 Web Chat	Running	--	Edit 

[Get bot embed codes](#)

Add a channel



The channels include many popular services, such as [Bing](#), [Cortana](#), [Facebook Messenger](#), [Kik](#), and [Slack](#), as well as several others. [Skype](#) and [Web Chat](#) are pre-configured for you.

You configure your bot to run on channels after you have [registered](#) your bot with the Bot Framework and [deployed](#) it to the cloud.

Get started

For most channels, you must provide channel configuration information to run your bot on the channel. Most channels require that your bot have an account on the channel, and others, like Facebook Messenger, require your bot to have an application registered with the channel also.

To configure your bot to connect to a channel, complete the following steps:

1. Sign in to the [Bot Framework Portal](#).
2. Click **My bots**.
3. Select the bot that you want to configure.
4. Click the **Channels** tab.
5. Under **Add channel** on the bot dashboard, click the channel to add.

After you've configured the channel, users on that channel can start using your bot.

Publish a bot

The publishing process is different for each channel. If the channel requires a review, ensure that the bot meets the [review guidelines](#) before submitting it for publishing.

Bing

Bots are published to Bing from the [configuration page](#). Publishing a bot submits it for review. After approval, users can find the bot in Bing search results and use Bing, or any other supported channel, to interact with the bot.

Publishing to the Bing channel makes the bot discoverable by the widest possible audience. However, not all bots should be easily discoverable. For example, a bot designed for use by company employees should not be made generally available. A link to the bot can be privately distributed instead.

Cortana

Bots are published to Cortana from the [dashboard](#) and are used to power Cortana skills. Publishing a bot submits it for review. Cortana skills can be deployed for your own use, deployed to a small group, or published to the world.

Skype

Bots are published to Skype from the [configuration page](#). Publishing a bot submits it for review. Before review, the bot is limited to 100 contacts. Approved bots do not have limited contacts and you may opt to have the bot included in the Skype bot directory.

Skype for Business

Skype for Business bots are registered with a [Skype for Business Online tenant](#) by a Tenant Administrator.

TIP

To view the status of a review, open the bot in the [Bot Framework Portal](#) and click **Channels**. If the bot is not approved, the result will link to the reason why. After making the required changes, resubmit the bot for review.

Preview bot features with the Channel Inspector

9/25/2017 • 4 min to read • [Edit Online](#)

The Bot Framework enables you to create bots with a variety of features such as text, buttons, images, rich cards displayed in carousel or list format, and more. However, each channel ultimately controls how features are rendered by its messaging clients.

Even when a feature is supported by multiple channels, each channel may render the feature in a slightly different way. In cases where a message contains feature(s) that a channel does not natively support, the channel may attempt to down-render message contents as text or as a static image, which can significantly impact the message's appearance on the client. In some cases, a channel may not support a particular feature at all. For example, GroupMe clients cannot display a typing indicator.

The Channel Inspector

The [Channel Inspector](#) is created to give you a preview on how various Bot Framework features look and work on different channels. By understanding how features are rendered by various channels, you'll be able to design your bot to deliver an exceptional user experience on the channels where it communicates. The Channel Inspector also provides a great way to learn about and visually explore Bot Framework features.

NOTE

Rich cards are a developing standard for bot information exchange to ensure consistent display across multiple channels. See the [.NET](#) or [Node.js](#) documentation for more information about rich cards.

Text formatting

Text formatting can enhance your text messages visually. Besides **plain** text, your bot can send text messages using **markdown** or **xml** formatting to channels that support them. The following tables list some of the most commonly used text formatting in **markdown** and **xml**. Each channel may support fewer or more text formatting than what is listed here. You can check the [Channel Inspector](#) to see if a feature you want to use is supported on a channel that you target.

Markdown text format

These styles may be supported when `textFormat` is set to **markdown**:

STYLE	MARKDOWN	EXAMPLE
bold	**text**	text
italic	<i>*text*</i>	<i>text</i>
header (1-5)	# H1	# H1
strikethrough	~~text~~	text
horizontal rule	----	----

STYLE	MARKDOWN	EXAMPLE
unordered list	* text	• text
ordered list	1. text	1. text
preformatted text	`text`	<code>text</code>
blockquote	> text	<div style="border: 1px solid #ccc; padding: 5px;">text</div>
hyperlink	[bing](http://www.bing.com)	bing
image link	![duck](http://aka.ms/Fo983c)	

NOTE

HTML tags in **Markdown** are not supported in Microsoft Bot Framework Web Chat channels. If you need to use HTML tags in your **Markdown**, you can render them in a [Direct Line](#) channel that supports them. Alternatively, you can use the HTML tags below by setting the `textFormat` to **xml** and connect your bot to Skype channel.

XML text format

These styles may be supported when `textFormat` is set to **xml**:

STYLE	XML	EXAMPLE
bold	text	text
italic	<i>text</i>	<i>text</i>
underline	<u>text</u>	<u>text</u>
strikethrough	<s>text</s>	text
hyperlink	bing	bing
paragraph	<p>text</p>	text
line break	 	line 1 line 2
horizontal rule	<hr/>	

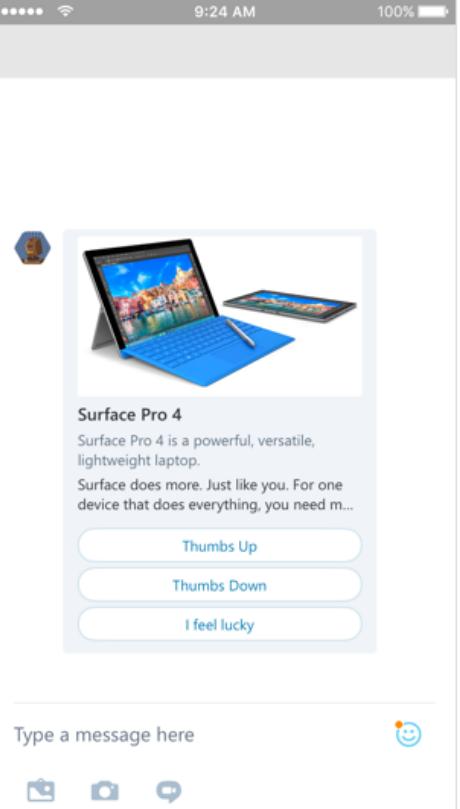
STYLE	XML	EXAMPLE
header (1-4)	<h1>text</h1>	Heading 1
code	<code>code block</code>	code block
image		

NOTE

The `textFormat` XML is supported only by the Skype channel.

Preview features across various channels

To see how a channel renders a particular feature, go to the [Channel Inspector](#), select the channel from **Channel** list and the feature from the **Feature** list. For example, to see how Skype renders a Hero Card, set **Channel** to *Skype* and **Feature** to *HeroCard*.



The screenshot shows a mobile-style interface with a navigation bar at the top displaying signal strength, time (9:24 AM), and battery level (100%). Below the navigation bar is a large image of a Microsoft Surface Pro 4 laptop. To the left of the image is a small user profile icon. Below the image, the text "Surface Pro 4" is displayed, followed by a description: "Surface Pro 4 is a powerful, versatile, lightweight laptop. Surface does more. Just like you. For one device that does everything, you need m...". At the bottom of the card are three interactive buttons labeled "Thumbs Up", "Thumbs Down", and "I feel lucky".

Channel:

Feature:

HeroCard

A multipurpose formatted attachment; it typically hosts a single image, optional buttons, and a 'tap action', along with text content to display on the card.

[Bot Framework Documentation](#)

Notes

Appearance	Native Control.
Images	Min 1 image per card

[See something wrong? Send Feedback](#)

The Channel Inspector shows a preview of the selected feature as it will be rendered by the specified channel. The **Notes** section conveys important information about message limitations and/or display changes. For example, some types of rich cards support only one image and some features may be down-rendered on certain channels.

NOTE

The Channel Inspector currently supports these channels:

- Bing
- Cortana
- Email
- Facebook
- GroupMe
- Kik
- Skype
- Skype for Business
- Slack
- SMS
- Microsoft Teams
- Telegram
- WeChat
- WebChat

Additional channels may be added in the future.

Features that can be previewed

The Channel Inspector currently allows you to preview the following features.

FEATURE	DESCRIPTION
Adaptive Cards	A card that can contain any combination of text, speech, images, buttons, and input fields.
Buttons	Buttons that the user can click. Buttons appear on the conversation canvas with the message they belong to.
Carousel	A compact, scrollable horizontal list of cards. For a vertical layout, use List.
ChannelData	A way to pass metadata to access channel-specific functionality beyond cards, text, and attachments.
Codesample	A multi-line, pre-formatted block of text designed to display computer code.
DirectMessage	A message sent to a single member of a group conversation.
Document	Send and receive non-media attachments.
Emoji	Display supported emoji.
GroupChat	Bot sends messages to participate in group conversations.
HeroCard	A formatted attachment typically containing a single large image, a tap action, and buttons (optional), along with descriptive text content.

FEATURE	DESCRIPTION
Images	Display of image attachments.
List Layout	A vertical list of cards. For a horizontal, scrollable layout, use Carousel.
Location	Share the user's physical location with the bot.
Markdown	Renders text formatted with Markdown.
Members	Shares the list of members in the conversation with the bot during a group chat.
Receipt Card	Display a receipt to the user.
Signin Card	Request the user to enter authentication credentials.
Suggested Actions	Actions presented as buttons that the user can tap to provide input.
Thumbnail Card	A formatted attachment containing a single small image (thumbnail), a tap action, and buttons (optional), along with descriptive text content.
Typing	Displays a typing indicator. This is helpful to inform the user that the bot is still functioning, but performing some action in the background.
Video	Displays video attachments and play controls.

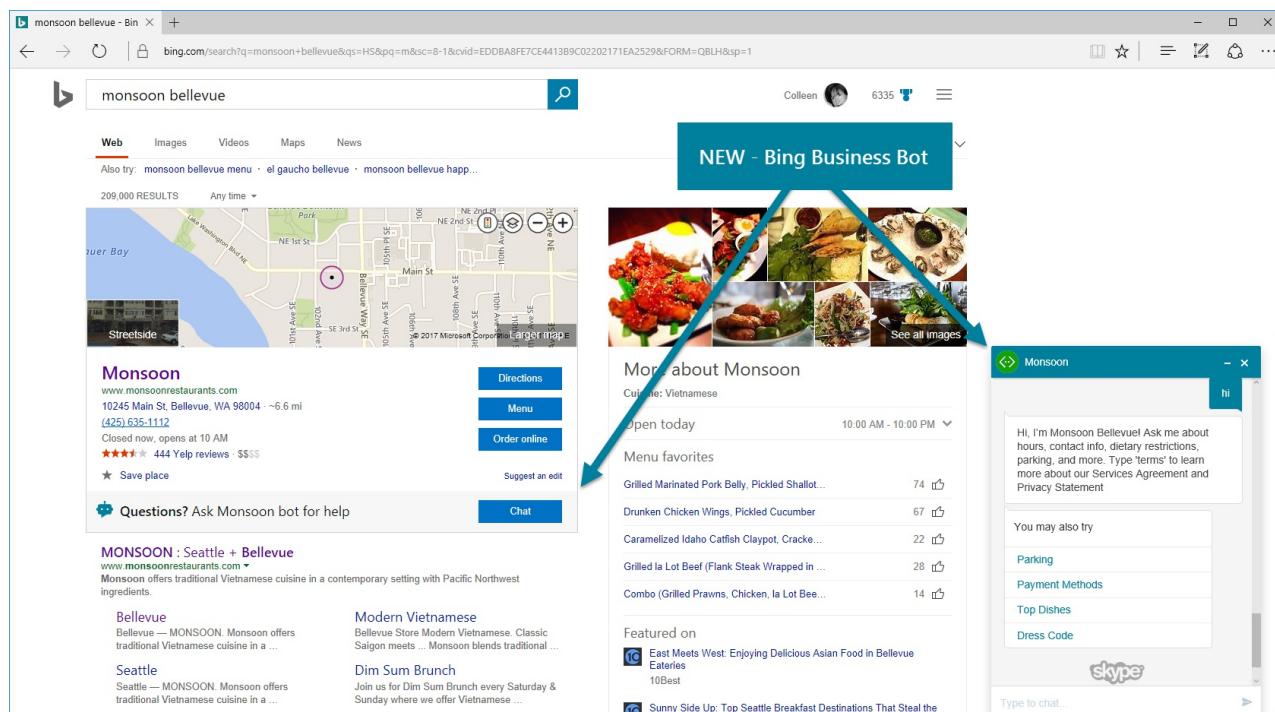
Additional resources

- [Channel Inspector](#)
- Rich cards in [Node.js](#) and [.NET](#)
- Media attachments in [Node.js](#) and [.NET](#)
- Suggested actions in [Node.js](#) and [.NET](#)

Connect a bot to Bing

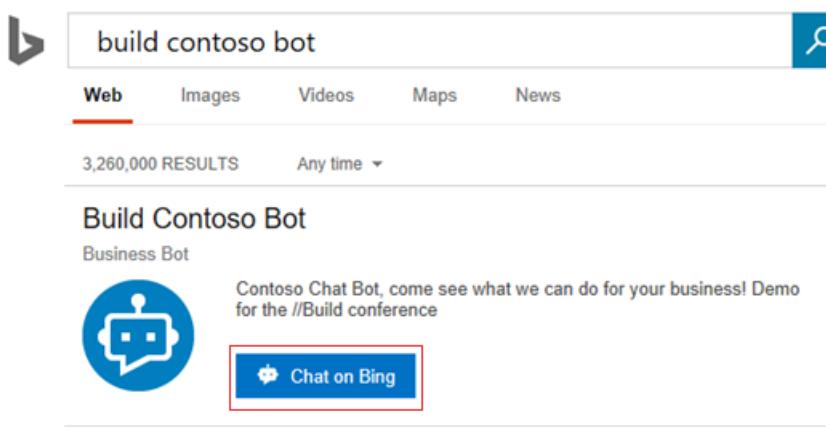
10/24/2017 • 3 min to read • [Edit Online](#)

Connecting your bot to the Bing channel allows people to discover and chat with your bot directly in the Bing search results page. You can build Bing bots using the [Bot Builder SDK](#) and connect them to the Bing channel. The Bing bot development experience allows you to publish your bots to Bing. People will also be able to chat with the bot directly on Bing.com.



To find a bot on Bing, go to [Bing.com](#) and enter a search query in the form of "*BotName* bot." For example, if the bot's name is "Contoso," enter "Contoso bot."

The bot will appear in Bing search results as a rich answer when the user queries specifically for the bot. Click **Chat on Bing** to launch the bot.



If a website has been associated with the bot, the link to the bot will appear **underneath** the website's Bing search result. Click the link to the bot to launch the bot.

Publish your bot on Bing

Publishing a bot on Bing is easy. Open the bot on the [Bot Framework Portal](#), click the Channels tab, and then click

Bot Framework My bots Documentation Blog Robert

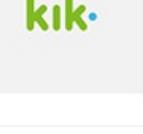
BeepBoop CHANNELS ANALYTICS SETTINGS Test

Connect to channels

Name	Health	Published	
 Skype	Running	--	Edit 
 Web Chat	Running	--	Edit 

[Get bot embed codes](#)

Add a channel
















All fields marked with an asterisk (*) are required. Bots must be [registered](#) on the Bot Framework before they can be connected to Bing.

NOTE

Be sure you read the [Review Guidelines](#), [Terms of Use](#) and [Code of Conduct](#) before publishing a bot on Bing.

Open the bot on the [Bot Framework Portal](#), click the **Channels** tab, and then click **Bing**.

* Display Name

* Long Description

* Bot Website

Display Name

Enter a name for this bot. This is the name that will appear in search results.

Long Description

Describe the bot's purpose and function. The bot must operate as described in its bot description.

Website

Link to a website with more information about this bot. There will be an additional verification step to link this bot to the site.

Bot category

Categories provide another way for users to find bots on the Bing Bot directory. Users can filter by category to find all "entertainment bots", "music bots", "news bots", and so on.

Bot category

* Bot Category

Select a category

Tags for your Bot

Select category

Choose the most appropriate category for this bot.

Tags for this bot

Enter appropriate tags to help users find this bot.

Publisher information

This is the contact information Microsoft will use to communicate with the bot's publisher.

Publisher information

* Publisher Name

* Publisher Email

* Publisher Phone

Publisher name

Enter the name of person to contact about this bot.

Publisher email

Enter an email address to use for bot-related communications.

Publisher phone

Enter a phone number to use for bot-related communications.

Privacy and terms of use

These are required for published bots.

Privacy and terms of use

* Privacy Statement URL

https://

* Terms of use URL

https://

Privacy Statement URL

If this bot handles users' personal data, provide a link to the applicable privacy policy. The [Code of Conduct](#) contains third party resource links to help create a privacy policy.

Terms of use URL

A link to the bot's Terms of Service is required. The [Terms of Use](#) contains sample terms to help create an appropriate Terms of Service document.

Submit for review and publish

Click **Submit for review**.

After submitting this bot for review, click the **Test on Bing** link to preview a sample of how the bot will appear on Bing. The bot logo will not appear in the sample, but it will appear once the bot is published.

NOTE

Do **not** distribute the test link. Its only purpose is to provide a preview before the bot is approved.

The review process takes a few business days and you may be contacted. After approval, users can find the bot in Bing search results and use Bing, or any other supported channel, to interact with the bot.

Publishing to the Bing channel makes the bot discoverable by the widest possible audience. However, not all bots should be easily discoverable. For example, a bot designed for use by company employees should not be made generally available. A link to the bot can be privately distributed instead.

TIP

To view the status of a review, open the bot in the [Bot Framework Portal](#) and click **Channels**. If the bot is not approved, the result will link to the reason why. After making the required changes, resubmit the bot for review.

A screenshot of a Bing search results page. The search bar at the top contains the query "contoso". Below the search bar, there are five tabs: "Web" (which is underlined in red), "Images", "Videos", "Maps", and "News". Under the "Web" tab, it says "575,000 RESULTS" and "Any time ▾". The first search result is for "Contoso" with the URL "www.contosoco.com". Below the URL, there is a snippet of text: "Contoso Co One Microsoft Way Redmond, WA 98052. Copyright © Contoso Co. All Rights Reserved . Home; About Us; Contact Us; Practice Areas ...". Underneath the snippet, there are three links: "Contact Us · Attorneys · Practice Areas". At the bottom of the result card, there is a button with a blue speech bubble icon and the text "Chat with Build Contoso Bot".

Connect a bot to Cortana

6/13/2017 • 3 min to read • [Edit Online](#)

Cortana is a speech-enabled channel that can send and receive voice messages in addition to textual conversation. A bot intended to connect to Cortana should be designed for speech as well as text. A Cortana *skill* is a bot that can be invoked using a Cortana client. Publishing a bot adds the bot to the list of available skills.

Open the bot on the [Bot Framework Portal](#), click the **Channels** tab, and then click **Cortana**.



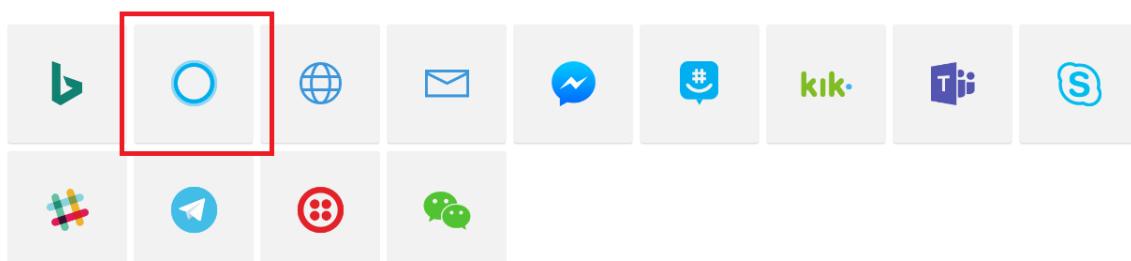
The screenshot shows the Bot Framework Portal interface. At the top, there are navigation links: Bot Framework, My bots, Documentation, and Blog. On the right, there's a user profile with the name "Jon". Below the navigation, the title "Dice Roller" is displayed. The main content area has tabs: CHANNELS (which is selected and highlighted in blue), ANALYTICS, SETTINGS, and a blue "Test" button. The CHANNELS tab shows a list of connected channels: Skype (Running) and Web Chat (Running). There are "Edit" buttons next to each entry.

Connect to channels

Name	Health	Published	
 Skype	Running	--	Edit
 Web Chat	Running	--	Edit

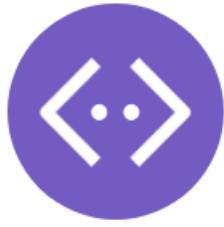
[Get bot embed codes](#)

Add a channel



General bot information

All fields marked with an asterisk (*) are required. Bots must be published on the Bot Framework before they can be connected to Cortana.



* Skill icon

[Upload custom icon](#)

32,000 byte maximum size, png only

* Display name

Dice Roller

* Invocation name

Roller

* Long description

This is a bot that rolls dice.

* Short description

Rolls dice.

Skill Icon

Select a custom icon to represent this bot. This icon is displayed in the Cortana canvas when this skill is invoked and anywhere skills are discoverable, such as the Microsoft store. The image must be a PNG, 60 x 60 pixels, and no larger than 30kb.

Display Name

The name of this skill as displayed to the user at the top of Cortana's visual UI.

Invocation Name

The name that Cortana will recognize and use to invoke this skill when spoken aloud by the user. See the [Invocation Name Guidelines](#) for more information on how to choose this phrase.

Skill description

Describe the skill. This is used where bots are discoverable, like the Microsoft store.

Short description

This summary will be used to describe the skill in Cortana's Notebook.

NOTE

For basic skills, this is all that is required. Click **Connect to Cortana** to complete the connection. For more advanced skills, add a connected account or configure access to user profile and contextual information.

Manage user identity

The default value is *Disabled*. If *Enabled*, the user will be required to log in before using this skill. Cortana can manage this skill's user identity and tokens across devices if an OAuth2 identity provider is added as a [connected service](#).

NOTE

Only **Auth Code Grant** authentication is supported. **Implicit Grant** is currently not supported.

Manage user identity through connected services

Cortana can manage your skill's user identity and tokens across devices if you add your oAuth2 identity provider as a connected service

To learn more about connected services, please refer to the [connected services developer docs](#).

Cortana should manage my user's identity



* Connected service icon



[Upload icon](#)

* Account name

* Client ID for third-party services

* Client secret/password for third party services

Space-separated list of scopes

List of scopes

* Authorization URL

<https://>

* Token URL

<https://>

* Client authentication scheme

Credentials in request body ▾

* Token options



This skill's connected service requires intranet access to authenticate users
(If you are unsure, leave this blank.)

FIELD	DESCRIPTION
Icon	The icon to display when the sign-in form is displayed.
Account Name	The name to be shown when the sign-in form is displayed.
Client ID for third-party services	The client id that Cortana needs as part of the OAuth flow.
Client secret/password for third party services	The client secret Cortana needs as part of the OAuth flow.
List of scopes	A list of scopes Cortana needs as part of the skill execution flow.
Authorization URL	The URI Cortana will call to authorize the user.
Token URL	The URI Cortana will call to fetch tokens.
Client authentication scheme	The mechanism by which Cortana will pass a bearer token.
Token option	How Cortana should obtain tokens. If unsure, select POST.
Intranet toggle	Select if this skill's connected service requires <i>intranet</i> access to authenticate users. (If unsure, leave this blank.)

Request user profile data

Cortana provides access to several different types of user profile information, which is useful in customizing the bot for different users.

Request user profile data

With the user's consent, bots can utilize user information from Cortana's notebook to customize interactions. Request user profile information below.

Data	Friendly Name
User.SemanticLocation.Current ▼ CurrentLocation Remove	
Request data	

1. Click **Add a user profile request**.
2. Click **Select Data** and then select the user profile information from the drop-down list.
3. Add a **Friendly name** to identify this information in the `UserInfo` entity.
4. Click **Remove** to remove this request for data or click **Add a user profile request** again to define another request.

Click **Save**. The bot will be deployed automatically as a Cortana skill.

Enable speech recognition priming

If your bot uses a Language Understanding Intelligent Service (LUIS) app, register the LUIS application ID.

Click the **Settings** tab and then under **Configuration**, enter the LUIS application ID in the **Speech recognition priming with LUIS** text box. This helps your bot recognize spoken utterances that are defined in your LUIS model.

Speech recognition priming with LUIS

In order to use speech recognition with your LUIS-powered bot, please select the LUIS applications referenced in your bot.

Did not find any LUIS apps in your account

Add

Next steps

- [Cortana Skills Kit](#)
- [Enable debugging](#)
- [Publish a Cortana skill](#)

Connect a bot to Skype for Business

6/13/2017 • 1 min to read • [Edit Online](#)

Skype for Business Online keeps you connected with co-workers and business partners through instant messaging, phone, and video calls. Extend this functionality by building bots that users can discover and interact with through the Skype for Business interface.

NOTE

The Skype for Business Bot Framework channel is currently supported for Skype for Business Online. Skype for Business Server 2015 is not supported.

Enable the channel

Open the bot on the [Bot Framework Portal](#), click the **Channels** tab, and then click **Skype for Business**. The bot is now enabled.

Registering the bot with Skype for Business Online is performed by a **Tenant Administrator** of that Skype for Business tenant.

Next steps

- [Register a bot with Skype for Business](#)

Connect a bot to Direct Line

7/26/2017 • 1 min to read • [Edit Online](#)

You can enable your own client application to communicate with your bot by using the Direct Line channel.

Add the Direct Line channel

To add the Direct Line channel, open the bot on the [Bot Framework Portal](#), click the **Channels** tab, and then click **Direct Line**.



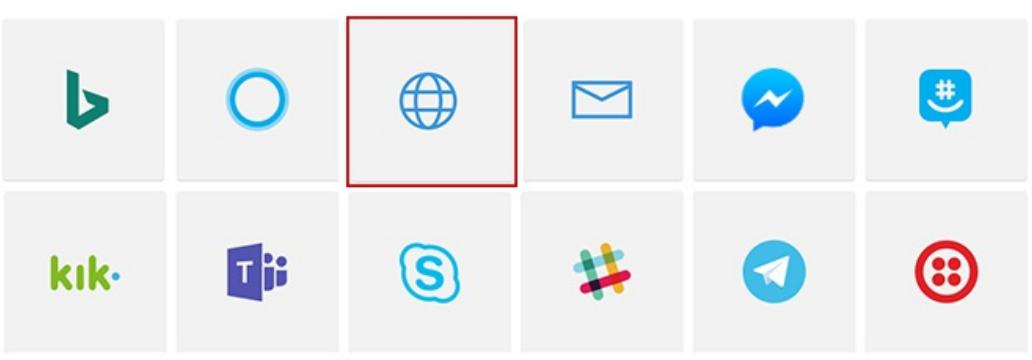
The screenshot shows the Bot Framework Portal interface. At the top, there are navigation links: Bot Framework, My bots, Documentation, and Blog. On the right, it shows the user's name, Robert. Below the navigation, the bot name BeepBoop is displayed, along with tabs for CHANNELS, ANALYTICS, and SETTINGS. A blue button labeled 'Test' is visible. The main content area is titled 'Connect to channels'. It lists two channels: 'Skype' (Running, Edit) and 'Web Chat' (Running, Edit). A link 'Get bot embed codes' is present. Below this, a section titled 'Add a channel' shows a grid of icons for various platforms: Bing, Cortana, Direct Line (highlighted with a red box), Email, Facebook, Google Hangouts, Kik, Microsoft Teams, Skype, Slack, Telegram, and Twilio. The Direct Line icon is the third icon in the top row, third column.

Connect to channels

Name	Health	Published	
Skype	Running	--	Edit
Web Chat	Running	--	Edit

[Get bot embed codes](#)

Add a channel



The screenshot shows a grid of icons representing different communication channels. The icons include: Bing, Cortana, Direct Line (highlighted with a red box), Email, Facebook, Google Hangouts, Kik, Microsoft Teams, Skype, Slack, Telegram, and Twilio. The Direct Line icon is the third icon in the top row, third column.

Add new site

Next, add a new site that represents the client application that you want to connect to your bot. Click **Add new site**, enter a name for your site, and click **Done**.



Configure Direct Line



+ Add new site

Manage secret keys

When your site is created, the Bot Framework generates secret keys that your client application can use to [authenticate](#) the Direct Line API requests that it issues to communicate with your bot. To view a key in plain text, click **Show** for the corresponding key.



Configure Direct Line



+ Add new site

Default Site

Disable |

Default Site

Secret keys

XXXXXXXXXXXXXXXXXXXXXXXXXXXX

[Show](#) | [Regenerate](#)

XXXXXXXXXXXXXXXXXXXXXXXXXXXX

[Show](#) | [Regenerate](#)

Version

Select which versions of the Direct Line protocol are enabled on this site. More information about these versions can be found in the [Direct Line reference documentation](#).

- 1.1
- 3.0 [PREVIEW]
- High-speed storage [PREVIEW]

[Done](#)

Copy and securely store the key that is shown. Then use the key to [authenticate](#) the Direct Line API requests that your client issues to communicate with your bot, or alternatively, use the Direct Line API to [exchange the key for a token](#) that your client can use to authenticate its subsequent requests within the scope of a single conversation.



Configure Direct Line



+ Add new site Default Site Disable |

Default Site

Secret keys

B-QmrgitzCY.cwA.sko.C7XO8ROjEKmRNxFzsVDkaKVhQA41HO74Lmgf-

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

Version
Select which versions of the Direct Line protocol are enabled on this site. More information about these versions can be found in the [Direct Line reference documentation](#).

1.1
 3.0 [PREVIEW]
 High-speed storage [PREVIEW]

Done

Configure settings

Finally, configure settings for the site.

- Select the Direct Line protocol version that your client application will use to communicate with your bot.

TIP

If you are creating a new connection between your client application and bot, use Direct Line API 3.0.

- To enable [Direct Line API performance improvements](#), select **High-speed storage [PREVIEW]**.

When finished, click **Done** to save the site configuration. You can repeat this process, beginning with [Add new site](#), for each client application that you want to connect to your bot.

Connect a bot to Web Chat

8/22/2017 • 2 min to read • [Edit Online](#)

When you register a bot with the Bot Framework, the Web Chat channel is automatically configured for you. The Web Chat channel includes the web chat control, which provides the ability for users to interact with your bot directly in a web page.

The screenshot shows a Microsoft Bot Framework article titled "Explore rich cards with the Web Chat control". The article is dated 2017-5-11 and has 1 min to read. It discusses the ability to attach richer objects like adaptive cards. A sidebar on the left lists navigation links such as Overview, Get started, Samples, How to, Reference, and Resources. The main content area shows a sample hero card with text about building intelligent bots and a "Get Started" button. Below the card, there's a note about rich cards samples in C# and Node.js. At the bottom is a web chat interface with a message input field and a send button.

The Web Chat channel in the Bot Framework Portal contains everything you need to embed the web chat control in a web page. All you have to do to use the web chat control is get your bot's secret key and embed the control in a web page.

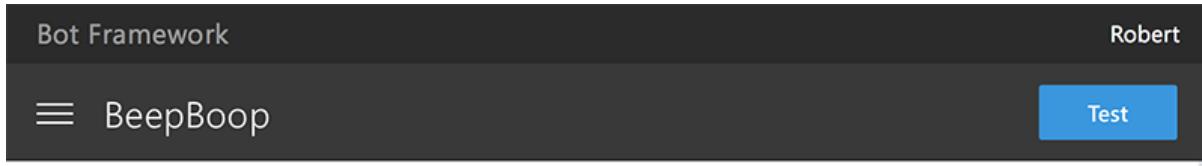
TIP

To see how various Bot Framework features look and work on this channel, [use the Channel Inspector](#).

Get your bot secret key

1. Sign in to the [Bot Framework Portal](#).
2. Click **My bots**.
3. Select your bot.

4. Under **Channels** on the bot dashboard, click **Edit** for the **Web Chat** channel.



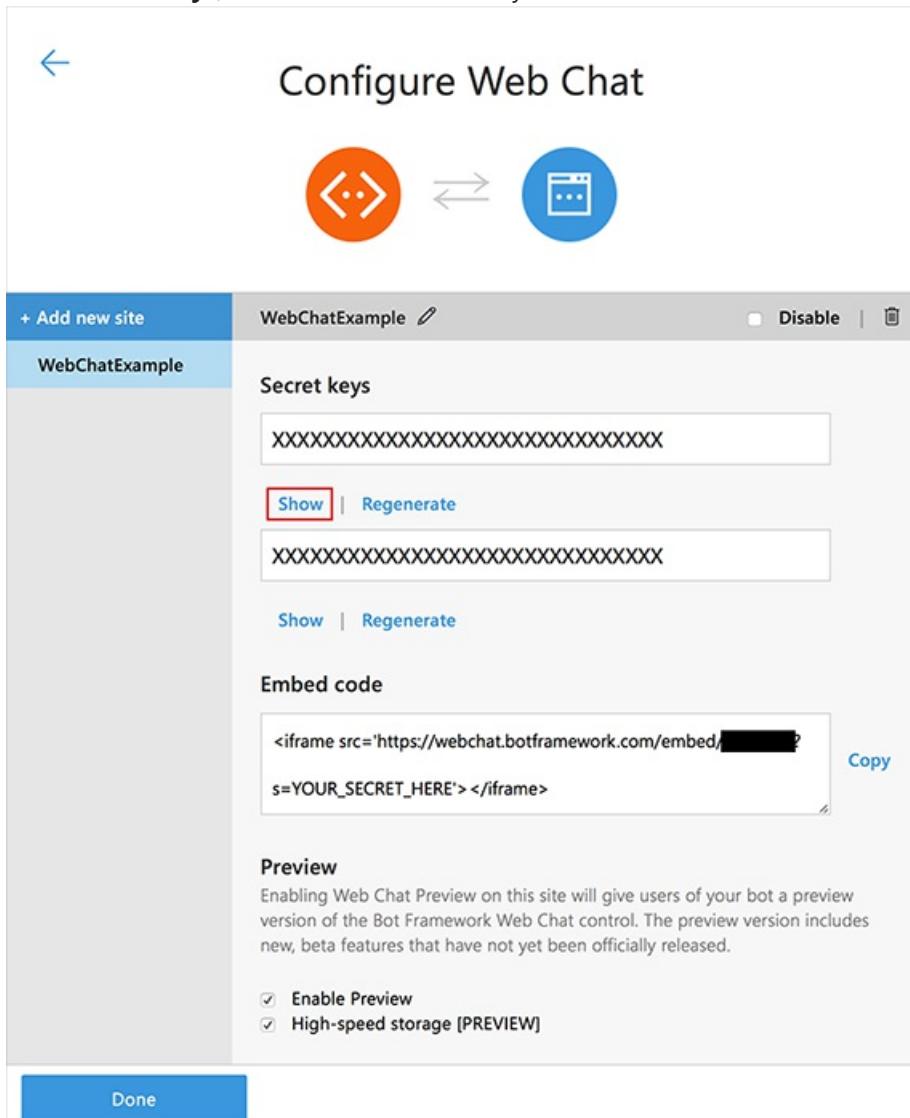
Connect to channels

Name	Health	Published	
Skype	Running	--	Edit
Web Chat	Running	--	Edit

[Get bot embed codes](#)

5. Click **Add new site**, name your site, and click **Done**.

6. Under **Secret keys**, click **Show** for the first key.



Configure Web Chat

WebChatExample | Disable |

Secret keys

XXXXXXXXXXXXXXXXXXXXXXXXXXXX
Show | Regenerate

XXXXXXXXXXXXXXXXXXXXXXXXXXXX
Show | Regenerate

Embed code

```
<iframe src='https://webchat.botframework.com/embed/[REDACTED]'  
s=YOUR_SECRET_HERE'></iframe>
```

Preview
Enabling Web Chat Preview on this site will give users of your bot a preview version of the Bot Framework Web Chat control. The preview version includes new, beta features that have not yet been officially released.

Enable Preview
 High-speed storage [PREVIEW]

Done

7. Copy the **Secret key** and the **Embed code**.

8. Click **Done**.

Embed the web chat control in your website

You can embed the web chat control in your website by using one of two options.

Option 1 - Keep your secret hidden, exchange your secret for a token, and generate the embed

Use this option if you can execute a server-to-server request to exchange your web chat secret for a temporary token, and if you want to make it difficult for other developers to embed your bot in their websites. Although using this option will not absolutely prevent other developers from embedding your bot in their websites, it does make it difficult for them to do so.

To exchange your secret for a token and generate the embed:

1. Issue a **GET** request to `https://webchat.botframework.com/api/tokens` and pass your web chat secret via the `Authorization` header. The `Authorization` header uses the `BotConnector` scheme and includes your secret, as shown in the example request below.
2. The response to your **GET** request will contain the token (surrounded with quotation marks) that can be used to start a conversation by rendering the web chat control within an **iframe**. A token is valid for one conversation only; to start another conversation, you must generate a new token.
3. Within the `iframe` **Embed code** that you copied from the Web Chat channel within the Bot Framework Portal (as described in Step 1 above), change the `s=` parameter to `t=` and replace "YOUR_SECRET_HERE" with your token.

NOTE

Tokens will automatically be renewed before they expire.

Example request

```
GET https://webchat.botframework.com/api/tokens
Authorization: BotConnector YOUR_SECRET_HERE
```

Example response

```
"IIbSpLnn8sA.dBB.MQBhAFMAZwBXAHoANgBQAGcAZABKAEcAMwB2ADQASABjAFMAegBuAHYANwA.bbguxy0v0gE.cccJjh-
TFDs.ruXQiyVZIcgvosGaFs_4jRj1AyPnDt1wk1HMBb5Fuw"
```

Example iframe (using token)

```
<iframe src="https://webchat.botframework.com/embed/YOUR_BOT_ID?t=YOUR_TOKEN_HERE"></iframe>
```

Option 2 - Embed the web chat control in your website using the secret

Use this option if you want to allow other developers to easily embed your bot into their websites.

WARNING

If you use this option, other developers can embed your bot into their websites by simply copying your embed code.

To embed your bot in your website by specifying the secret within the `iframe` tag:

1. Copy the `iframe` **Embed code** from the Web Chat channel within the Bot Framework Portal (as described in Step 1 above).
2. Within that **Embed code**, replace "YOUR_SECRET_HERE" with the **Secret key** value that you copied from the same page.

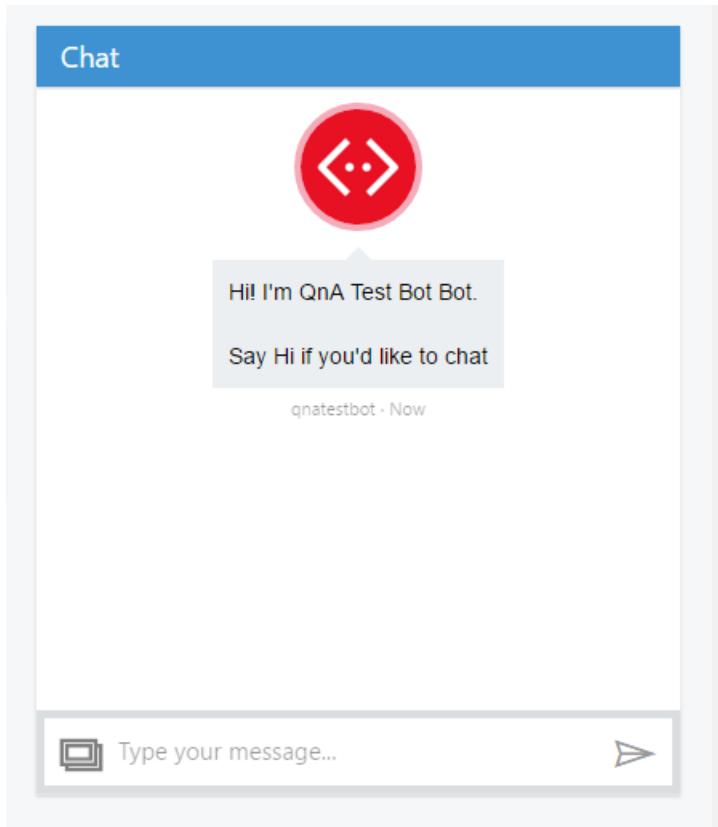
Example `iframe` (using secret)

```
<iframe src="https://webchat.botframework.com/embed/YOUR_BOT_ID?s=YOUR_SECRET_HERE"></iframe>
```

Style the web chat control

You may change the size of the web chat control by using the `style` attribute of the `iframe` to specify `height` and `width`.

```
<iframe style="height:480px; width:402px" src="... SEE ABOVE ..."></iframe>
```



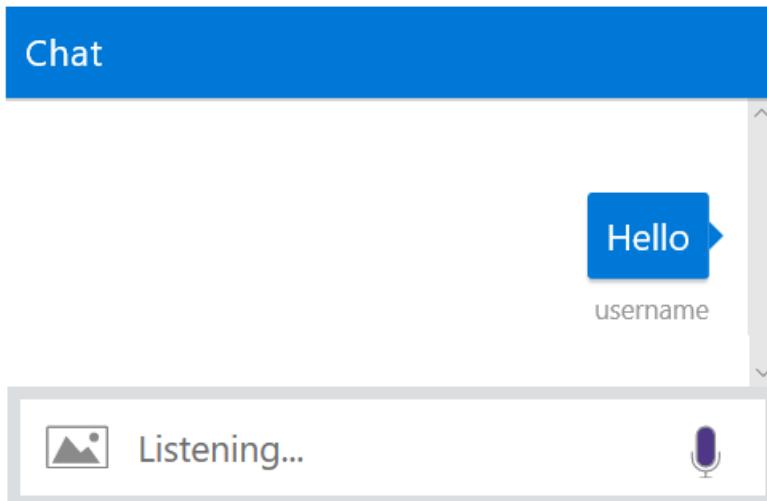
Additional resources

You can [download the source code](#) for the web chat control on GitHub.

How to enable speech in Web Chat

10/23/2017 • 5 min to read • [Edit Online](#)

You can enable a voice interface in the Web Chat control. Users interact with the voice interface by using the microphone in the Web Chat control.



If the user types instead of speaking a response, Web Chat turns off the speech functionality and the bot gives only a textual response instead of speaking out loud. To re-enable the spoken response, the user can use the microphone to respond to the bot the next time. If the microphone is accepting input, it appears dark or filled-in. If it's grayed out, the user clicks on it to enable it.

Prerequisites

Before you run the sample, you need to have a Direct Line secret or token for the bot that you want to run using the Web Chat control.

- See [Connect a bot to Direct Line](#) for information on getting a Direct Line secret associated with your bot.
- See [Generate a Direct Line token](#) for information on exchanging the secret for a token.

Customizing Web Chat for speech

To enable the speech functionality in Web Chat, you need to customize the JavaScript code that invokes the Web Chat control. You can try out voice-enabled Web Chat locally using the following steps.

1. Download the [sample index.html](#).
2. Edit the code in `index.html` according to the type of speech support you want to add. The types of speech implementations are described in [Enable speech services](#).
3. Start a web server. One way to do so is to use `npm http-server` at a Node.js command prompt.
 - To install `http-server` globally so it can be run from the command line, run this command:

```
npm install http-server -g
```

- To start a web server using port 8000, from the directory that contains `index.html`, run this command:

```
http-server -p 8000
```

4. Aim your browser at `http://localhost:8000/samples?parameters`. For example, `http://localhost:8000/samples?s=YOURDIRECTLINESECRET` invokes the bot using a Direct Line secret. The parameters that can be set in the query string are described in the following table:

PARAMETER	DESCRIPTION
s	Direct Line secret. See Connect a bot to Direct Line for information on getting a Direct Line secret.
t	Direct Line token. See Generate a Direct Line token for info on how to generate this token.
domain	Optional. The URL of an alternate Direct Line endpoint.
webSocket	Optional. Set to 'true' to use WebSocket to receive messages. Default is <code>false</code> .
userid	Optional. The ID of the bot user.
username	Optional. The user name of the bot's user.
botid	Optional. ID of the bot.
botname	Optional. Name of the bot.

Enable speech services

The customization allows you to add speech functionality in any of the following ways:

- **Browser-provided speech** - Use speech functionality built into the web browser. At this time, this functionality is only available on the Chrome browser.
- **Use Bing Speech service** - You can use the Bing Speech service to provide speech recognition and synthesis. This way of access speech functionality is supported by a variety of browsers. In this case, the processing is done on a server instead of on the browser.
- **Create a custom speech service** - You can create your own custom speech recognition and voice synthesis components.

Browser-provided speech

The following code instantiates speech recognizer and speech synthesis components that come with the browser. This method of adding speech is not supported by all browsers.

NOTE

Google Chrome supports the browser speech recognizer. However, Chrome may block the microphone in the following cases:

- If the URL of the page that contains Web Chat begins with `http://` instead of `https://`.
- If the URL is a local file using the `file://` protocol instead of `http://localhost:8000`.

```
const speechOptions = {
    speechRecognizer: new BotChat.Speech.BrowserSpeechRecognizer(),
    speechSynthesizer: new BotChat.Speech.BrowserSpeechSynthesizer()
};
```

Bing Speech service

The following code instantiates speech recognizer and speech synthesis components that use the Bing Speech service. The recognition and generation of speech is performed on the server. This mechanism is supported in multiple browsers.

TIP

You can use speech recognition priming to improve your bot's speech recognition accuracy if you use the Bing Speech service. For more information, check out the [Speech Support in Bot Framework](#) blog post.

```
const speechOptions = {
    speechRecognizer: new CognitiveServices.SpeechRecognizer({ subscriptionKey: 'YOUR_COGNITIVE_SPEECH_API_KEY' }),
    speechSynthesizer: new CognitiveServices.SpeechSynthesizer({
        gender: CognitiveServices.SynthesisGender.Female,
        subscriptionKey: 'YOUR_COGNITIVE_SPEECH_API_KEY',
        voiceName: 'Microsoft Server Speech Text to Speech Voice (en-US, JessaRUS)'
    })
};
```

Use the Bing Speech service with a token

You also have the option to enable Cognitive Services speech recognition using a token. The token is generated in a secure back end using your API key.

The following example code shows how the token fetch is done from a secure back end to avoid exposing the API key.

```
function getToken() {
// This call would be to your backend, or to retrieve a token that was served as part of the original page.
return fetch(
    'https://api.cognitive.microsoft.com/sts/v1.0/issueToken',
    {
        headers: {
            'Ocp-Apim-Subscription-Key': 'YOUR_COGNITIVE_SPEECH_API_KEY'
        },
        method: 'POST'
    }
).then(res => res.text());
}

const speechOptions = {
    speechRecognizer: new CognitiveServices.SpeechRecognizer({
        fetchCallback: (authFetchEventId) => getToken(),
        fetchOnExpiryCallback: (authFetchEventId) => getToken()
    }),
    speechSynthesizer: new BotChat.Speech.BrowserSpeechSynthesizer()
};
```

Custom Speech service

You can also provide your own custom speech recognition that implements ISpeechRecognizer or speech synthesis that implements ISpeechSynthesis.

```
const speechOptions = {
    speechRecognizer: new YourOwnSpeechRecognizer(),
    speechSynthesizer: new YourOwnSpeechSynthesizer()
};
```

Pass the speech options to Web Chat

The following code passes the speech options to the Web Chat control:

```
BotChat.App({
    bot: bot,
    locale: params['locale'],
    resize: 'detect',
    // sendTyping: true,      // defaults to false. set to true to send 'typing' activities to bot (and other
    users) when user is typing
    speechOptions: speechOptions,
    user: user,
    directLine: {
        domain: params['domain'],
        secret: params['s'],
        token: params['t'],
        webSocket: params['webSocket'] && params['webSocket'] === 'true' // defaults to true
    }
}, document.getElementById('BotChatGoesHere'));
```

Next steps

Now that you can enable voice interaction with Web Chat, learn how your bot constructs spoken messages and adjusts the state of the microphone:

- [Add speech to messages \(C#\)](#)
- [Add speech to messages \(Node.js\)](#)

Additional resources

- You can [download the source code](#) for the web chat control on GitHub.
- The [Bing Speech API documentation](#) provides more information on the Bing Speech API.

Connect a bot to Office 365 email

8/9/2017 • 1 min to read • [Edit Online](#)

Bots can communicate with users via Office 365 email in addition to other [channels](#). When a bot is configured to access an email account, it receives a message when a new email arrives. The bot can then respond as indicated by its business logic. For example, the bot could send an email reply acknowledging an email was received with the message, "Hi! Thanks for your order! We will begin processing it immediately."

WARNING

It is a violation of the Bot Framework [Code of Conduct](#) to create "spambots", including bots that send unwanted or unsolicited bulk email.

Configure email credentials

You can connect a bot to the Email channel by entering Office 365 credentials in the Email channel configuration.

1. Sign in to the [Bot Framework Portal](#).
2. Click **My bots**.
3. Select your bot.
4. Click the **Email** channel.
5. Enter valid email credentials.
 - Email address
 - Password
6. Click **Save**.

Enter your Email credentials

Email Address

address@email.com

Email Password

password

The Email channel currently works with Office 365 only. Other email services are not currently supported.

Customize emails

The Email channel supports sending custom properties to create more advanced, customized emails using the `channelData` property.

PROPERTY	DESCRIPTION
HtmlBody	The HTML to use for the body of the message.

PROPERTY	DESCRIPTION
Subject	The subject to use for the message.
Importance	The importance flag to use for the message. (Low/Normal/High)

The following example message shows a JSON file that includes the `channelData` properties.

```
{
  "type": "message",
  "locale": "en-US",
  "channelID": "email",
  "from": { "id": "mybot@mydomain.com", "name": "My bot" },
  "recipient": { "id": "joe@otherdomain.com", "name": "Joe Doe" },
  "conversation": { "id": "123123123123", "topic": "awesome chat" },
  "channelData":
  {
    "htmlBody" : "<html><body style = /"font-family: Calibri; font-size: 11pt;/" >This is more than
awesome</body></html>",
    "subject": "Super awesome message subject",
    "importance": "high"
  }
}
```

For more information about using `channelData`, see the [Node.js](#) sample or [.NET](#) documentation.

Additional resources

- Connect a bot to [channels](#)
- [Implement channel-specific functionality](#) with the Bot Builder SDK for .NET
- Use the [Channel Inspector](#) to see how a channel renders a particular feature of your bot application

Connect a bot to GroupMe

8/9/2017 • 1 min to read • [Edit Online](#)

You can configure a bot to communicate with people using the GroupMe group messaging app.

TIP

To see how various Bot Framework features look and work on this channel, [use the Channel Inspector](#).

Sign up for a GroupMe account

If you don't have a GroupMe account, [sign up for a new account](#).

Create a GroupMe application

[Create a GroupMe application](#) for your bot.

Use this callback URL: `https://groupme.botframework.com/Home/Login`

Create Application

Application Name

Callback URL

Callback URL must be https, localhost, or a deep link.

Developer Name

Developer Email

Developer Phone Number

Developer Company

Developer Address

I agree to abide by the [Terms of Use](#) and the [Brand Standards](#)

[Save](#)

[Cancel](#)

Gather credentials

1. In the **Redirect URL** field, copy the value after **client_id=**.

2. Copy the **Access Token** value.

{YOUR BOT NAME}

Details Settings Delete

Settings

Redirect URL https://oauth.groupme.com/oauth/authorize?client_id=Your Client Id

Callback URL https://groupme.botframework.com/Home/Login

Your Access Token

Use the access token string to authenticate as yourself when making API requests.

Access Token **Your Access Token**

Submit credentials

1. On dev.botframework.com, paste the **client_id** value you just copied into the **Client ID** field.
2. Paste the **Access Token** value into the **Access Token** field.
3. Click **Save**.

Access Token	Access token can be found in your GroupMe account settings
Client ID	Client ID can be found in your GroupMe account settings

Connect a bot to Facebook Messenger

10/24/2017 • 2 min to read • [Edit Online](#)

To learn more about developing for Facebook Messenger, see the [Messenger platform documentation](#). You may wish to review Facebook's [pre-launch guidelines](#), [quick start](#), and [setup guide](#).

To configure a bot to communicate using Facebook Messenger, enable Facebook Messenger on a Facebook page and then connect the bot to the app.

TIP

To see how various Bot Framework features look and work on this channel, use the [Channel Inspector](#).

NOTE

The Facebook UI may appear slightly different depending on which version you are using.

Copy the Page ID

The bot is accessed through a Facebook Page. [Create a new Facebook Page](#) or go to an existing Page.

- Open the Facebook Page's **About** page and then copy and save the **Page ID**.

Create a Facebook app

[Create a new Facebook App](#) on the Page and generate an App ID and App Secret for it.

Create a New App ID

Get started integrating Facebook into your app or website

Display Name

The name you want to associate with this App ID

Contact Email

Used for important communication about your app

By proceeding, you agree to the [Facebook Platform Policies](#)

[Cancel](#) [Create App ID](#)

- Copy and save the **App ID** and the **App Secret**.

Dashboard

TestBot

This app is in development mode and can only be used by app admins, developers and testers [?]

API Version [?]

v2.6

App ID

.....

App Secret

.....

Show

Set the "Allow API Access to App Settings" slider to "Yes".

No

Require App Secret

Only allow calls from a server and require app secret or app secret proof for all API calls.

No

Require 2-Factor Reauthorization

Require 2-fac to change application settings

Yes

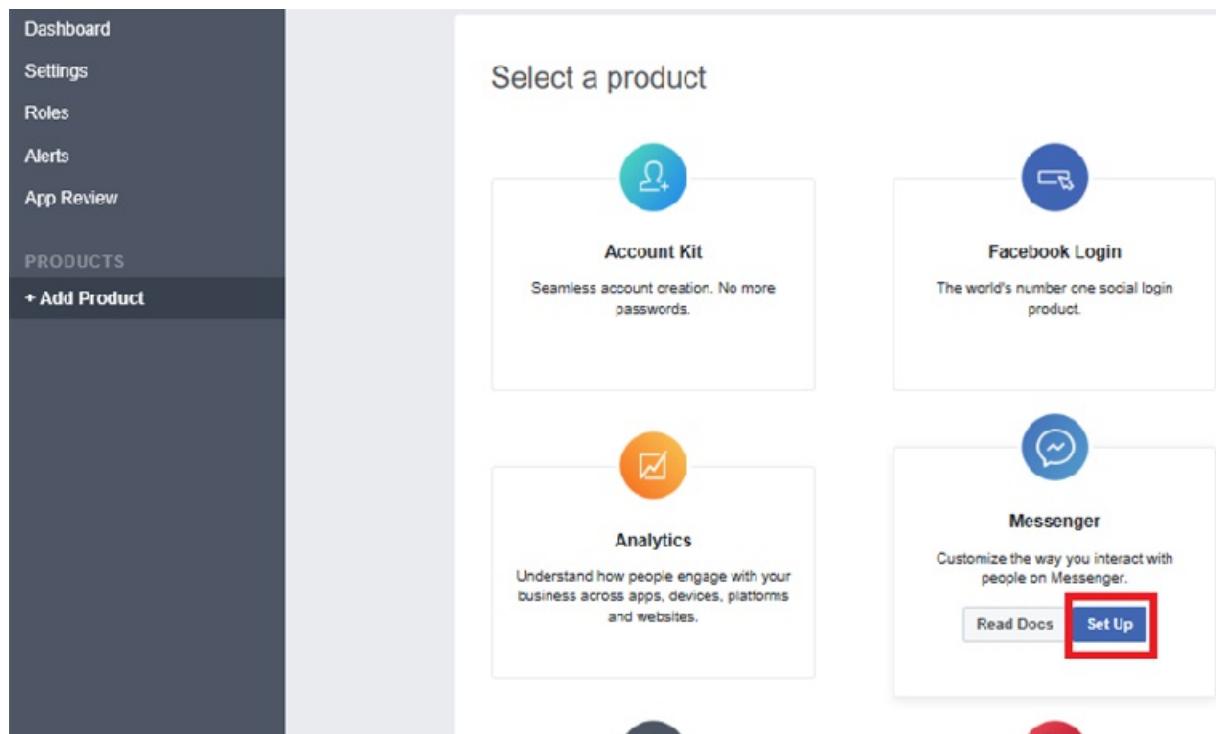
Allow API Access to App Settings

Set to No to prevent changes to app settings through API calls

Enable messenger

Enable Facebook Messenger in the new Facebook App.

- On the **Product Setup** page of the app, click **Get Started** and then click **Get Started** again.



The screenshot shows the 'Select a product' section of the Facebook Product Setup page. On the left, there's a sidebar with links like Dashboard, Settings, Roles, Alerts, App Review, and a 'PRODUCTS' section with '+ Add Product'. The main area shows four product options: 'Account Kit' (blue icon), 'Facebook Login' (blue icon), 'Analytics' (orange icon), and 'Messenger' (blue icon). Below each product is a brief description and a 'Read Docs' or 'Set Up' button. The 'Messenger' 'Set Up' button is highlighted with a red box.

Generate a Page Access Token

In the **Token Generation** panel of the Messenger section, select the target Page. A Page Access Token will be generated.

- Copy and save the **Page Access Token**.

Token Generation

Page token is required to start using the APIs. This page token will have all messenger permissions even if your app is not approved to use them yet, though in this case you will be able to message only app admins. You can also generate page tokens for the pages you don't own using Facebook Login.

Page

Select a Page ▾

✓ Select a Page

{your page}

Page Access Token

You must select a Page to generate an access token.

Webhooks

Setup Webhooks

To receive messages and other events sent by Messenger users, the app should enable webhooks integration.

Enable webhooks

Click **Set up Webhooks** to forward messaging events from Facebook Messenger to the bot.

Webhooks

Setup Webhooks

To receive messages and other events sent by Messenger users, the app should enable webhooks integration.

Provide webhook callback URL and verify token

Return to the [Bot Framework Portal](#). Open the bot, click the **Channels** tab, and then click **Facebook Messenger**.

- Copy the **Callback URL** and **Verify Token** values from the portal.

Callback URL and Verify Token for Facebook

[What do I do with my Callback URL and Verify token?](#)

Callback URL (Copy and paste in Facebook)

Verify Token (Copy and paste in Facebook)

- Return to Facebook Messenger and paste the **Callback URL** and **Verify Token** values.
- Under **Subscription Fields**, select *message_deliveries*, *messages*, *messaging_options*, and *messaging_postbacks*.
- Click **Verify and Save**.

New Page Subscription

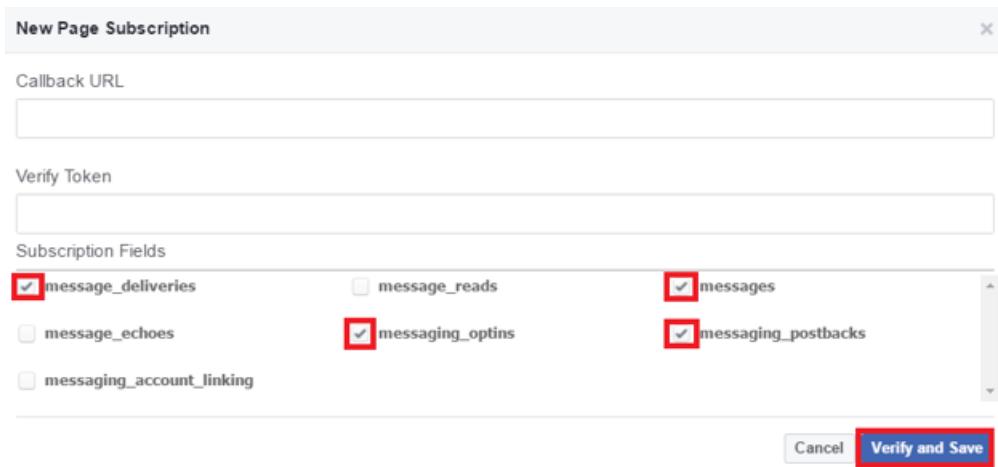
Callback URL

Verify Token

Subscription Fields

<input checked="" type="checkbox"/> message_deliveries	<input type="checkbox"/> message_reads	<input checked="" type="checkbox"/> messages
<input type="checkbox"/> message_echoes	<input checked="" type="checkbox"/> messaging_optins	<input checked="" type="checkbox"/> messaging_postbacks
<input type="checkbox"/> messaging_account_linking		

Cancel **Verify and Save**



Provide Facebook credentials

On the Bot Framework Portal, paste the **Page ID**, **App ID**, **App Secret**, and **Page Access Token** values copied from Facebook Messenger previously.

Enter your Facebook Messenger credentials

[Where do I find my Facebook Messenger credentials?](#)

Facebook Page ID

Facebook App ID

Facebook App Secret

Page Access Token

Submit for review

Facebook requires a Privacy Policy URL and Terms of Service URL on its basic app settings page. The [Code of Conduct](#) page contains third party resource links to help create a privacy policy. The [Terms of Use](#) page contains sample terms to help create an appropriate Terms of Service document.

After the bot is finished, Facebook has its own [review process](#) for apps that are published to Messenger. The bot will be tested to ensure it is compliant with Facebook's [Platform Policies](#).

Make the App public and publish the Page

NOTE

Until an app is published, it is in [Development Mode](#). Plugin and API functionality will only work for admins, developers, and testers.

After the review is successful, in the App Dashboard under App Review, set the app to Public. Ensure that the

Facebook Page associated with this bot is published. Status appears in Pages settings.

Connect a bot to Kik

8/9/2017 • 1 min to read • [Edit Online](#)

You can configure your bot to communicate with people using the Kik messaging app.

TIP

To see how various Bot Framework features look and work on this channel, [use the Channel Inspector](#).

Install Kik on your phone

If you don't have Kik installed on your phone, you can install it via your phone's app store or at [the Kik website](#)

Log into the dev portal with your mobile phone

[Log into the Kik portal](#) on your mobile phone.

Follow the bot setup process

Give the bot a name.



Botsworth



Welcome! Let's get started. What would you like your bot's username to be?

R

Yourbotname



I will create a new bot named "yourbotname". Is this correct?



Tap a message



Yes

No

Gather credentials

On the Configuration tab, copy the **Name** and **API key**.



Account Settings



yourbotname

Display Name

yourbotname

Cancel

Save

Admins

Add an admin...

Add

larsliden

API Key

API Key:

39020300000000000000000000000000

Generated:

14:20 Apr 13, 2016

Regenerate

Submit credentials

Bot Name

yourbothame

API Key

Submit Kik Credentials

Click **Submit Kik Credentials**.

Enable the bot

Check **Enable this bot on Kik**. Then click **I'm done configuring Kik**.

When you have completed these steps, your bot will be successfully configured to communicate with users in Kik.

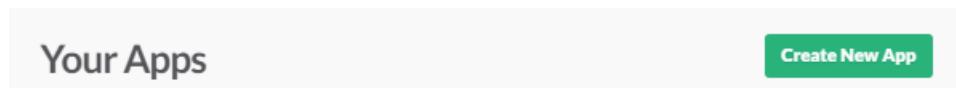
Connect a bot to Slack

10/24/2017 • 3 min to read • [Edit Online](#)

You can configure your bot to communicate with people using the Slack messaging app.

Create a Slack Application for your bot

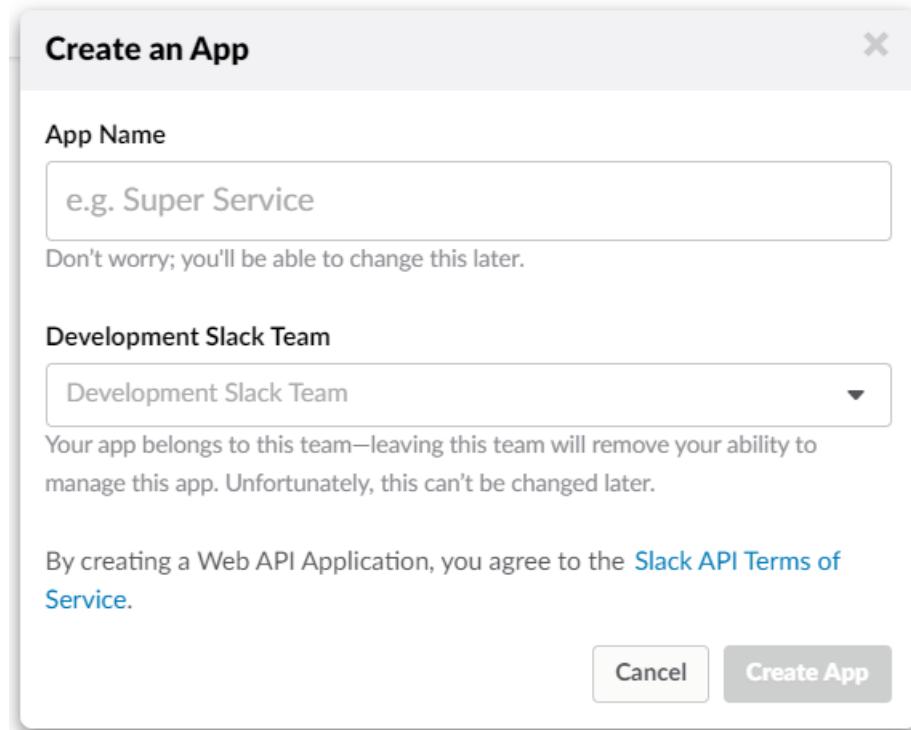
Log into Slack and [create a Slack application](#).



The screenshot shows a light gray header bar with the text "Your Apps" on the left and a green rectangular button on the right containing the white text "Create New App".

Create an app and assign a Development Slack team

Enter an App Name and select a Development Slack Team. If you are not already a member of a Development Slack Team, [create or join one](#).



The dialog box has a title bar "Create an App" with a close button. It contains two main sections: "App Name" and "Development Slack Team".

App Name: A text input field with placeholder text "e.g. Super Service". Below it is a note: "Don't worry; you'll be able to change this later."

Development Slack Team: A dropdown menu set to "Development Slack Team". Below the dropdown is a note: "Your app belongs to this team—leaving this team will remove your ability to manage this app. Unfortunately, this can't be changed later."

At the bottom, there is a note: "By creating a Web API Application, you agree to the [Slack API Terms of Service](#)." Below the note are two buttons: "Cancel" and "Create App".

Click **Create App**. Slack will create your app and generate a Client ID and Client Secret.

Add a new Redirect URL

Next you will add a new Redirect URL.

1. Select the **OAuth & Permissions** tab.
2. Click **Add a new Redirect URL**.
3. Enter <https://slack.botframework.com>.
4. Click **Add**.
5. Click **Save URLs**.

YourNewApp ▾

OAuth & Permissions

Settings

- Basic Information
- Collaborators
- Install App
- Manage Distribution

Features

- Incoming Webhooks
- Interactive Messages
- Slash Commands

OAuth & Permissions

- Event Subscriptions
- Bot Users

Slack ❤️

- Help
- Contact
- Policies
- Our Blog

OAuth Tokens & Redirect URLs

These [OAuth Tokens](#) will be automatically generated when you finish connecting the app to your team. You'll use these tokens to authenticate your app.

[Install App to Team](#)

Redirect URLs

You will need to configure redirect URLs in order to automatically generate the Add to Slack button or to distribute your app. If you pass a URL in an OAuth request, it must (partially) match one of the URLs you enter here. [Learn more](#)

Redirect URLs

https://slack.botframework.com

Add a new Redirect URL

[Save URLs](#)

Cancel [Add](#) <

Create a Slack Bot User

Adding a Bot User allows you to assign a username for your bot and choose whether it is always shown as online.

1. Select the **Bot Users** tab.
2. Click **Add a Bot User**.

YourNewApp ▾

Bot User

Settings

- Basic Information
- Collaborators
- Install App
- Manage Distribution

Features

- Incoming Webhooks
- Interactive Messages
- Slash Commands
- OAuth & Permissions
- Event Subscriptions

Bot Users

You can bundle a bot user with your app to interact with users in a more conversational manner. Learn more about [how bot users work](#).

[Add a Bot User](#) <

Click **Add Bot User** to validate your settings, click **Always Show My Bot as Online** to **On**, and then click **Save Changes**.

YourNewApp ▾

Bot User

Settings

Basic Information
Collaborators
Install App
Manage Distribution

Features

Incoming Webhooks
Interactive Messages
Slash Commands
OAuth & Permissions
Event Subscriptions

Bot Users

You can bundle a bot user with your app to interact with users in a more conversational manner. Learn more about [how bot users work](#).

Default username

@yournewapp <

If this username isn't available on any team that tries to install it, we will slightly change it to make it work. Usernames must be all lowercase. They cannot be longer than 21 characters and can only contain letters, numbers, periods, hyphens, and underscores.

Always Show My Bot as Online  On

When this is off, Slack automatically displays whether your bot is online based on usage of the RTM API.

Add Bot User <

Slack ❤

Subscribe to Bot Events

Follow these steps to subscribe to six particular bot events. By subscribing to bot events, your app will be notified of user activities at the URL you specify.

TIP

Your bot handle is a property of your bot. To find a bot's handle, visit <https://dev.botframework.com/bots>, choose a bot, and click **SETTINGS**.

1. Select the **Event Subscriptions** tab.
2. Click **Enable Events** to **On**.
3. In **Request URL**, enter this URL, but replace `{YourBotHandle}` with your bot handle.
`https://slack.botframework.com/api/Events/{YourBotHandle}`
4. In **Subscribe to Bot Events**, click **Add Bot User Event**.
5. In the list of events, click **Add Bot User Event** and select these six event types:
 - `member_joined_channel`
 - `member_left_channel`
 - `message.channels`
 - `message.groups`
 - `message.im`
 - `message.mpim`
6. Click **Save Changes**.

The screenshot shows the Slack App Dashboard for 'YourNewApp'. On the left, there's a sidebar with 'Settings' and 'Features' sections, and a blue highlighted 'Event Subscriptions' tab. The main area is titled 'Event Subscriptions' and contains a section titled 'Enable Events' with an 'On' toggle switch. Below it is a 'Request URL' input field containing 'https://slack.botframework.com/api/Events/{YourBotHandle}' with a 'challenge' placeholder. A note explains that Slack will send HTTP POST requests to this URL when events occur, including a challenge parameter.

Add and Configure Interactive Messages (optional)

If your bot will use Slack-specific functionality such as buttons, follow these steps:

1. Select the **Interactive Components** tab and click **Enable Interactive Components**.
2. Enter `https://slack.botframework.com/api/Actions` as the **Request URL**.
3. Click the **Enable Interactive Messages** button, and then click the **Save changes** button.

The screenshot shows the Slack App Dashboard for 'YourNewApp'. On the left, there's a sidebar with 'Settings' and 'Features' sections, and a blue highlighted 'Interactive Messages' tab. The main area is titled 'Interactive Messages' and contains a note about adding buttons to messages. It has a 'Request URL' input field with 'https://slack.botframework.com/api/Actions' and a note explaining it sends an HTTP POST request when users invoke message buttons. Below is an 'Options Load URL' input field with 'https://my.app.com/slack/options-load-endpoint' and a note about loading options from this URL when users invoke message menus. At the bottom is a green 'Enable Interactive Messages' button.

Gather credentials

Select the **Basic Information** tab and scroll to the **App Credentials** section. The Client ID, Client Secret, and Verification Token required for configuration of your Slack bot are displayed.

Slack ❤

Help
Contact
Policies
Our Blog

App Credentials

These credentials allow your app to access the Slack API. They are secret. Please don't share your app credentials with anyone, include them in public code repositories, or store them in insecure ways.

Client ID

16 00

Client Secret

Show Regenerate

You'll need to send this secret along with your client ID when making your [oauth.access](#) request.

Verification Token

Zj h0

Regenerate

For interactive messages and events, use this token to verify that requests are actually coming from Slack. Slash commands and interactive messages will both use this verification token.

Submit credentials

In a separate browser window, return to the Bot Framework site at <https://dev.botframework.com/>.

1. Select **My bots** and choose the Bot that you want to connect to Slack.
2. In the **Add a channel** section, click the Slack icon.
3. In the **Enter your Slack credentials** section, paste the App Credentials from the Slack website into the appropriate fields.
4. The **Landing Page URL** is optional. You may omit or change it.
5. Click **Save**.

▲ Submit your Credentials

Client Id	16 00
Client Secret	6 1
Verification Token	Z h0
Landing Page URL	https://yourbot.example.com/slack_help (optional)

Users will be redirected to this URL after adding your bot to Slack.

Submit Slack Credentials

Follow the instructions to authorize your Slack app's access to your Development Slack Team.

Enable the bot

On the Configure Slack page, confirm the slider by the Save button is set to **Enabled**. Your bot is configured to communicate with users in Slack.

Create an Add to Slack button

Slack provides HTML you can use to help Slack users find your bot in the *Add the Slack button* section of [this page](#). To use this HTML with your bot, replace the href value (begins with <https://>) with the URL found your bot's Slack

channel settings. Follow these steps to get the replacement URL.

1. On <https://dev.botframework.com/bots>, click your bot.
2. Click **CHANNELS**, right-click the entry named **Slack**, and click **Copy link**. This URL is now in your clipboard.
3. Paste this URL from your clipboard into the HTML provided for the Slack button. This URL replaces the href value provided by Slack for this bot.

Authorized users can click the **Add to Slack** button provided by this modified HTML to reach your bot on Slack.

Connect a bot to Telegram

6/28/2017 • 1 min to read • [Edit Online](#)

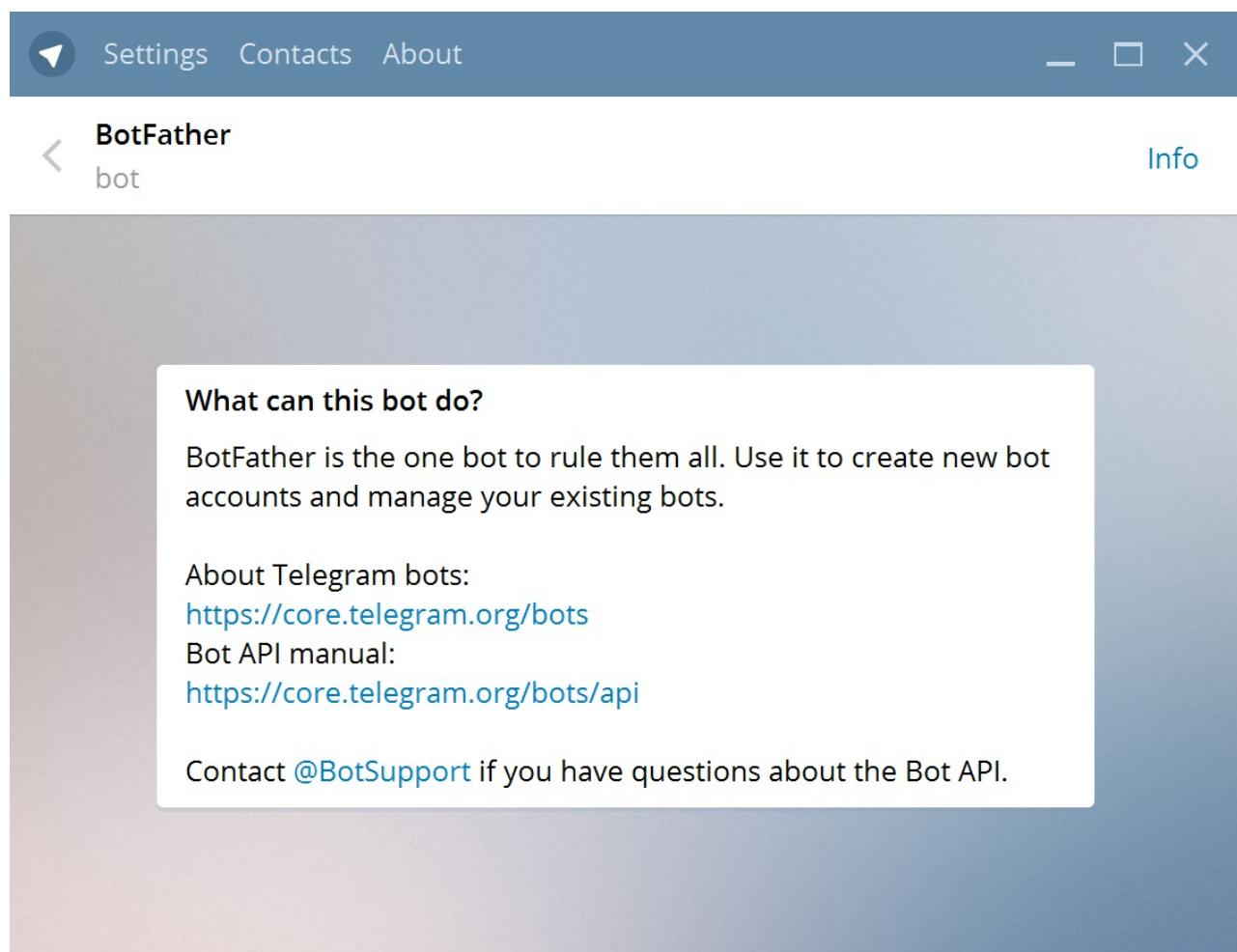
You can configure your bot to communicate with people using the Telegram messaging app.

TIP

To see how various Bot Framework features look and work on this channel, [use the Channel Inspector](#).

Visit the Bot Father to create a new Telegram bot

Create a new [Telegram bot](#) using the Bot Father.



Create a new Telegram bot

To create a new Telegram bot, send command `/newbot`.



Settings Contacts About

- □ ×

BotFather

bot

Info

<https://core.telegram.org/bots>

You can control me by sending these commands:

/newbot - create a new bot
/token - generate authorization token
/revoke - revoke bot access token
/setname - change a bot's name
/setdescription - change bot description
/setabouttext - change bot about info
/setuserpic - change bot profile photo
/setinline - change inline settings
/setinlinefeedback - change inline feedback settings
/setcommands - change bot commands list
/setjoininggroups - can your bot be added to groups?
/setprivacy - what messages does your bot see in groups?
/deletebot - delete a bot



/newbot

create a new bot



/newbot



Send

Specify a friendly name

Give the Telegram bot a friendly name.



Settings Contacts About



BotFather

bot

Info

/revoke - revoke bot access token
/setname - change a bot's name
/setdescription - change bot description
/setabouttext - change bot about info
/setuserpic - change bot profile photo
/setinline - change inline settings
/setinlinefeedback - change inline feedback settings
/setcommands - change bot commands list
/setjoininggroups - can your bot be added to groups?
/setprivacy - what messages does your bot see in groups?
/deletebot - delete a bot
/cancel - cancel the current operation

1:44 PM

/newbot 1:44 PM ✓

Alright, a new bot. How are we going to call it? Please choose a name for your bot.

1:44 PM



Delightful Bot



Send

Specify a username

Give the Telegram bot a unique username.



Settings Contacts About

- □ ×

BotFather

bot

Info

/setinline - change inline settings
/setinlinefeedback - change inline feedback settings
/setcommands - change bot commands list
/setjoininggroups - can your bot be added to groups?
/setprivacy - what messages does your bot see in groups?
/deletebot - delete a bot
/cancel - cancel the current operation

1:44 PM

/newbot 1:44 PM ✓

Alright, a new bot. How are we going to call it? Please choose a name for your bot.

1:44 PM

Delightful Bot 1:44 PM ✓

Good. Now let's choose a username for your bot. It must end in `bot`. Like this, for example: TetrisBot or tetris_bot.

1:44 PM



DelightfulBot



Send

Copy the access token

Copy the Telegram bot's access token.



Good. Now let's choose a username for your bot. It must end in `bot`. Like this, for example: TetrisBot or tetris_bot.

1:44 PM

DelightfulBot

1:45 PM

Done! Congratulations on your new bot. You will find it at telegram.me/DelightfulBot. You can now add a description, about section and profile picture for your bot, see [/help](#) for a list of commands.

Use this token to access the HTTP API:

183547168:AAHB5Ne2yzV5qfOvgAAGW0DHWRG0OjQLDEg

For a description of the Bot API, see this page:

<https://core.telegram.org/bots/api>

1:45 PM



| Write a message..



Enter the Telegram bot's access token

Paste the token you copied previously into the **Access Token** field and click **Submit**.

Enable the bot

Check **Enable this bot on Telegram**. Then click **I'm done configuring Telegram**.

When you have completed these steps, your bot will be successfully configured to communicate with users in Telegram.

Connect a bot to Twilio

9/6/2017 • 1 min to read • [Edit Online](#)

You can configure your bot to communicate with people using the Twilio cloud communication platform.

Log in to or create a Twilio account for sending and receiving SMS messages

If you don't have a Twilio account, [create a new account](#)

Create a TwiML Application

[Create a TwiML application](#)

Create TwiML App

Properties

Friendly Name

- ANY VALUE -

Voice

Request URL

- LEAVE BLANK -

HTTP POST

Messaging

Request URL

<https://sms.botframework.com/api/sms>

HTTP POST

[Save](#)

[Cancel](#)

Under Messaging, the Request URL should be <https://sms.botframework.com/api/sms>.

Select or add a phone number

[Select or add a phone number](#). Click the number to add it to the TwiML application you created.

Manage Numbers

[Buy a number](#)

Number	Voice URL	Capabilities	Configuration	Filter
Number	Friendly Name	Capabilities	Configuration	
+1 555-555-5555	(206) 488-1850		<input checked="" type="checkbox"/> Voice TwiML App: <input checked="" type="checkbox"/> Messaging TwiML App:	

Specify application to use for Messaging

In the **Messaging** section, set the **TwiML App** to the name of the TwiML App you just created. Copy the **Phone Number** value for later use.

(555) 555-5555

[Configure](#) [Event Log](#)

▼ Properties

Friendly Name (555) 555-5555

SID

Phone Number +155555555555 <- COPY FOR LATER USE

Location N/A

Capabilities Voice, SMS, MMS

▼ Voice

[View Calls](#) [Inbound](#) | [Outbound](#)

Configure with URL TwiML App SIP Trunking

TwiML App

- LEAVE BLANK -

+ Create a TwiML app

▼ Messaging

[View Messages](#) [Inbound](#) | [Outbound](#)

Configure with URL TwiML App Messaging Service

TwiML App

- SET TO YOUR TWIML NAME -

+ Create a new TwiML App

[Save](#)

[Cancel](#)

Release Number

Gather credentials

Gather credentials and then click the "eye" icon to see the Auth Token.

Account Settings

This is a subaccount of master account fuseman@microsoft.com's Account

Account Name

Name your account with your business name, organization or purpose (Dev, Stage or Prod).

Two-Factor Authentication

Add an extra layer of protection to your account, with a verification code sent via SMS or voice call.

Disabled

Do not require a verification code.

Once Per Computer

Trust computers and only ask for verification code every 30 days.

Every Log-in

We'll always ask for a verification code.

API Credentials

	AccountSID Used to exercise the REST API 
Live	AuthToken (Request a Secondary Token) Keep this somewhere safe and secure  Learn more about REST API Credentials

<- ACCOUNT SID

<- AUTH TOKEN (Click Lock To View)

Submit credentials

Enter the phone number, accountSID, and Auth Token you copied earlier and click **Submit Twilio Credentials**.

Enable the bot

Check **Enable this bot on SMS**. Then click **I'm done configuring SMS**.

When you have completed these steps, your bot will be successfully configured to communicate with users using Twilio.

General Bot Review Guidelines

5/10/2017 • 3 min to read • [Edit Online](#)

We welcome you and thank you for investing your talents and time in building bots using Microsoft's Bot Framework. Following are the minimum requirements your bot must meet before it may be published to a Microsoft channel such as Skype, Bing and Cortana. Each channel may have specific guidelines in addition to the requirements detailed below. If applicable, you'll find channel specific guidelines on each channel's configuration page.

Microsoft will review your bot submission to make sure it meets certain minimum requirements before it may be available on a directory. Following are some criteria we use to evaluate your bot before publication:

1. Your bot must do something meaningful that adds value to the user (e.g., respond to basic commands, have at least some level of communication etc.).
2. The bot profile image, name, keywords and description must NOT:
 - a. be offensive or explicit;
 - b. include third party trademarks, service marks or logos;
 - c. impersonate or imply endorsement by a third party;
 - d. use names unrelated to the bot;
 - e. use Microsoft logos, trademarks or service marks unless you have permission from Microsoft; or
 - f. be too long or verbose. The description should be 8-10 words.
3. For payment enabled bots:
 - a. Your bot may not transmit financial instrument details through the user to bot interface;
 - b. Subject to any channel or third party platform restrictions, your bot may: (a) support payments through the [Microsoft Seller Center](#) subject to the terms of the [Microsoft Seller Center Agreement](#); or (b) transmit links to other secure payment services;
 - c. If your bot enables the foregoing payment mechanisms, you must disclose this in your bot's terms of use and privacy policy (and any profile page or website for the bot) before the end user agrees to use your bot;
 - d. You must clearly indicate that the bot is payment-enabled in the bot profile and provide end users with the merchant's customer service details; and
 - e. You may not publish bots on any Microsoft channel that include links or otherwise direct users to payment services for the purchase of digital goods.
4. Your Terms of Service link is required for submission and publication to all Microsoft channels. If your bot handles users' personal data you must also provide a link to an applicable privacy policy, and such privacy policy must comply with all applicable laws, regulations and policy requirements. In addition, you will need to ensure that you follow the privacy notice requirements as communicated in the Developer Code of Conduct for the Microsoft Bot Framework referenced here: <https://aka.ms/bf-conduct>.
5. Your bot must operate as described in its bot description, profile, terms of use and privacy policy, and you must notify Microsoft in advance by sending an email to bf-reports@microsoft.com, if you make any material changes to your bot. Microsoft has the right, in its sole discretion, to intermittently review bots on any channel and remove bots without notice.
6. Microsoft reserves the right to dismantle a channel's bot directory at any time without notice.
7. Your bot must operate in accordance with the requirements set forth in the Microsoft Bot Framework

[Online Services Agreement](#) and [Developer Code of Conduct](#) for the Microsoft Bot Framework. In addition, your bot must meet the applicable requirements for each channel upon which your bot is published.

8. Changes made to your bot's registration may require your bot to be re-reviewed to ensure that it continues to meet the requirements stated here.
9. Although Microsoft will review your bot to confirm it meets certain minimum requirements prior to publication, you are solely responsible for:
 - a. your bot;
 - b. its content and actions;
 - c. compliance with all applicable laws;
 - d. compliance with any third party terms and conditions; and
 - e. compliance with the Microsoft Bot Framework Online Services Agreement, Privacy Statement and Developer Code of Conduct for the Microsoft Bot Framework.
10. Microsoft's review and publication of your bot to a channel's bot directory is not an endorsement of your bot.
11. Keep your Microsoft Account email active as we will use that for all bot-related communication with you.

Troubleshooting general problems

10/24/2017 • 16 min to read • [Edit Online](#)

These frequently asked questions can help you to troubleshoot common bot development or operational issues.

How can I troubleshoot issues with my bot?

1. Debug your bot's source code with [Visual Studio Code](#) or Visual Studio.
2. Test your bot using the [emulator](#) before you deploy it to the cloud.
3. Deploy your bot to a cloud hosting platform such as Azure and then test connectivity to your bot by using the built-in web chat control on your bot's dashboard in the [Bot Framework Portal](#). If you encounter issues with your bot after you deploy it to Azure, you might consider using this guide: [Troubleshoot a web app in Azure App Service using Visual Studio](#).
4. Rule out [authentication](#) as a possible issue.
5. Test your bot on Skype. This will help you to validate the end-to-end user experience.
6. Consider testing your bot on channels that have additional authentication requirements such as Direct Line or Web Chat.

How can I troubleshoot authentication issues?

For details about troubleshooting authentication issues with your bot, see [Troubleshooting Bot Framework authentication](#).

I'm using the Bot Builder SDK for .NET. How can I troubleshoot issues with my bot?

Look for exceptions.

In Visual Studio 2017, go to **Debug > Windows > Exception Settings**. In the **Exceptions Settings** window, select the **Break When Thrown** checkbox next to **Common Language Runtime Exceptions**. You may also see diagnostics output in your Output window when there are thrown or unhandled exceptions.

Look at the call stack.

In Visual Studio, you can choose whether or you are debugging [Just My Code](#) or not. Examining the full call stack may provide additional insight into any issues.

Ensure all dialog methods end with a plan to handle the next message.

All `IDialog` methods should complete with `IDialogStack.Call`, `IDialogStack.Wait`, or `IDialogStack.Done`. These `IDialogStack` methods are exposed through the `IDialogContext` that is passed to every `IDialog` method. Calling `IDialogStack.Forward` and using the system prompts through the `PromptDialog` static methods will call one of these methods in their implementation.

Ensure that all dialogs are serializable.

This can be as simple as using the `[Serializable]` attribute on your `IDialog` implementations. However, be aware that anonymous method closures are not serializable if they reference their outside environment to capture variables. The Bot Framework supports a reflection-based serialization surrogate to help serialize types that are not marked as serializable.

Why doesn't the Typing activity do anything?

Some channels do not support transient typing updates in their client.

What is the difference between the Connector library and Builder library in the SDK?

The Connector library is the exposition of the REST API. The Builder library adds the conversational dialog programming model and other features such as prompts, waterfalls, chains, and guided form completion. The Builder library also provides access to cognitive services such as LUIS.

What causes an error with HTTP status code 429 "Too Many Requests"?

An error response with HTTP status code 429 indicates that too many requests have been issued in a given amount of time. The body of the response should include an explanation of the problem and may also specify the minimum required interval between requests. One possible source for this error is [ngrok](#). If you are on a free plan and running into ngrok's limits, go to the pricing and limits page on their website for more [options](#).

How can I run background tasks in ASP.NET?

In some cases, you may want to initiate an asynchronous task that waits for a few seconds and then executes some code to clear the user profile or reset conversation/dialog state. For details about how to achieve this, see [How to run Background Tasks in ASP.NET](#). In particular, consider using `HostingEnvironment.QueueBackgroundWorkItem`.

How do user messages relate to HTTPS method calls?

When the user sends a message over a channel, the Bot Framework web service will issue an HTTPS POST to the bot's web service endpoint. The bot may send zero, one, or many messages back to the user on that channel, by issuing a separate HTTPS POST to the Bot Framework for each message that it sends.

My bot is slow to respond to the first message it receives. How can I make it faster?

Bots are web services and some hosting platforms, including Azure, automatically put the service to sleep if it does not receive traffic for a certain period of time. If this happens to your bot, it must restart from scratch the next time it receives a message, which makes its response much slower than if it was already running.

Some hosting platforms enable you to configure your service so that it will not be put to sleep. To do this in Azure, navigate to your bot's service in the [Azure Portal](#), select **Application settings**, and then select **Always on**. This option is available in most, but not all, service plans.

How can I guarantee message delivery order?

The Bot Framework will preserve message ordering as much as possible. For example, if you send message A and wait for the completion of that HTTP operation before you initiate another HTTP operation to send message B, the Bot Framework will automatically understand that message A should precede message B. However, in general, message delivery order cannot be guaranteed since the channel is ultimately responsible for message delivery and may reorder messages. To mitigate the risk of messages being delivered in the wrong order, you might choose to implement a time delay between messages.

How can I intercept all messages between the user and my bot?

Using the Bot Builder SDK for .NET, you can provide implementations of the `IPostToBot` and `IBotToUser` interfaces to the `Autofac` dependency injection container. Using the Bot Builder SDK for Node.js, you can use

middleware for much the same purpose. The [BotBuilder-Azure](#) repository contains C# and Node.js libraries that will log this data to an Azure table.

Why are parts of my message text being dropped?

The Bot Framework and many channels interpret text as if it were formatted with [Markdown](#). Check to see if your text contains characters that may be interpreted as Markdown syntax.

How can I support multiple bots at the same bot service endpoint?

This [sample](#) shows how to configure the `Conversation.Container` with the right `MicrosoftAppCredentials` and use a simple `MultiCredentialProvider` to authenticate multiple App IDs and passwords.

Identifiers

How do identifiers work in the Bot Framework?

For details about identifiers in the Bot Framework, see the [Bot Framework guide to identifiers](#).

How can I get access to the user ID?

SMS and email messages will provide the raw user ID in the `from.Id` property. In Skype messages, the `from.Id` property will contain a unique ID for the user which differs from the user's Skype ID. If you need to connect to an existing account, you can use a sign-in card and implement your own OAuth flow to connect the user ID to your own service's user ID.

Why are my Facebook user names not showing anymore?

Did you change your Facebook password? Doing so will invalidate the access token, and you will need to update your bot's configuration settings for the Facebook Messenger channel in the [Bot Framework Portal](#).

Why is my Kik bot replying "I'm sorry, I can't talk right now"?

Bots in development on Kik are allowed 50 subscribers. After 50 unique users have interacted with your bot, any new user that attempts to chat with your bot will receive the message "I'm sorry, I can't talk right now." For more information, see [Kik documentation](#).

How can I use authenticated services from my bot?

For Azure Active Directory authentication, consider using the [AuthBot NuGet](#) library. For Facebook authentication examples, see the [Bot Builder SDK for .NET samples](#) on GitHub.

NOTE

If you add authentication and security functionality to your bot, you should ensure that the patterns you implement in your code comply with the security standards that are appropriate for your application.

How can I limit access to my bot to a pre-determined list of users?

Some channels, such as SMS and email, provide unscoped addresses. In these cases, messages from the user will contain the raw user ID in the `from.Id` property.

Other channels, such as Skype, Facebook, and Slack, provide either scoped or tenanted addresses in a way that

prevents a bot from being able to predict a user's ID ahead of time. In these cases, you will need to authenticate the user via a login link or shared secret in order to determine whether or not they are authorized to use the bot.

Why does my Direct Line 1.1 conversation start over after every message?

If your Direct Line conversation appears to start over after every message, the `from` property is likely missing or `null` in messages that your Direct Line client sent to the bot. When a Direct Line client sends a message with the `from` property either missing or `null`, the Direct Line service automatically allocates an ID, so every message that the client sends will appear to originate from a new, different user.

To fix this, set the `from` property in each message that the Direct Line client sends to a stable value that uniquely represents the user who is sending the message. For example, if a user is already signed-in to a webpage or app, you might use that existing user ID as the value of the `from` property in messages that the user sends.

Alternatively, you might choose to generate a random user ID on page-load or on application-load, store that ID in a cookie or device state, and use that ID as the value of the `from` property in messages that the user sends.

What causes the Direct Line 3.0 service to respond with HTTP status code 502 "Bad Gateway"?

Direct Line 3.0 returns HTTP status code 502 when it tries to contact your bot but the request does not complete successfully. This error indicates that either the bot returned an error or the request timed out. For more information about errors that your bot generates, go to the bot's dashboard within the [Bot Framework Portal](#) and click the "Issues" link for the affected channel. If you have Application Insights configured for your bot, you can also find detailed error information there.

What is the `IDialogStack.Forward` method in the Bot Builder SDK for .NET?

The primary purpose of `IDialogStack.Forward` is to reuse an existing child dialog that is often "reactive", where the child dialog (in `IDialog.StartAsync`) waits for an object `T` with some `ResumeAfter` handler. In particular, if you have a child dialog that waits for an `IMessageActivity` `T`, you can forward the incoming `IMessageActivity` (already received by some parent dialog) by using the `IDialogStack.Forward` method. For example, to forward an incoming `IMessageActivity` to a `LuisDialog`, call `IDialogStack.Forward` to push the `LuisDialog` onto the dialog stack, run the code in `LuisDialog.StartAsync` until it schedules a wait for the next message, and then immediately satisfy that wait with the forwarded `IMessageActivity`.

`T` is usually an `IMessageActivity`, since `IDialog.StartAsync` is typically constructed to wait for that type of activity. You might use `IDialogStack.Forward` to a `LuisDialog` as a mechanism to intercept messages from the user for some processing before forwarding the message to an existing `LuisDialog`. Alternatively, you can also use `DispatchDialog` with `ContinueToNextGroup` for that purpose.

You would expect to find the forwarded item in the first `ResumeAfter` handler (e.g. `LuisDialog.MessageReceived`) that is scheduled by `StartAsync`.

What is the difference between "proactive" and "reactive"?

From the perspective of your bot, "reactive" means that the user initiates the conversation by sending a message to the bot, and the bot reacts by responding to that message. In contrast, "proactive" means that the bot initiates the conversation by sending the first message to the user. For example, a bot may send a proactive message to notify a user when a timer expires or an event occurs.

How can I send proactive messages to the user?

For examples that show how to send proactive messages, see the [C# samples](#) and [Node.js samples](#) within the BotBuilder-Samples repository on GitHub.

How can I reference non-serializable services from my C# dialogs?

There are multiple options:

- Resolve the dependency through `Autofac` and `FiberModule.Key_DoNotSerialize`. This is the cleanest solution.
- Use `NonSerialized` and `OnDeserialized` attributes to restore the dependency on deserialization. This is the simplest solution.
- Do not store that dependency, so that it won't be serialized. This solution, while technically feasible, is not recommended.
- Use the reflection serialization surrogate. This solution may not be feasible in some cases and risks serializing too much.

Where is conversation state stored?

Data in the user, conversation, and private conversation property bags is stored using the Connector's `IBotState` interface. Each property bag is scoped by the bot's ID. The user property bag is keyed by user ID, the conversation property bag is keyed by conversation ID, and the private conversation property bag is keyed by both user ID and conversation ID.

If you use the Bot Builder SDK for .NET or the Bot Builder SDK for Node.js to build your bot, the dialog stack and dialog data will both automatically be stored as entries in the private conversation property bag. The C# implementation uses binary serialization, and the Node.js implementation uses JSON serialization.

The `IBotState` REST interface is implemented by two services.

- The Bot Framework Connector provides a cloud service that implements this interface and stores data in Azure. This data is encrypted at rest and does not intentionally expire.
- The Bot Framework Emulator provides an in-memory implementation of this interface for debugging your bot. This data expires when the emulator process exits.

If you want to store this data within your data centers, you can provide a custom implementation of the state service. This can be done at least two ways:

- Use the REST layer to provide a custom `IBotState` service.
- Use the Builder interfaces in the language (Node.js or C#) layer.

What is an ETag? How does it relate to bot data bag storage?

An [ETag](#) is a mechanism for [optimistic concurrency control](#). The bot data bag storage uses ETags to prevent conflicting updates to the data. An ETag error with HTTP status code 412 "Precondition Failed" indicates that there were multiple "read-modify-write" sequences executing concurrently for that bot data bag.

The dialog stack and state are stored in bot data bags. For example, you might see the "Precondition Failed" ETag error if your bot is still processing a previous message when it receives a new message for that conversation.

What causes an error with HTTP status code 412 "Precondition Failed" or HTTP status code 409 "Conflict"?

The Connector's `IBotState` service is used to store the bot data bags (i.e., the user, conversation, and private bot data bags, where the private bot data bag includes the dialog stack "control flow" state). Concurrency control in the

`IBotState` service is managed by optimistic concurrency via ETags. If there is an update conflict (due to a concurrent update to a single bot data bag) during a "read-modify-write" sequence, then:

- If ETags are preserved, an error with HTTP status code 412 "Precondition Failed" is thrown from the `IBotState` service. This is the default behavior in the Bot Builder SDK for .NET.
- If ETags are not preserved (i.e., ETag is set to `*`), then the "last write wins" policy will be in effect, which prevents the "Precondition Failed" error but risks data loss. This is the default behavior in the Bot Builder SDK for Node.js.

How can I fix "Precondition Failed" (412) or "Conflict" (409) errors?

These errors indicate that your bot processed multiple messages for the same conversation at once. If your bot is connected to services that require precisely ordered messages, you should consider locking the conversation state to make sure messages are not processed in parallel. The Bot Builder SDK for .NET provides a mechanism (class `LocalMutualExclusion` which implements `IScope`) to pessimistically serialize the handling of a single conversations with an in-memory semaphore. You could extend this implementation to use a Redis lease, scoped by the conversation address.

If your bot is not connected to external services or if processing messages in parallel from the same conversation is acceptable, you can add this code to ignore any collisions that occur in the Bot State API. This will allow the last reply to set the conversation state.

```
var builder = new ContainerBuilder();
builder
    .Register(c => new CachingBotDataStore(c.Resolve<ConnectorStore>(),
    CachingBotDataStoreConsistencyPolicy.LastWriteWins))
    .As<IBotDataStore<BotData>>()
    .AsSelf()
    .InstancePerLifetimeScope();
builder.Update(Conversation.Container);
```

Is there a limit on the amount of data I can store using the State API?

Yes, each state store (i.e., user, conversation, and private bot data bag) may contain up to 64kb of data. For more information, see [Manage State data](#).

How do I version the bot data stored through the State API?

The State service enables you to persist progress through the dialogs in a conversation so that a user can return to a conversation with a bot later without losing their position. To preserve this, the bot data property bags that are stored via the State API are not automatically cleared when you modify the bot's code. You should decide whether or not the bot data should be cleared, based upon whether your modified code is compatible with older versions of your data.

- If you want to manually reset the conversation's dialog stack and state during development of your bot, you can use the `/deleteprofile` command to delete state data. Make sure to include the leading space in this command, to prevent the channel from interpreting it.
- After your bot has been deployed to production, you can version your bot data so that if you bump the version, the associated state data is cleared. With the Bot Builder SDK for Node.js, this can be accomplished using middleware and with the Bot Builder SDK for .NET, this can be accomplished using an `IPostToBot` implementation.

NOTE

If the dialog stack cannot be deserialized correctly, due to serialization format changes or because the code has changed too much, the conversation state will be reset.

What are the possible machine-readable resolutions of the LUIS built-in date, time, duration, and set entities?

For a list of examples, see the [Pre-built entities section](#) of the LUIS documentation.

How can I use more than the maximum number of LUIS intents?

You might consider splitting up your model and calling the LUIS service in series or parallel.

How can I use more than one LUIS model?

Both the Bot Builder SDK for Node.js and the Bot Builder SDK for .NET support calling multiple LUIS models from a single LUIS intent dialog. Keep in mind the following caveats:

- Using multiple LUIS models assumes the LUIS models have non-overlapping sets of intents.
- Using multiple LUIS models assumes the scores from different models are comparable, to select the "best matched intent" across multiple models.
- Using multiple LUIS models means that if an intent matches one model, it will also strongly match the "none" intent of the other models. You can avoid selecting the "none" intent in this situation; the Bot Builder SDK for Node.js will automatically scale down the score for "none" intents to avoid this issue.

Where can I get more help on LUIS?

- [Introduction to Language Understanding Intelligent Service \(LUIS\) - Microsoft Cognitive Services](#) (video)
- [Advanced Learning Session for Language Understanding Intelligent Service \(LUIS\)](#) (video)
- [LUIS documentation](#)
- [Language Understanding Intelligent Service Forum](#)

What are some community-authored dialogs?

- [AuthBot](#) - Azure Active Directory authentication
- [BestMatchDialog](#) - Regular expression-based dispatch of user text to dialog methods

What are some community-authored templates?

- [ES6 BotBuilder](#) - ES6 Bot Builder template

Where can I get more help?

- Leverage the information in previously answered questions on [Stack Overflow](#), or post your own questions using the `botframework` tag. Please note that Stack Overflow has guidelines such as requiring a descriptive title, a complete and concise problem statement, and sufficient details to reproduce your issue. Feature requests or overly broad questions are off-topic; new users should visit the [Stack Overflow Help Center](#) for more details.
- Consult [BotBuilder issues](#) in GitHub for information about known issues with the Bot Builder SDK, or to report a new issue.
- Leverage the information in the BotBuilder community discussion on [Gitter](#).

Troubleshooting Bot Framework authentication

6/13/2017 • 9 min to read • [Edit Online](#)

This guide can help you to troubleshoot authentication issues with your bot by evaluating a series of scenarios to determine where the problem exists.

NOTE

To complete all steps in this guide, you will need to download and use the [Bot Framework Emulator](#) and must have access to the bot's registration settings in the [Bot Framework Portal](#).

App ID and password

Bot security is configured by the **Microsoft App ID** and **Microsoft App Password** that you obtain when you register your bot with the Bot Framework. These values are typically specified within the bot's configuration file and used to retrieve access tokens from the Microsoft Account service.

If you have not yet done so, [register your bot](#) to obtain a **Microsoft App ID** and **Microsoft App Password** that it can use for authentication. If you have already registered your bot but do not know its app ID and/or password, complete these steps:

1. Sign in to the [Bot Framework Portal](#).
2. Click **My Bots**.
3. Select the bot that you want to configure.
4. Click **Settings** and scroll down to **Configuration**.
5. Click **Manage Microsoft App ID and password**. A new browser tab will open to show the bot's registration settings in the Microsoft Application Registration Portal.
6. If you need the bot's app ID, copy and save the value that appears under **Application Id**.
7. If you need the bot's password, you must generate a new one. Click **Generate New Password** to generate a new password and save that value.

Step 1: Disable security and test on localhost

In this step, you will verify that your bot is accessible and functional on localhost when security is disabled.

WARNING

Disabling security for your bot may allow unknown attackers to impersonate users. Only implement the following procedure if you are operating in a protected debugging environment.

Disable security

To disable security for your bot, edit its configuration settings to remove the values for app ID and password.

If you're using the Bot Builder SDK for .NET, edit these settings in the Web.config file:

```
<appSettings>
  <add key="MicrosoftAppId" value="" />
  <add key="MicrosoftAppPassword" value="" />
</appSettings>
```

If you're using the Bot Builder SDK for Node.js, edit these values (or update the corresponding environment variables):

```
var connector = new builder.ChatConnector({
  appId: null,
  appPassword: null
});
```

Test your bot on localhost

Next, test your bot on localhost by using the Bot Framework Emulator.

1. Start your bot on localhost.
2. Start the Bot Framework Emulator.
3. Connect to your bot using the emulator.
 - Type `http://localhost:port-number/api/messages` into the emulator's address bar, where **port-number** matches the port number shown in the browser where your application is running.
 - Ensure that the **Microsoft App ID** and **Microsoft App Password** fields are both empty.
 - Click **Connect**.
4. To test connectivity to your bot, type some text into the emulator and press Enter.

If the bot responds to the input and there are no errors in the chat window, you have verified that your bot is accessible and functional on localhost when security is disabled. Proceed to [Step 2](#).

If one or more error(s) are indicated in the chat window, click the error(s) for details. Common issues include:

- The emulator settings specify an incorrect endpoint for the bot. Make sure you have included the proper port number in the URL and the proper path at the end of the URL (e.g., `/api/messages`).
- The emulator settings specify a bot endpoint that begins with `https`. On localhost, the endpoint should begin with `http`.
- The emulator settings specify a value for the **Microsoft App ID** field and/or the **Microsoft App Password** field. Both fields should be empty.
- Security has not been disabled for the bot. [Verify](#) that the bot does not specify a value for either app ID or password.

Step 2: Verify your bot's app ID and password

In this step, you will verify that the app ID and password that your bot will use for authentication are valid. (If you do not know these values, [obtain them](#) now.)

WARNING

The following instructions disable SSL verification for `login.microsoftonline.com`. Only perform this procedure on a secure network and consider changing your application's password afterward.

Issue an HTTP request to the Microsoft login service

These instructions describe how to use [cURL](#) to issue an HTTP request to the Microsoft login service. You may use an alternative tool such as Postman, just ensure that the request conforms to the Bot Framework [authentication protocol](#).

To verify that your bot's app ID and password are valid, issue the following request using [cURL](#), replacing `APP_ID` and `APP_PASSWORD` with your bot's app ID and password.

```
curl -k -X POST https://login.microsoftonline.com/botframework.com/oauth2/v2.0/token -d  
"grant_type=client_credentials&client_id=APP_ID&client_secret=APP_PASSWORD&scope=https%3A%2F%2Fapi.botframework.com%2F.default"
```

This request attempts to exchange your bot's app ID and password for an access token. If the request is successful, you will receive a JSON payload that contains an `access_token` property, amongst others.

```
{"token_type": "Bearer", "expires_in": 3599, "ext_expires_in": 0, "access_token": "eyJ0eXAJKV1Q...")}
```

If the request is successful, you have verified that the app ID and password that you specified in the request are valid. Proceed to [Step 3](#).

If you receive an error in response to the request, examine the response to identify the cause of the error. If the response indicates that the app ID or the password is invalid, [obtain the correct values](#) from the Bot Framework Portal and re-issue the request with the new values to confirm that they are valid.

Step 3: Enable security and test on localhost

At this point, you have verified that your bot is accessible and functional on localhost when security is disabled and confirmed that the app ID and password that the bot will use for authentication are valid. In this step, you will verify that your bot is accessible and functional on localhost when security is enabled.

Enable security

Your bot's security relies on Microsoft services, even when your bot is running only on localhost. To enable security for your bot, edit its configuration settings to populate app ID and password with the values that you verified in [Step 2](#).

If you're using the Bot Builder SDK for .NET, populate these settings in the Web.config file:

```
<appSettings>  
  <add key="MicrosoftAppId" value="APP_ID" />  
  <add key="MicrosoftAppPassword" value="PASSWORD" />  
</appSettings>
```

If you're using the Bot Builder SDK for Node.js, populate these settings (or update the corresponding environment variables):

```
var connector = new builder.ChatConnector({  
  appId: 'APP_ID',  
  appPassword: 'PASSWORD'  
});
```

Test your bot on localhost

Next, test your bot on localhost by using the Bot Framework Emulator.

1. Start your bot on localhost.
2. Start the Bot Framework Emulator.
3. Connect to your bot using the emulator.
 - Type `http://localhost:port-number/api/messages` into the emulator's address bar, where **port-number** matches the port number shown in the browser where your application is running.
 - Enter your bot's app ID into the **Microsoft App ID** field.
 - Enter your bot's password into the **Microsoft App Password** field.

- Click **Connect**.
4. To test connectivity to your bot, type some text into the emulator and press Enter.
- If the bot responds to the input and there are no errors in the chat window, you have verified that your bot is accessible and functional on localhost when security is enabled. Proceed to [Step 4](#).
- If one or more error(s) are indicated in the chat window, click the error(s) for details. Common issues include:
- The emulator settings specify an incorrect endpoint for the bot. Make sure you have included the proper port number in the URL and the proper path at the end of the URL (e.g., `/api/messages`).
 - The emulator settings specify a bot endpoint that begins with `https`. On localhost, the endpoint should begin with `http` .
 - In the emulator settings, the **Microsoft App ID** field and/or the **Microsoft App Password** do not contain valid values. Both fields should be populated and each field should contain the corresponding value that you verified in [Step 2](#).
 - Security has not been enabled for the bot. [Verify](#) that the bot configuration settings specify values for both app ID and password.

Step 4: Test your bot in the cloud

At this point, you have verified that your bot is accessible and functional on localhost when security is disabled, confirmed that your bot's app ID and password are valid, and verified that your bot is accessible and functional on localhost when security is enabled. In this step, you will deploy your bot to the cloud and verify that it is accessible and functional there with security enabled.

Deploy your bot to the cloud

The Bot Framework requires that bots be accessible from the internet, so you must deploy your bot to a cloud hosting platform such as Azure. Be sure to enable security for your bot prior to deployment, as described in [Step 3](#).

NOTE

If you do not already have a cloud hosting provider, you can register for a [free trial](#) of Azure.

If you deploy your bot to Azure, SSL will automatically be configured for your application, thereby enabling the **HTTPS** endpoint that the Bot Framework requires. If you deploy to another cloud hosting provider, be sure to verify that your application is configured for SSL so that the bot will have an **HTTPS** endpoint.

Test your bot

To test your bot in the cloud with security enabled, complete the following steps.

1. Ensure that your bot has been successfully deployed and is running.
2. Sign in to the [Bot Framework Portal](#).
3. Click **My Bots**.
4. Select the bot that you want to test.
5. Click **Test** to open the bot in an embedded web chat control.
6. To test connectivity to your bot, type some text into the web chat control and press Enter.

If an error is indicated in the chat window, use the error message to determine the cause of the error. Common issues include:

- The **Messaging endpoint** specified on the **Settings** page for your bot in the Bot Framework Portal is incorrect. Make sure you have included the proper path at the end of the URL (e.g., `/api/messages`).
- The **Messaging endpoint** specified on the **Settings** page for your bot in the Bot Framework Portal does not

begin with `https` or is not trusted by the Bot Framework. Your bot must have a valid, chain-trusted certificate.

- The bot is configured with missing or incorrect values for app ID or password. [Verify](#) that the bot configuration settings specify valid values for app ID and password.

If the bot responds appropriately to the input, you have verified that your bot is accessible and functional in the cloud with security enabled. At this point, your bot is ready to securely [connect to a channel](#) such as Skype, Facebook Messenger, Direct Line, and others.

Additional resources

If you are still experiencing issues after completing the steps above, you can:

- [Debug your bot in the cloud](#) using the Bot Framework Emulator and [ngrok](#).
- Use a proxying tool like [Fiddler](#) to inspect HTTPS traffic to and from your bot. *Fiddler is not a Microsoft product.*
- Review the [Bot Connector authentication guide](#) to learn about the authentication technologies that the Bot Framework uses.
- Solicit help from others by using the Bot Framework [support](#) resources.

Bot Framework reference

10/24/2017 • 1 min to read • [Edit Online](#)

The Bot Framework includes comprehensive reference documentation for the Bot Builder SDK for .NET, the Bot Builder SDK for Node.js, and the Bot Framework REST APIs.

Bot Builder SDK for .NET

To learn about the structure of the Bot Builder SDK for .NET, see the [Bot Builder SDK for .NET reference documentation](#).

You can also explore the Bot Builder SDK for .NET [source code](#) on GitHub.

Bot Builder SDK for Node.js

To learn about the structure of the Bot Builder SDK for Node.js, see the [Bot Builder SDK for Node.js reference documentation](#).

You can also explore the Bot Builder SDK for Node.js [source code](#) on GitHub.

Bot Framework REST APIs

To learn about the Bot Connector service and the Bot State service, see the [Bot Framework REST API reference documentation](#).

To learn about Direct Line API v3.0, see the [Direct Line 3.0 reference documentation](#). To learn about Direct Line API v1.1, see the [Direct Line 1.1 reference documentation](#).

4 min to read •

Bot Framework Frequently Asked Questions

8/9/2017 • 5 min to read • [Edit Online](#)

This article contains answers to some frequently asked questions about the Bot Framework.

Background and availability

Why did Microsoft develop the Bot Framework?

While the Conversation User Interface (CUI) is upon us, at this point few developers have the expertise and tools needed to create new conversational experiences or enable existing applications and services with a conversational interface their users can enjoy. We have created the Bot Framework to make it easier for developers to build and connect great bots to users, wherever they converse, including on Microsoft's premier channels.

Who are the people behind the Bot Framework?

In the spirit of One Microsoft, the Bot Framework is a collaborative effort across many teams, including Microsoft Technology and Research, Microsoft's Applications and Services Group and Microsoft's Developer Experience teams.

When did work begin on the Bot Framework?

The core Bot Framework work has been underway since the summer of 2015.

Is the Bot Framework publicly available now?

Yes. The Bot Framework was released in preview on March 30th of 2016 in conjunction with Microsoft's annual developer conference [/build](#).

How long will the Bot Framework be in preview? Can I start building/shipping products based on a preview framework?

The Bot Framework is currently in preview. As indicated at Build 2016, Microsoft is making significant investments in Conversation as a Platform - among those investments is the Bot Framework. Building upon a preview offering is, of course, at your discretion.

What is the future of the Bot Framework?

We are excited to provide initial availability of the Bot Framework at [/build 2016](#) and plan to continuously improve the framework with additional tools, samples, and channels. The [Bot Builder SDK](#) is an open source SDK hosted on GitHub and we look forward to the contributions of the community at large. [Feedback](#) as to what you'd like to see is welcome.

Channels

When will you add more conversation experiences to the Bot Framework?

We plan on making continuous improvements to the Bot Framework, including additional channels, but cannot provide a schedule at this time.

If you would like a specific channel added to the framework, [let us know](#).

I have a communication channel I'd like to be configurable with Bot Framework. Can I work with Microsoft to do that?

We have not provided a general mechanism for developers to add new channels to Bot Framework, but you can connect your bot to your app via the [Direct Line API](#). If you are a developer of a communication channel and would like to work with us to enable your channel in the Bot Framework [we'd love to hear from you](#).

If I want to create a bot for Skype, what tools and services should I use?

The Bot Framework is designed to build, connect, and deploy high quality, responsive, performant and scalable bots for Skype and many other channels. The SDK can be used to create text/sms, image, button and card-capable bots (which constitute the majority of bot interactions today across conversation experiences) as well as bot interactions which are Skype-specific such as rich audio and video experiences.

If you already have a great bot and would like to reach the Skype audience, your bot can easily be connected to Skype (or any supported channel) via the Bot Builder for REST API (provided it has an internet-accessible REST endpoint).

Is it possible for me to build a bot using the Bot Framework/SDK that is a “private or enterprise-only” bot that is only available inside my company?

At this point, we do not have plans to enable a private instance of the Bot Directory, but we are interested in exploring ideas like this with the developer community.

Security and Privacy

Do the bots registered with the Bot Framework collect personal information? If yes, how can I be sure the data is safe and secure? What about privacy?

Each bot is its own service, and developers of these services are required to provide Terms of Service and Privacy Statements per their Developer Code of Conduct. You can access this information from the bot's card in the Bot Directory.

to provide the I/O service, the Bot Framework transmits your message and message content (including your ID), from the chat service you used to the bot.

How do you ban or remove bots from the service?

Users have a way to report a misbehaving bot via the bot's contact card in the directory. Developers must abide by Microsoft terms of service to participate in the service.

Which specific URLs do I need to whitelist in my corporate firewall to access bot services?

You'll need to whitelist the following URLs in your corporate firewall:

- login.botframework.com (Bot authentication)
- login.microsoftonline.com (Bot authentication)
- westus.api.cognitive.microsoft.com (for Luis.ai NLP integration)
- state.botframework.com (Bot state storage for prototyping)
- cortanabfchanneleastus.azurewebsites.net (Cortana channel)
- cortanabfchannelwestus.azurewebsites.net (Cortana Channel)
- *.botFramework.com (channels)

Related Services

How does the Bot Framework relate to Cognitive Services?

Both the Bot Framework and [Cognitive Services](#) are new capabilities introduced at [Microsoft Build 2016](#) that will also be integrated into Cortana Intelligence Suite at GA. Both these services are built from years of research and use in popular Microsoft products. These capabilities combined with 'Cortana Intelligence' enable every organization to take advantage of the power of data, the cloud and intelligence to build their own intelligent systems that unlock new opportunities, increase their speed of business and lead the industries in which they serve their customers.

What is Cortana Intelligence?

Cortana Intelligence is a fully managed Big Data, Advanced Analytics and Intelligence suite that transforms your data into intelligent action.

It is a comprehensive suite that brings together technologies founded upon years of research and innovation

throughout Microsoft (spanning advanced analytics, machine learning, big data storage and processing in the cloud) and:

- Allows you to collect, manage and store all your data that can seamlessly and cost effectively grow over time in a scalable and secure way.
- Provides easy and actionable analytics powered by the cloud that allow you to predict, prescribe and automate decision making for the most demanding problems.
- Enables intelligent systems through cognitive services and agents that allow you to see, hear, interpret and understand the world around you in more contextual and natural ways.

With Cortana Intelligence, we hope to help our enterprise customers unlock new opportunities, increase their speed of business and be leaders in their industries.

What is the Direct Line channel?

Direct Line is a REST API that allows you to add your bot into your service, mobile app, or webpage.

You can write a client for the Direct Line API in any language. Simply code to the [Direct Line protocol](#), generate a secret in the Direct Line configuration page, and talk to your bot from wherever your code lives.

Direct Line is suitable for:

- Mobile apps on iOS, Android, and Windows Phone, and others
- Desktop applications on Windows, OSX, and more
- Webpages where you need more customization than the [embeddable Web Chat channel](#) offers
- Service-to-service applications

Bot Framework additional resources

10/12/2017 • 1 min to read • [Edit Online](#)

These resources provide additional information and support for developing bots with the Bot Framework.

IMPORTANT

Please use one of these resources for support rather than posting comments on this article. This article is not monitored for support requests.

SUPPORT TYPE	CONTACT
Community support	Questions can be posted at Stack Overflow using the <code>botframework</code> tag. Please note that Stack Overflow has guidelines such as requiring a descriptive title, a complete and concise problem statement, and sufficient details to reproduce your issue. Feature requests or overly broad questions are off-topic; new users should visit the Stack Overflow Help Center for more details.
Community chat group	Gitter.IM
Using a bot	Contact the bot's developer through their publisher e-mail
Bot Builder SDK issues/suggestions	Use the issues tab on the GitHub repo
Documentation issues	Submit an issue to the Bot Framework documentation GitHub repo.
Documentation updates	Click the Edit link on an article and submit a pull request to the Bot Framework documentation GitHub repo .
Reporting abuse	Contact us at bf-reports@microsoft.com

ID fields in the Bot Framework

8/7/2017 • 5 min to read • [Edit Online](#)

This guide describes the characteristics of ID fields in the Bot Framework.

Channel ID

Every Bot Framework channel is identified by a unique ID.

Example: `"channelId": "skype"`

Channel IDs serve as namespaces for other IDs. Runtime calls in the Bot Framework protocol must take place within the context of a channel; the channel gives meaning to the conversation and account IDs used when communicating.

By convention all channel IDs are lowercase. Channels guarantee that the channel IDs they emit have consistent casing, and thus bots may use ordinal comparisons to establish equivalence.

Rules for channel IDs

- Channel IDs are case-sensitive.

Bot Handle

Every bot that has been registered with the Bot Framework has a bot handle.

Example: `FooBot`

A bot handle represents a bot's registration with the online Bot Framework. This registration is associated with an HTTP webhook endpoint and registrations with channels.

The Bot Framework dev portal ensures uniqueness of bot handles. The portal performs a case-insensitive uniqueness check (meaning that case variations of bot handle are treated as a single handle) although this is a characteristic of the dev portal, and not necessarily the bot handle itself.

Rules for bot handles

- Bot handles are unique (case-insensitive) within the Bot Framework.

App ID

Every bot that has been registered with the Bot Framework has an App ID.

NOTE

Previously, apps were commonly referred to as "MSA Apps" or "MSA/AAD Apps." Apps are now more commonly referred to simply as "apps", but some protocol elements may refer to apps as "MSA Apps" in perpetuity.

Example: `"msaAppId": "353826a6-4557-45f8-8d88-6aa0526b8f77"`

An app represents a registration with the Identity team's App portal, and serves as the service-to-service identity mechanism within the Bot Framework runtime protocol. Apps may have other non-bot associations, such as websites and mobile/desktop applications.

Every registered bot has exactly one app. Although it's not possible for a bot owner to independently change which

app is associated with their bot, the Bot Framework team can do so in a small number of exceptional cases.

Bots and channels may use app IDs to uniquely identify a registered bot.

App IDs are guaranteed to be GUIDs. App IDs should be compared without case sensitivity.

Rules for app IDs

- App IDs are unique (GUID comparison) within the Microsoft App platform.
- Every bot has exactly one corresponding app.
- Changing which app a bot is associated with requires the assistance of the Bot Framework team.

Channel Account

Every bot and user has an account within each channel. The account contains an identifier (`id`) and other informative bot non-structural data, like an optional name.

Example: `"from": { "id": "john.doe@contoso.com", "name": "John Doe" }`

This account describes the address within the channel where messages may be sent and received. In some cases, these registrations exist within a single service (e.g., Skype, Facebook). In others, they are registered across many systems (email addresses, phone numbers). In more anonymous channels (e.g., Web Chat), the registration may be ephemeral.

Channel accounts are nested within channels. A Facebook account, for example, is simply a number. This number may have a different meaning in other channels, and it doesn't have meaning outside all channels.

The relationship between channel accounts and users (actual people) depends on conventions associated with each channel. For example, an SMS number typically refers to one person for a period of time, after which the number may be transferred to someone else. Conversely, a Facebook account typically refers to one person in perpetuity, although it is not uncommon for two people to share a Facebook account.

In most channels, it's appropriate to think of a channel account as a kind of mailbox where messages can be delivered. It's typical for most channels to allow multiple address to map to a single mailbox; for example, `"jdoe@contoso.com"` and `"john.doe@service.contoso.com"` may resolve to the same inbox. Some channels go a step further and alter the account's address based on which bot is accessing it; for example, both Skype and Facebook alter user IDs so every bot has a different address for sending and receiving messages.

While it's possible in some cases to establish equivalency between addresses, establishing equivalency between mailboxes and equivalency between people requires knowledge of the conventions within the channel, and is in many cases not possible.

A bot is informed of its channel account address via the `recipient` field on activities sent to the bot.

Rules for channel accounts

- Channel accounts have meaning only within their associated channel.
- More than one ID may resolve to the same account.
- Ordinal comparison may be used to establish that two IDs are the same.
- There is generally no comparison that can be used to identify whether two different IDs resolve to the same account, bot or person.
- The stability of associations between IDs, accounts, mailboxes, and people depends on the channel.

Conversation ID

Messages are sent and received in the context of a conversation, which is identifiable by ID.

Example: `"conversation": { "id": "1234" }`

A conversation contains an exchange of messages and other activities. Every conversation has zero or more activities, and every activity appears in exactly one conversation. Conversations may be perpetual, or may have distinct starts and ends. The process of creating, modifying, or ending a conversation occurs within the channel (i.e., a conversation exists when the channel is aware of it) and the characteristics of these processes are established by the channel.

The activities within a conversation are sent by users and bots. The definition for which users "participate" in a conversation varies by channel, and can theoretically include present users, users who have ever received a message, users who sent a message.

Several channels (e.g., SMS, Skype, and possibly others) have the quirk that the conversation ID assigned to a 1:1 conversation is the remote channel account ID. This quirk has two side-effects:

1. The conversation ID is subjective based on who is viewing it. If Participants A and B are talking, participant A sees the conversation ID to be "B" and participant B sees the conversation ID to be "A."
2. If the bot has multiple channel accounts within this channel (for example, if the bot has two SMS numbers), the conversation ID is not sufficient to uniquely identify the conversation within the bot's field of view.

Thus, a conversation ID does not necessarily uniquely identify a single conversation within a channel even for a single bot.

Rules for conversation IDs

- Conversations have meaning only within their associated channel.
- More than one ID may resolve to the same conversation.
- Ordinal equality does not necessarily establish that two conversation IDs are the same conversation, although in most cases, it does.

Activity ID

Activities are sent and received within the Bot Framework protocol, and these are sometimes identifiable.

Example: `"id": "5678"`

Activity IDs are optional and employed by channels to give the bot a way to reference the ID in subsequent API calls, if they are available:

- Replying to a particular activity
- Querying for the list of participants at the activity level

Because no further use cases have been established, there are no additional rules for the treatment of activity IDs.

Application Insights keys

5/25/2017 • 1 min to read • [Edit Online](#)

Azure Application Insights displays data about your application in a Microsoft Azure resource. To add telemetry to your bot you need an Azure subscription and an Application Insights resource created for your bot. From this resource, you can obtain the three keys to configure your bot:

1. Instrumentation key
2. Application ID
3. API key

To configure these three keys, you must [register your bot](#). Then, locate these fields in your bot's settings page under the section labeled **Analytics**. The following screenshot shows the Analytics fields.

The screenshot shows the Bot Framework interface with the following navigation bar:

- Bot Framework
- My bots
- Documentation
- Blog

The main area shows the bot type as "Speech bot" and has tabs for CHANNELS, ANALYTICS, and SETTINGS. The ANALYTICS tab is selected, displaying the following fields:

- AppInsights Instrumentation key: A text input field containing "Instrumentation key (Azure App Insights key)".
- AppInsights API key: A text input field containing "API key (User-Generated App Insights API key)".
- AppInsights Application ID: A text input field containing "Application ID (App Insights Application ID)".

Instrumentation key

To get the Instrumentation key, do the following:

1. From the portal.azure.com, under the Monitor section, create a new **Application Insights** resource (or use an existing one).

The screenshot shows the Azure Monitor - Application Insights interface. On the left, there's a sidebar with various monitoring icons. One icon, labeled 'Application Insights', is highlighted with a red box and has a red arrow pointing to it from the left. Another red arrow points from the 'Application Insights' section in the sidebar to the same section in the main content area. The main content area displays a table of 'Subscriptions'. The first row in the table is also highlighted with a red box and has a red arrow pointing to it from the right.

2. From the list of Application Insights resources, click the Application Insight resource you just created.
3. Click **Overview**.
4. Expand the **Essentials** block and find the **Instrumentation Key**.

The screenshot shows the 'ContosoFlowersSupportBot' Application Insights Overview page. On the left, there's a sidebar with 'Overview', 'Activity log', 'Access control (IAM)', and 'Tags'. The 'Overview' section is highlighted with a red box and has a red arrow pointing to it from the left. To its right is the 'Essentials' block, which is also highlighted with a red box and has a red arrow pointing to it from the left. The 'Instrumentation Key' field is shown under the 'Type ASP.NET' section, and it is also highlighted with a red box and has a red arrow pointing to it from the right.

5. Copy the **Instrumentation Key** and paste it to the **AppInsights Instrumentation Key** field of your bot's settings.

Application ID

To get the Application ID, do the following:

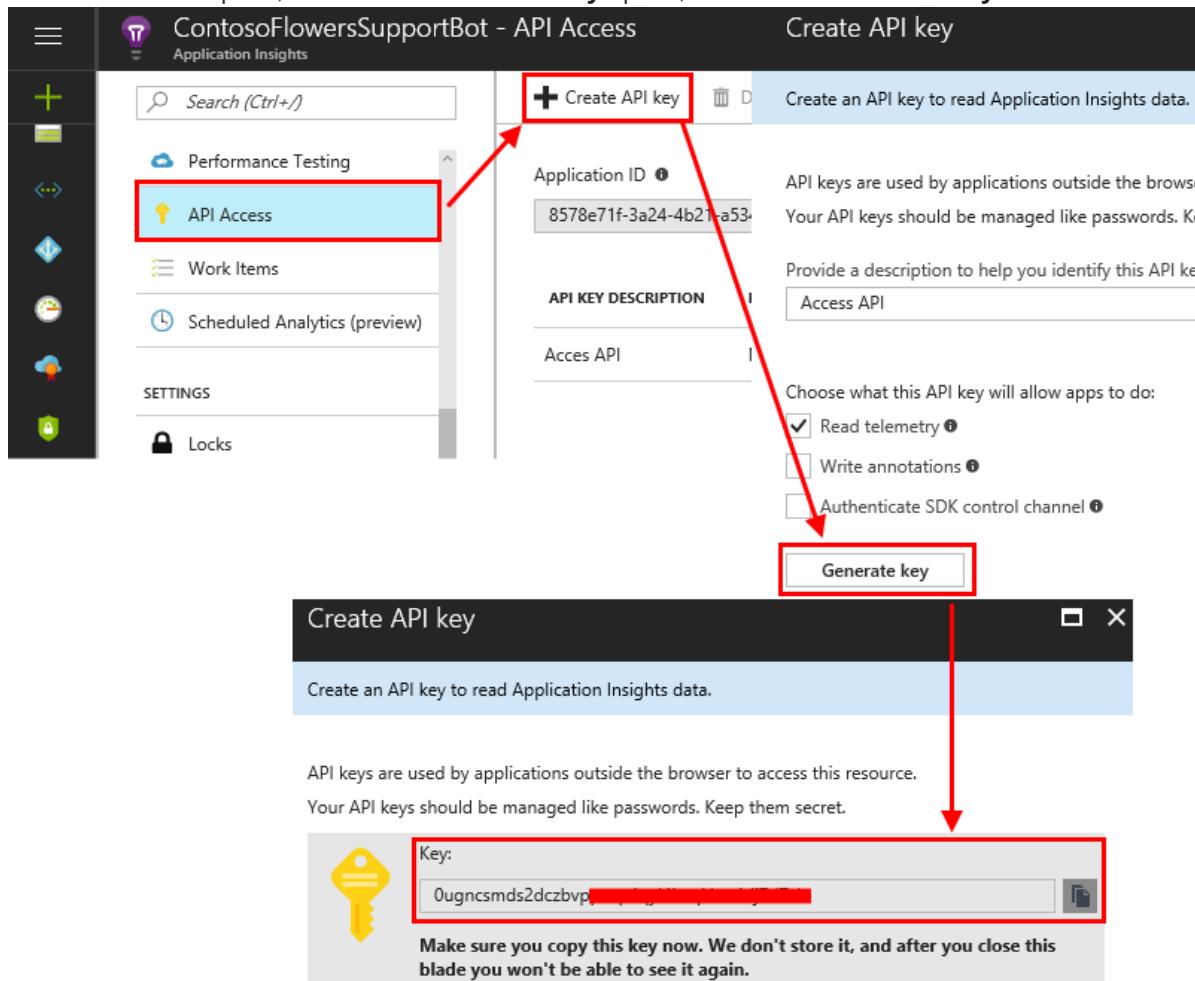
1. From your Application Insights resource, click **API Access**.
2. Copy the **Application ID** and paste it to the **AppInsights Application ID** field of your bot's settings.

The screenshot shows the 'ContosoFlowersSupportBot - API Access' page. On the left, there's a sidebar with 'Performance Testing', 'API Access', 'Work Items', and 'Scheduled Analytics (preview)'. The 'API Access' section is highlighted with a red box and has a red arrow pointing to it from the left. To its right is the 'Application ID' field, which is also highlighted with a red box and has a red arrow pointing to it from the right. The value '8578e71f-3a24-4b21-a534-dae2c3b4abb...' is displayed in the field.

API key

To get the API key, do the following:

1. From the Application Insights resource, click **API Access**.
2. Click **Create API Key**.
3. Enter a short description, check the **Read telemetry** option, and click the **Generate key** button.



WARNING

Copy this **API key** and save it because this key will never be shown to you again. If you lose this key, you have to create a new one.

4. Copy the API key to the **AppInsights API key** field of your bot's settings.

Additional resources

For more information on how to connect these fields into your bot's settings, see [Enable analytics](#).

Upgrade your bot to Bot Framework API v3

8/7/2017 • 6 min to read • [Edit Online](#)

At Build 2016 Microsoft announced the Microsoft Bot Framework and its initial iteration of the Bot Connector API, along with Bot Builder and Bot Connector SDKs. Since then, we've been collecting your feedback and actively working to improve the REST API and SDKs.

In July 2016, Bot Framework API v3 was released and Bot Framework API v1 was deprecated. Bots that use API v1 ceased to function on Skype in December 2016 and on all remaining channels on February 23, 2017. If you created a bot using API v1 and want to make it functional again, you must upgrade it to API v3 by following the instructions in this article. To ensure that you understand the upgrade process from end-to-end, read through this article completely before you begin.

Step 1: Get your App ID and password from the Bot Framework Portal

Sign in to the [Bot Framework Portal](#), click **My bots**, then select your bot to open its dashboard. Next, click the **SETTINGS** link that is located near the top-right corner of the page.

Within the **Configuration** section of the settings page, examine the contents of the **App ID** field and proceed with next steps depending on whether or not the **App ID** field is already populated.

Case 1: App ID field is already populated

If the **App ID** field is already populated, complete these steps:

1. Click **Manage Microsoft App ID and password**.

The screenshot shows the 'Configuration' section of the Bot Framework Portal. It includes fields for 'Messaging endpoint' (containing 'https URL') and a button labeled 'Manage Microsoft App ID and password'. Below these, there is a placeholder text 'Paste your app ID below to continue' with a corresponding input field containing the value 'fc41305c-99aa-4687-a4bf-b33d1bcb92db'.

2. Click **Generate New Password**.

Name

Yourbot

Application Id

b46216dd-e43f-4661-8c3d-ea983c637ff9

Application Secrets Learn More

[Generate New Password](#)

[Generate New Key Pair](#)

Type

Password/Public Key

Created

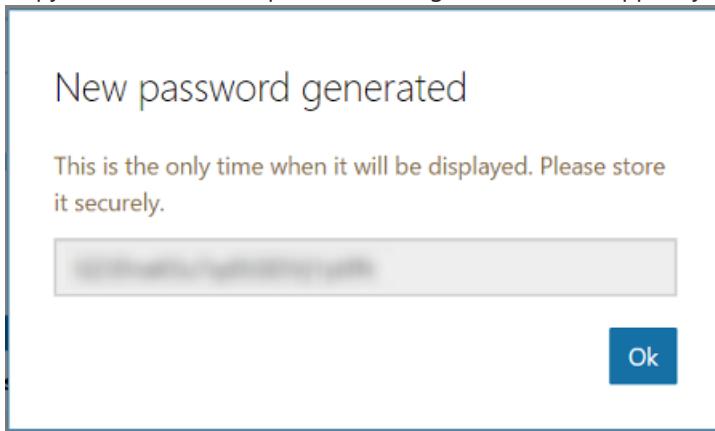
Password

Vuq*****

Jun 30, 2016 2:15:24 PM

[Delete](#)

3. Copy and save the new password along with the MSA App ID; you will need these values in the future.



Case 2: App ID field is empty

If the **App ID** field is empty, complete these steps:

1. Click **Create Microsoft App ID and password**.

Configuration

Bot Framework version

Version 1.0

Messaging endpoint

App secret

.....

Show

Version 3.0

Messaging endpoint

https URL

* App ID

[Create Microsoft App ID and password](#)

You need an App ID and password to authenticate your bot. Paste the password into your bot configuration file. [Learn more](#).

IMPORTANT

Do not select the **Version 3.0** radio button yet. You will do this later, after you have [updated your bot code](#).

2. Click **Generate a password to continue**.

Generate App ID and password

App name

ABCBot

App ID

25e253bf-e5ca-4cc8-b7cc-a39a375c483c

[Generate a password to continue](#)

3. Copy and save the new password along with the MSA App Id; you will need these values in the future.

New password generated

This is the only time when it will be displayed. Please store it securely.

[Ok](#)

4. Click **Finish and go back to Bot Framework**.

Generate App ID and password

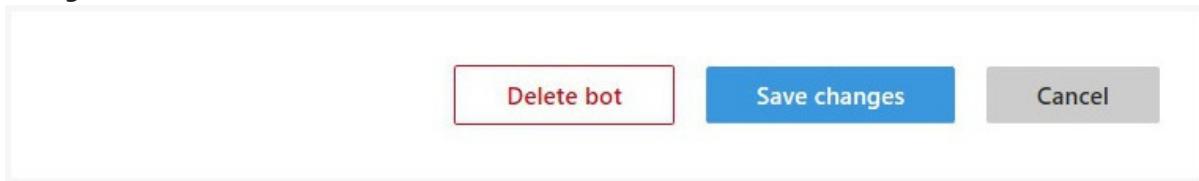
App name
ABCBot

App ID
25e253bf-e5ca-4cc8-b7cc-a39a375c483c

Password [Learn More](#)
AVt*****
[Delete](#)

[Finish and go back to Bot Framework](#)

5. Back on the bot settings page in the Bot Framework Portal, scroll to the bottom of the page and click **Save changes**.



Step 2: Update your bot code to version 3.0

To update your bot code to version 3.0, complete these steps:

1. Update to the latest version of the [Bot Builder SDK](#) for your bot's language.
2. Update your code to apply the necessary changes, according the guidance below.
3. Use the [Bot Framework Emulator](#) to test your bot locally and then in the cloud.

The following sections describe the key differences between API v1 and API v3. After you have updated your code to API v3, you can finish the upgrade process by [updating your bot settings](#) in the Bot Framework Portal.

BotBuilder and Connector are now one SDK

Instead of having to install separate SDKs for the Builder and Connector by using multiple NuGet packages (or NPM modules), you can now get both libraries in a single Bot Builder SDK:

- Bot Builder SDK for .NET: `Microsoft.Bot.Builder` NuGet package
- Bot Builder SDK for Node.js: `botbuilder` NPM module

The standalone `Microsoft.Bot.Connector` SDK is now obsolete and is no longer being maintained.

Message is now Activity

The `Message` object has been replaced with the `Activity` object in API v3. The most common type of activity is **message**, but there are other activity types that can be used to communicate various types of information to a bot or channel. For more information about messages, see [Create messages](#) and [Send and receive activities](#).

Activity types & events

Some events have been renamed and/or refactored in API v3. In addition, a new `ActivityTypes` enumeration has been added to the Connector to eliminate the need to remember specific activity types.

- The `conversationUpdate` Activity type replaces Bot/User Added/Removed To/From Conversation with a single method.
- The new `typing` Activity type enables your bot to indicate that it is compiling a response and to know when the user is typing a response.

- The new `contactRelationUpdate` Activity type enables your bot to know if it has been added to or removed from user's contact list.

When your bot receives a `conversationUpdate` activity, the `MembersRemoved` property and `MembersAdded` property will indicate who was added to or removed from the conversation. When your bot receives a `contactRelationUpdate` activity, the `Action` property will indicate whether the user added the bot to or removed the bot from their contact list. For more information about activity types, see [Activities overview](#).

Addressing messages

Where the sender, recipient, and channel information is specified within a message has changed slightly in API v3:

API V1 FIELD	API V3 FIELD
<code>From</code> object	<code>From</code> object
<code>To</code> object	<code>Recipient</code> object
<code>ChannelConversationID</code> property	<code>Conversation</code> object
<code>ChannelId</code> property	<code>ChannelId</code> property

For more information about addressing messages, see [Send and receive activities](#).

Sending replies

In Bot Framework API v3, all replies to the user will be sent asynchronously over a separately initiated HTTP request rather than inline with the HTTP POST for the incoming message to the bot. Since no message will be returned inline to the user through the Connector, the return type of your bot's post method will be `HttpResponseMessage`. This means that your bot does not synchronously "return" the string that you wish to send to the user, but instead sends a reply message at any point in your code instead of having to reply back as a response to the incoming POST. These two methods will both send a message to a conversation:

- `SendToConversation`
- `ReplyToConversation`

The `SendToConversation` method will append the specified message to the end of the conversation, while the `ReplyToConversation` method will (for conversations that support it) add the specified message as a direct reply to a prior message in the conversation. For more information about these methods, see [Send and receive activities](#).

Starting conversations

In Bot Framework API v3, you can start a conversation by either using the new method `CreateDirectConversation` to start a private conversation with a single user or by using the new method `CreateConversation` to start a group conversation with multiple users. For more information about starting conversations, see [Send and receive activities](#).

Attachments and options

Bot Framework API v3 introduces a more robust implementation of attachments and cards. The `options` type is no longer supported in API v3 and has instead been replaced by cards. For more information about adding attachments to messages using .NET, see [Add media attachments to messages](#) and [Add rich card attachments to messages](#).

Bot data storage (bot state)

In Bot Framework API v1, the API for managing bot state data was folded into the messaging API. In Bot Framework API v3, these APIs are separate. Now, you must use the Bot State service to get state data (instead of assuming that it will be included within the `Message` object) and to store state data (instead of passing it as part of the `Message`

object). For information about managing bot state data using the Bot State service, [Manage state data](#).

Web.config changes

Bot Framework API v1 stored the authentication properties with these keys in **Web.Config**:

- `AppID`
- `AppSecret`

Bot Framework API v3 stores the authentication properties with these keys in **Web.Config**:

- `MicrosoftAppID`
- `MicrosoftAppPassword`

Step 3: Update your bot settings in the Bot Framework Portal

After you have upgraded your bot code to API v3 and deployed it to the cloud, finish the upgrade process by completing these steps:

1. Sign in to the [Bot Framework Portal](#).
2. Click **My bots** and select your bot to open its dashboard.
3. Click the **SETTINGS** link that is located near the top-right corner of the page.
4. Under **Version 3.0** within the **Configuration** section, paste your bot's endpoint into the **Messaging endpoint** field.

The screenshot shows the 'Configuration' section of the Bot Framework Portal. It includes fields for 'Bot Framework version' (set to 'Version 1.0'), 'Messaging endpoint' (containing a blurred URL), 'App secret' (containing a blurred string of asterisks), and a 'Show' link. Below these, there is a 'Version 3.0' section with a radio button selected. A red arrow points to the 'Messaging endpoint' input field in this section, which also contains a blurred URL. Further down, there is a field for '* App ID' with a blurred URL and a 'Create Microsoft App ID and password' button. A note at the bottom states: 'You need an App ID and password to authenticate your bot. Paste the password into your bot configuration file. [Learn more](#)'.

5. Select the **Version 3.0** radio button.

Configuration

Bot Framework version

Version 1.0

Messaging endpoint

App secret

.....

Show

Version 3.0

Messaging endpoint

* App ID

[Create Microsoft App ID and password](#)

You need an App ID and password to authenticate your bot. Paste the password into your bot configuration file. [Learn more.](#)

6. Scroll to the bottom of the page and click **Save changes**.

[Delete bot](#)

[Save changes](#)

[Cancel](#)

Bot Framework User-Agent requests

9/13/2017 • 1 min to read • [Edit Online](#)

If you're reading this message, you've probably received a request from a Microsoft Bot Framework service. This guide will help you understand the nature of these requests and provide steps to stop them, if so desired.

If you received a request from our service, it likely had a User-Agent header formatted similar to the string below:

```
User-Agent: BF-DirectLine/3.0 (Microsoft-BotFramework/3.0 +https://botframework.com/ua)
```

The most important part of this string is the **Microsoft-BotFramework** identifier, which is used by the Microsoft Bot Framework, a collection of tools and services that allows independent software developers to create and operate their own bots.

If you're a bot developer, you may already know why these requests are being directed to your service. If not, continue reading to learn more.

Why is Microsoft contacting my service?

The Bot Framework connects users on chat services like Skype and Facebook Messenger to bots, which are web servers with REST APIs running on internet-accessible endpoints. The HTTP calls to bots (also called webhook calls) are sent only to URLs specified by a bot developer who registered with the Bot Framework developer portal.

If you're receiving unsolicited requests from Bot Framework services to your web service, it is likely because a developer has either accidentally or knowingly entered your URL as the webhook callback for their bot.

To stop these requests

For assistance in stopping undesired requests from reaching your web service, please contact bf-reports@microsoft.com.