Bot Concepts

Bot Concepts

Bot Concepts

At its most fundamental level, a bot (a shortened version of the word “robot”) is a term used to describe **a program or process that operates as an agent for a user** (or another program) and attempts to simulate a human activity. However, in the context of "Conversation as a Platform" what we're really talking about is a **conversation bot** (a.k.a. “chatbot”, "talkbot", “chatterbot”, "Bot", “IM bot”, "interactive agent", or even the unwieldy “Artificial Conversational Entity”. **A conversation bot is a computer program, service, or process that conducts a conversation via auditory or textual methods**.

Bot Concepts

Start of transcript. Skip to the end.

0:00 / 1:42 1.0x

Try and answer the question of, what are bots, and
more specifically conversation bots, you may be expecting some
sort of complex, complicated, convoluted answer.
But let's just simplify it.
In general, bots, specifically conversation bots,
are essentially just decision trees where the user navigates
based on an identified range of choices, typically by leveraging
artificial intelligence and machine learning.

Video
Download video file

Transcripts
Download SubRip (.srt) file
Download Text (.txt) file

As a rule, conversation bots are designed to convincingly simulate how a human would behave as a “conversational partner”, and thereby pass the infamous Turing test. Many bots use sophisticated natural language processing systems, such as Microsoft's Language Understanding Intelligent Service, however other bot implementations simply scan for keywords within the input, then pull a reply with the most matching keywords, or the most similar wording pattern, from a database.

💡This course will be cover Microsoft Language Understanding Intelligent Service (or LUIS, for short) in Module 3: Language Understanding.

Either way, bots (or more specifically conversation bots) have quickly becoming an integral part of digital experiences. As part of Microsoft's “Conversation as a Platform”, the Microsoft Bot Framework provides tools for developers to easily create, publish, and manage bot solutions—including automatic translation to more than 30 languages, user and conversation state management, debugging tools, an embeddable web chat control. direct line API methods, and even mechanisms for easy bot discovery.

Quick Check (True or False)

1.0/1.0 point (graded)

A conversation bot is a computer program, service, or process that conducts a conversation via auditory or textual methods. (True or False)

[✓ True] ✓

False

Submit You have used 1 of 2 attempts

Learn About Verified Certificates

Design Principles

Design Principles

The Bot Framework enables developers to create compelling bot experiences that solve a variety of both consumer needs and business challenges. By learning the concepts provided in this course, you'll become equipped to design bots that align with best practices and capitalize on "lessons learned" by other designers and developers.

Design Principles

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 using a 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, the real question becomes: 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.

💡As with all software design and development—including the design and implementation of bots—**a user's overall experience should be the top priority**.

Quick Check (True or False)

1.0/1.0 point (graded)

The top priority of a bot experience should be the color selections for bot icons. (True or False)

True [✓ False] ✓

Submit You have used 1 of 2 attempts

Learn About Verified Certificates

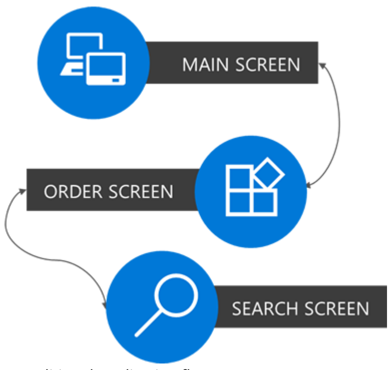
Conversation Flow

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 start their experience, that provides navigation leading 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 another, separate dialog to contain the logic that helps the user create a new order.

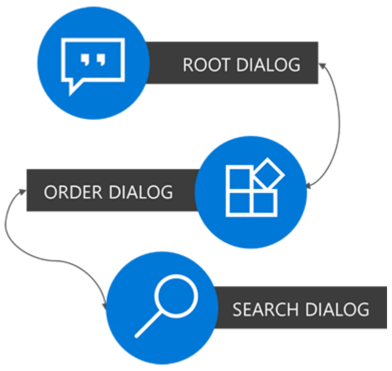
Dialogs may or may not have graphical interfaces. They may contain buttons, text, and other elements, or be entirely speech-based. Dialogs also contain actions to perform tasks such as invoking other dialogs or processing user input.

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**.



A traditional application flow

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**.



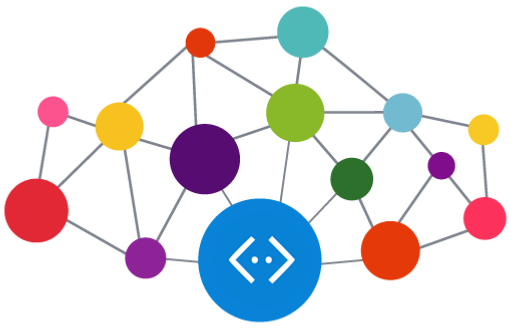
A bot experience flow

To help with the transition from traditional app development to bot development, and ensure developers are able to execute on the far-reaching vision of "Conversation as a Platform", Microsoft has created the **Microsoft Bot Framework**.

The Bot Connector

The Bot Connector

As the central entry point for bot communication and processes, **the Bot Connector handles communication, conversations, state, and authorization for all activities between a bot and its users**, including routing messages, managing state, registering bots, tracking user conversations, managing integrated services (such as translation) and even per-user and per-bot storage.



The Bot Connector

At the most basic level, the Bot Connector facilitates communication between a bot and a user by **relaying messages from bot to channel and from channel to bot**. Typically, a bot's logic is hosted as a web service, such as an Azure Web App, and, most importantly, has the ability to receive messages from users through the Bot Connector "service".

In reality, bot replies are actually sent to the Bot Connector using standard HTTPS POST methods, and are "wrapped" by methods exposed by the Bot Framework.

The Connector also has the task of normalizing messages sent to channels, which makes it easy to develop bots that are platform-agnostic. In Bot Framework-speak, **normalizing a message means converting it from the Bot Framework's schema into the channel's schema**. As an added bonus, in scenarios where a specific channel does not support all aspects of the framework's schema, the Bot Connector will attempt to convert the message to a format that the channel supports, providing built-in schema "failover". For example, if a bot sends a message via SMS that contains a "rich" card and action buttons, the Connector will intelligently render the card as an image, then include the actions as links in the message's text. Pretty cool, right?

Different Connectors, Different Platforms

The Microsoft Bot Framework Connector is typically just referred to as the Connector, however the specifics of working with the Connector vary based on development language.

In C#, for example, the Connector is accessed via the .NET `ConnectorClient`:

```
var connector = new ConnectorClient(new Uri(activity.ServiceUrl));
```

However, when using Node.js, you have some additional options, since the nature of Node.js is more generic. For example, in Node.js you might use the `ConsoleConnector` when working in development:

```
var connector = new builder.ConsoleConnector().listen();
```

When moving to production (or when testing in a local emulator) you might switch the Node.js `ChatConnector` scenario, and provide authorization values:

```
var connector = new builder.ChatConnector({
    appId: process.env.MicrosoftAppId,
    appPassword: process.env.MicrosoftAppPassword
});
```

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. The *ConsoleConnector* does not require an API endpoint.

By implementing this type of universal connection, the Microsoft Bot Framework **provides a single REST-based service that enables a bot to communicate across multiple channels** such as Skype, Teams, Facebook, and Slack, using a harmonized **connector service flow**.

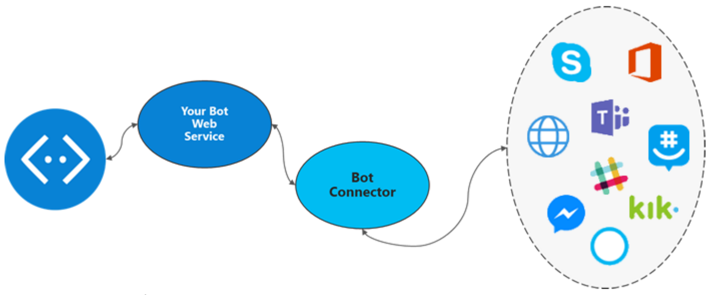
Quick Check (True or False)

1.0/1.0 point (graded)

Both the Bot Builder .NET SDK and Bot Builder Node.js SDK use the `ConnectorClient` object to facilitate communication between a bot and a user. (True or False)

Connector Service Flow

In the Microsoft Bot Framework, the Bot Connector encapsulates every aspect of bot to service communication, and performs virtually all of the “heavy lifting” required for communication between bots and third-party services. In the Microsoft Bot Framework, third-party services are referred to as "channels" and the Bot Connector simply provides the “glue” between the working parts.



Connector Service Flow

The Working Parts

Regardless of the simplicity or complexity of a bot implementation, the working parts remain the same:

- **Your bot logic**, written in C# or Node.js
- **Your bot Web Service**, such as an Azure Web App
- **The Bot Connector**, via the Microsoft Bot Framework
- **User interaction via a third-party service** (channel) such as Skype, Slack, or Facebook

Remember, the Connector manages communication, however **the Connector does not create the content** for communication. The creation of communication content, for example actual messages, is managed by creating and working with **activities**.

Learn About Verified Certificates

Activities

Activities

Activities

In the Microsoft Bot Framework, the Connector uses an **activity** object to pass information back and forth between a bot and a channel. The term activity is a bit generic, so, in this context, what's an activity? Simply stated, **an activity is any event that occurs between a bot and a user**, such as an a message being sent, or a notification that something about a conversation has changed.

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, such as a conversation update, a contact relation update, or deletion of user data.

The primary, supported activity types are:

- **Message**: Represents a communication between bot and user.
- **Conversation update**: Indicates that the bot was added to a conversation, other members were added to or removed from the conversation, or conversation metadata has changed.
- **Contact relation update**: Indicates that the bot was added or removed from a user's contact list.
- **Delete user data**: Indicates to a bot that a user has requested that the bot delete any user data it may have stored.
- **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.
- **End of conversation**: Indicates the end of a conversation.
- **Event**: Represents a communication sent to a bot that is not visible to the user.
- **Reaction**: Indicates that a user has reacted to an existing activity. For example, a user clicks the "Like" button on a message.

In general, most activities are simple notifications reflecting that an event has occurred, and contain little, if any, valuable information, however the message activity type is critical to bot development, as it provides the foundation for all communication content, from simple text strings, to rich, dynamic experience.

So, your bot will **send message activities** to communicate information to **and receive message activities** from users. This means all communicated bot content starts with a Microsoft Bot Framework **message**.

Quick Check (Multiple Choice)

1.0/1.0 point (graded)

Which of the following Microsoft Bot Framework activities represents an attempt to determine whether a bot's endpoint is accessible? (Choose one)

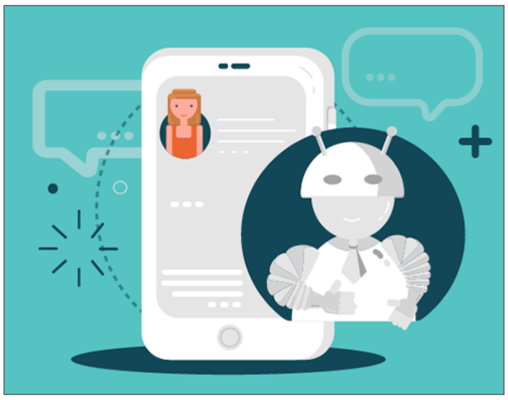
- Typing
- Event
- [✓ Ping] ✓
- Reaction

Submit You have used 1 of 2 attempts

Learn About Verified Certificates

Messages Overview

Messages Overview


Message Overview

In the Microsoft Bot Framework, bots send **message activities**, or simply **messages**, to communicate information to users, and likewise, also receive **messages** from users.

Understanding Message Activities

Start of transcript. Skip to the end.

Now, Microsoft Bot Framework, a message activity is used to

communicate information to and from a user.

And all though there are number of other activities,

like a conversation update or typing your ping activity.

Those activities are generally designed to communicate

with your code, to communicate with the back end,

things that a developer would write code to handle.

But a message activity is designed to communicate to and

Video
Download video file

Transcripts
Download SubRip (.srt) file
Download Text (.txt) file

Sending a bot message follows the same process, but happens a bit differently, depending on your development language. For example, sending a basic message using Node.js might look like this:

```
var customMessage = new builder.Message(session)
    .text("The Microsoft Bot Framework is awesome!")
    .textFormat("plain")
    .textLocale("en-us");
session.send(customMessage);
```

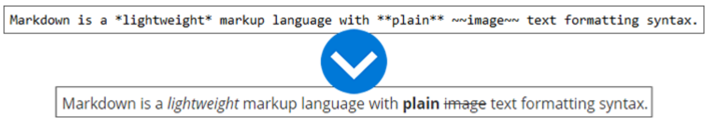
However, sending a message via C# would look more like this:

```
IMessageActivity message =  Activity.CreateMessageActivity();
message.Text = "The Microsoft Bot Framework is awesome!";
message.TextFormat = "plain";
message.Locale = "en-us";
await connector.Conversations.SendToConversationAsync((Activity)message);
```

Creating and Formatting Messages

Creating and Formatting Messages

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. Most developers are familiar with xml formatting, as its been a standard for over 20 years, however markdown has quickly become the formatting standard for just about everything. In fact, this course was originally created in markdown before being converted to HTML.



```
Markdown is a *lightweight* markup language with **plain** ~~image~~ text formatting syntax.
```

Markdown is a *lightweight* markup language with **plain** ~~image~~ text formatting syntax.

Markdown formatting

If you're not familiar with markdown, **markdown is a lightweight markup language with plain text formatting syntax**. Markdown is specifically designed to be easily converted HTML and many other formats and is often used to format readme files, writing messages in online discussion forums, and to create rich text using a plain text editor.

Every channel supports various levels of text formatting. Microsoft has gratefully provided the cool Channel Inspector for developers to evaluate whether a specific feature or formatting is supported on a channel.

Working with Markdown

7:27 / 7:29 1.0x

Markup.

For example, I could just say Preview Markdown in browser.

And you'll see how we have that content in a browser.

So you have to admit that pretty slick, pretty easy to work with.

And although you may not use all of these Markdown tags,

they certainly are helpful in Microsoft Bot Framework

conversations.

End of transcript. Skip to the start.

Video
Download video file

Transcripts
Download SubRip (.srt) file
Download Text (.txt) file

When creating conversation bot messages, markdown is ideal, as it provides simple syntax for highlighting and accentuating content with very little effort.

The following are some of the most commonly used text formatting in **markdown**.

- bold: **text** (**text**)
- italic: *text* (*text*)
- strikethrough: ~~text~~ (~~text~~)
- preformatted text: *codeblock* (`code block`)

To format content using XML, we could use the following examples for similar results:

- bold: <b>text</b> (**text**)
- italic: <i>text</i> (*text*)
- strikethrough: <s>text</s> (~~text~~)
- preformatted text: <code>code block</code> (`code block`)

These elements can be combined as needed, to enhance messages with emphasis, so **the markdown for a message might look like this**:

Which of the following **supported** operating systems would you like to choose to run your script: Windows, ~~Android~~, or *iOS*?

And then **get rendered like this**:

Which of the following **supported** operating systems would you like to choose to run your `script`: Windows, ~~Android~~, or *iOS*?

> In markdown, most formatting can be combined, and is cumulative. For example, you can combine **bold** and *italic* formatting to create a ***bold, italic*** format.

Creating messages using plain and formatted text is a breeze, however what if a user wants (or needs) to interact with your bot verbally? For this we need to take a look at text to be spoken, or **spoken text**.

Quick Check (Multiple Choice)

1.0/1.0 point (graded)

Which of the following lines of markdown would be used to format a line of text as both bold and italic? (Choose one) <<

- ○ **Formatted text**
- [✓ ***Formatted text ***] ✓
- ○ **~Formatted text ~**
- ○ *< b>Formatted text<\b>*

Submit You have used 1 of 2 attempts

Learn About Verified Certificates

Spoken Text

Spoken Text

When creating bots for a speech-enabled channels, such as Cortana, you can easily construct messages that specify the text to be spoken by your bot. This feature can be enhanced, as well, by providing an “input hint” which will attempt to influence the state of the client's microphone, or indicate whether your bot is accepting, expecting, or ignoring user input.

Spoken Text

In the Microsoft Bot Framework, there are multiple ways to specify the text to be spoken by your bot on a speech-enabled channel. For example, you can either use the **SayAsync** method, or specify the **Speak** property of the message to specify the text to be spoken by your bot.

In C#, creating a message activity and setting the Speak property might look like this:

```
Activity reply = activity.CreateReply();
reply.Speak = "This is the text that will be spoken.";
reply.InputHint = InputHints.AcceptingInput;
await connector.Conversations.ReplyToActivityAsync(reply);
```

In Node.js, it's almost the same code:

```
var msg = new builder.Message(session)
    .speak('This is the text that will be spoken.')
    .inputHint(builder.InputHint.acceptingInput);
session.send(msg).endDialog();
```

With a good bot conversation flow, input hints and spoken text can go a long way to providing a great user experience, however the reality is: your bot will always know more about what actions can be taken than a user will. Luckily, the Microsoft Bot Framework provides the concept of **suggested actions**.

Quick Check (Multiple Choice)

1.0/1.0 point (graded)

Which of the following properties would be used in an effort to influence the state of a client's microphone? (Choose one)

- ○ An influencer
- ○ An input accepter
- ○ A speaking hint
- ● An input hint ✓

Submit You have used 1 of 2 attempts

Learn About Verified Certificates

Suggested Actions

If you imagine a scenario where a bot is attempting to determine, for example, a paint color for an order, it would be pretty tough for a user to figure out all possible color choices. In these instances, **suggested actions enable your bot to present pre-defined choices** (based on what a bot already knows) as buttons, that the user can tap,

To clarify: suggested actions enhance the 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).

Providing suggested actions is, again, a simple process. In C#, your code might look like this:

```
var reply = activity.CreateReply("Thank you for expressing interest in our premium exterior paint colors! What
reply.Type = ActivityTypes.Message;
reply.TextFormat = TextFormatTypes.Plain;

reply.SuggestedActions = new SuggestedActions()
{
    Actions = new List<CardAction>()
    {
        new CardAction(){ Title = "Mauve", Type=ActionTypes.ImBack, Value="Mauve" },
        new CardAction(){ Title = "Cornflower Blue", Type=ActionTypes.ImBack, Value="CornflowerBlue" },
        new CardAction(){ Title = "Aztec Pink", Type=ActionTypes.ImBack, Value="AztecPink" }
    }
};
```

If you're working with Node.js, it's just as easy:

```
var msg = new builder.Message(session)
    .text("Thank you for expressing interest in our premium exterior paint colors! What color of paint would yo
    .suggestedActions(
        builder.SuggestedActions.create(
                session, [
                    builder.CardAction.imBack(session, "productId=1&color=mauve", "Mauve"),
                    builder.CardAction.imBack(session, "productId=1&color=cornflowerblue", "Cornflower Blue"),
                    builder.CardAction.imBack(session, "productId=1&color=aztecpink", "Aztec Pink")
                ]
            ));
session.send(msg);
```

In the Microsoft Bot Framework, *imBack* method will post the value to the chat window of the channel you are using. If you do not want the value to appear in the chat windows after selection, you can use the *postBack* method instead, which will post the selection back to your bot, but will not display the selection in the chat window.

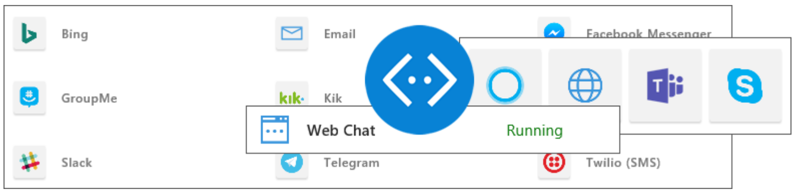
With a quick taste of what the Microsoft Bot Framework and Bot Builder SDK can provide, you might be ready to write some code of your own, however “hosting” a bot requires a little more knowledge. As with all software development, having a reliable test environment could make all the difference.

In bot development with the Microsoft Bot Framework, these tasks occur via the **Bot Framework Emulator**, which is provided as a separate component, and needs to be installed and configured.

Bot Channels

Bot Channels

In bot-speak, **a channel is a "conduit" into a third-party service, providing the connection between the Microsoft Bot Framework and user-facing communication apps**. Most channels support a core set of bot features, just as sending and receiving plain text messages. Since the Microsoft Bot Framework is designed to allow developers to create platform-agnostic bots, the framework targets all supported channels, and uses the Bot Connector to normalize and harmonize content and messages.



Bot Channels

The Microsoft Bot Framework currently supports the following 14 channels: **Bing**, **Cortana**, **Email**, **Facebook Messenger**, **GroupMe**, **Kik**, **Microsoft Teams**, **Skype**, **Skype for Business**, **Slack**, **SMS**, **Telegram**, **Web Chat**, and **WeChat**.

With such a broad range of third-party channel support, it may seem like all the bases are covered. However, what if your organization needs **a rich bot experience integrated into a proprietary system**, or you want to take advantage of the Microsoft Bot Framework to **surface a bot experience within existing apps and services**? Although not really a typical "channel", per se, the Microsoft Bot Framework supports programmatic access to backend bot services via the **Direct Line channel**.

Quick Check (Multiple Selection)

1.0/1.0 point (graded)

Which of the following channels are supported by the Microsoft Bot Framework? (Choose all that apply)

- [✓ Slack]
- Twitter
- [✓ Skype for Business]
- [✓ Microsoft Teams]

✓

Submit You have used 1 of 2 attempts

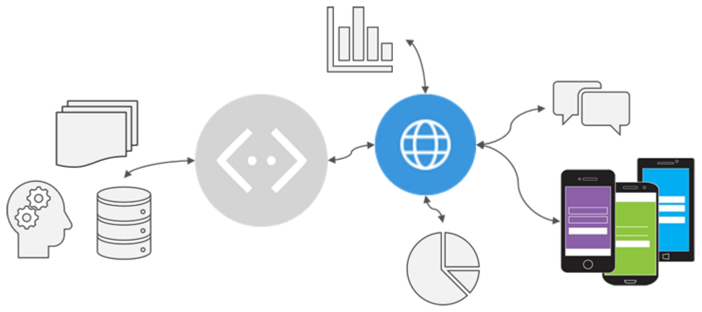
Learn About Verified Certificates

The Direct Line Channel

The Direct Line Channel

If you think about channels in practical terms, **a channel is really the “app” being used to converse with a bot**. As a rule, this is a simple scenario: you deploy your bot to a Skype channel so people can use your bot in Skype. You deploy your bot to a Microsoft Teams channel so people can use your bot in Microsoft Teams. You deploy your bot to a Slack channel so people can use your bot in...wait for it...Slack!

But, what if you want to take advantage of all the benefits of the Microsoft Bot Framework, including the Bot Builder SDK, in your own app? For example, it's quite common for organizations to have “roll your own” solutions spread out across their app landscape, and these solutions are often quite complex, pulling in data and resources from legacy, internal, and even external systems. This is where the Microsoft Bot Framework **Direction Line channel** fits in, exposing a robust REST-based API you can use to embed your bot into your own apps, webpages, or even server-to-server application—essentially making your own app a supported "channel".



The Direct Line Channel

Working with the Direct Line Channel

So go to Channels, and
we'll select Configure Direct Line Channel.
We'll call this channel Default.
And it's gonna give us some secret keys.

Now, key one and key two are gonna function the same,
they're really just for fail over.
We're gonna need these keys in order to authenticate
a direct line channel client.
So, flip over to Visual Studio and
take a look at how we might do that for
example using C sharp code.
So I'm over here in visual studio 2017 and
I've created a simple little direct line sample
client that is a Windows console app.
You notice that there's two things that we need to put in
here, direct line secret and our bot ID.
Now I know that our bot ID in this example is qnafactbot001.

Video
Download video file

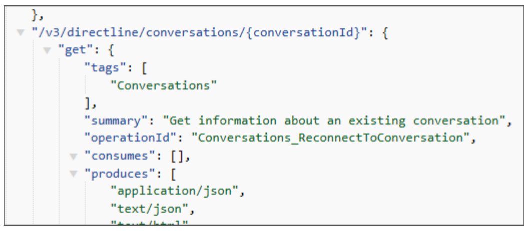
Transcripts
Download SubRip (.srt) file
Download Text (.txt) file

Client Libraries

Although the underlying services leverage platform-agnostic, REST-based services, as with everything else in the Microsoft Bot Framework, these services are wrapped in client libraries for ease of use in the two most common enterprise development languages: .NET and Node.js. And, like all libraries in the Bot Framework, **integrating Direct Line services into an app is simply a matter of installing the language-specific dependencies**. To use the Microsoft Bot Framework Direct Line libraries in a:

- **.NET client**: Install the `Microsoft.Bot.Connector.DirectLine` NuGet package
- **Node.js client**: Install the `botframework-directlinejs` NPM package

💡Want to use the Microsoft Bot Framework Direct Line API in Python, Java, Ruby, PHP, or any other development language? As an alternative to using the .NET 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.



```
},
"/v3/directline/conversations/{conversationId}": {
    "get": {
        "tags": [
            "Conversations"
        ],
        "summary": "Get information about an existing conversation",
        "operationId": "Conversations_ReconnectToConversation",
        "consumes": [],
        "produces": [
            "application/json",
            "text/json",
```

Direct Line API 3.0 Swagger file

Authentication

In a standard Microsoft Bot Framework channel, concepts and nuances of authentication are handled by the third-party client itself. Skype bot authentication is driven by a user's Skype account. Facebook Messenger authentication is driven by a user's Facebook account. When using the Direct Line channel, however, clients authentication is obtained with one of the following methods:

- A **secret**: A Direct Line secret is a master key that can be used to access **any conversation** that belong to the associated bot. **Direct line secrets do not expire.**

- A **token**. A Direct Line **token** is a key that can be used to access **a single conversation**. **Direct Line tokens expire, but can be refreshed.**

> A Direct Line **secret** is used to obtain a Direct Line **token**.

Deciding between these authentication options really depends on your business and technical requirements. For example, if you're creating a service-to-service application, using a **secret** is a reasonable and simple approach. However, 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**, where it will limit authorization to the current conversation —and then automatically expire, unless refreshed.

> 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.

The code you write to perform Direct Line authorization depends on your language of choice. For example, authenication via Node.js could look like this:

```
client.clientAuthorizations.add('AuthorizationBotConnector', new Swagger.ApiKeyAuthorization('Authorization', '
```

In C#, performing authentication is a bit more obfuscated, and would look more like this:

```
DirectLineClient client = new DirectLineClient(directLineSecret);
```

Either way, the authentication process is ultimately powered by the Direct Line REST-based API.

Conversations

In the Microsoft Bot Framework, Direct Line conversations are explicitly opened by clients (your apps and services) and can run as long as the bot and client continue participating with valid credentials. While the conversation is open, both the bot and client may send messages, and more than one client can connect to a given conversation—and even participate on behalf of multiple users.

Activities

Just like any other bot experience, clients and bots exchange several different types of activities, such as **message**, **ping**, and **typing** activities, and as a bonus, the Direct Line API even supports custom activities simply by specifying additional channel data.

Whether you're using Direct Line, or any of the more common third-party channels, bot feature support, especially when it comes to rich media, **varies greatly across channels**. For example, you can imagine rich media support for SMS messages may be quite a bit different from the support provided by Skype.

As you work through the bot development process, and especially as you make decisions regarding your targeted channels, your best friend will be the **Microsoft Bot Channel Inspector**.

Quick Check (True or False)

1.0/1.0 point (graded)

It is possible for a bot client to be notified that a user is typing a response, via the Direct Line API. (True or False)

[✓ True] ✓

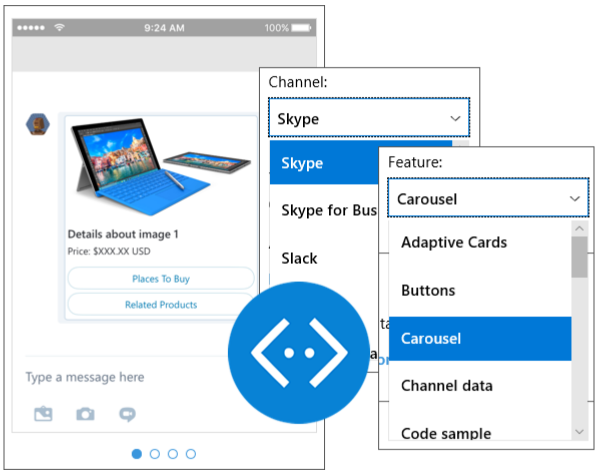
False

Submit   You have used 1 of 2 attempts

Learn About Verified Certificates

The Channel Inspector

The Microsoft Bot Framework makes it easy for developers to create bots with a variety of rich features such as formatted text, interactive buttons, images, video, animation—even rich cards displayed in carousel or list formats. This is great, however, a core concept of bot development with the Microsoft Bot Framework is that **developers write code in a platform-agnostic way**, yet **each channel ultimately controls how features are rendered** by its messaging clients.



The Channel Inspector

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 a feature a channel does not natively support, the channel may attempt to "down-render" message contents as text or as a static image. **This process could significantly impact the overall message's appearance and experience.** Or, in some cases, a channel may not support a particular feature at all. For example, GroupMe clients are not able to display the almost ubiquitous typing indicator.

The Channel Inspector is created to give you a preview of 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, especially when working with rich media by way of **attachments and cards**.

Media Attachments

Media Attachments

A message can certainly contain text strings, and of course various types of input actions, such as buttons, however, in the Microsoft Bot Framework **a message exchanged between a user and a bot can contain images, videos, audio content**, and even actual files, via the *Attachments* property.

Understanding Attachments

Start of transcript. Skip to the end.

Any type of rich content,

or really content beyond just simple plain or formatted text

is considered an attachment in the Microsoft Bot Framework.

But this term attachment can be a little bit misleading.

In the context of the Bot Framework attachment content

is often embedded into a message as the primary portion of

the displayed content.

Video
Download video file

Transcripts
Download SubRip (.srt) file
Download Text (.txt) file

The *Attachments* property of a message contains an array of *Attachment* objects that **represent the media attachments and rich cards to display within a message**, and writing this code would look very familiar to anyone who's worked with email or other messaging systems.

There are three (3) types of attachments that can be sent (although attachment support varies by channel) but the supported attachment types are **media and files**, **cards**, and **hero cards**.

- **Media and Files**: Send files such as images, audio and video
- **Cards**: Send a rich set of visual cards via JSON payload
- **Hero Cards**, See send a rich card containing a single large image, one or more buttons, and text

In C#, adding an image as an attachment to a message requires an accessible location (URL), content type, and a display name:

```
replyMessage.Attachments.Add(new Attachment()
{
    ContentUrl = "https://www.traininglabs.io/bot-images/botimage.png",
    ContentType = "image/png",
    Name = "botimage.png"
});
```

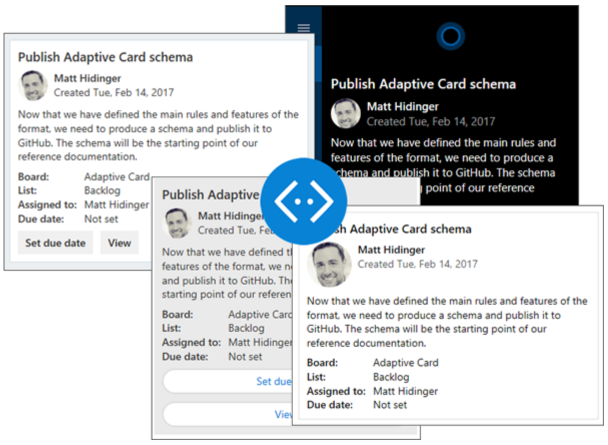
And, of course, in Node.js, almost the same code:

```
var attachment = msg.attachments[0];
session.send({
    attachments: [
    {
        contentUrl: "https://www.traininglabs.io/bot-images/botimage.png",
        contentType: "image/png",
        name: "botimage.png"
    }]
});
```

To extend the concept of attachments, the Microsoft Bot Framework goes beyond simple image and file based content, and uses the powerful and flexible concept of **cards**.

Cards Overview

Cards Overview


Publish Adaptive Card schema
Matt Hidinger
Created Tue, Feb 14, 2017
Now that we have defined the main rules and features of the format, we need to produce a schema and publish it to GitHub. The schema will be the starting point of our reference documentation.
Board: Adaptive Card
List: Backlog
Assigned to: Matt Hidinger
Due date: Not set

Cards Overview

In the Microsoft Bot Framework, a card comprises a title, description, link, and images. Some channels, such as Skype and Facebook, support sending these rich graphical cards to users, which can include interactive buttons. In the Microsoft Bot Framework these cards are, appropriately enough, referred to as **rich cards**. A message can contain multiple rich cards, displayed in either *list* format or *carousel* format.

The Bot Framework currently supports eight (8) types of rich cards:

- **Adaptive**: A customizable card that can contain any combination of text, speech, images, buttons, and input fields. See per-channel support.
- **Animation**: A card that can play animated GIFs or short videos.
- **Audio**: A card that can play an audio file.
- **Hero**: A card that typically contains a single large image, one or more buttons, and text.
- **Thumbnail**: A card that typically contains a single thumbnail image, one or more buttons, and text.
- **Receipt**: 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.
- **Sign in**: 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**: A card that can play videos.

💡To determine the type of rich cards that a channel supports and see how the channel renders rich cards, you can use the Channel Inspector covered in the previous lesson, or consult the channel's documentation for information about limitations on the contents of cards.

As you may recall, the Bot Connector handles communication between a bot and a third-party service (channel) and is also responsible for managing and normalizing messages. This means **the Bot Connector will render these cards using schema native to the channel**, providing 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.

Creation of a rich card is as simple as spinning up a card type, and adding it to the Attachments array, just like any other message attachment.

Creating a **Thumbnail Card** in C#:

```
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)
```

Or a **Receipt Card** might look like this:

```
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);
```

And finally, a **Sign In Card** is even easier:

```
SigninCard plCard = new SigninCard(title: "You need to authorize me", button: plButton);

Attachment plAttachment = plCard.ToAttachment();
replyToConversation.Attachments.Add(plAttachment);
```

Up until now, you've been working with activities and messages as a single entity, and have a good feel for creating and formatting messages. You've also got a pretty good handle on using standard and rich card attachments to create more engaging messages. However, in the real world, **bot experiences are part of a full conversation**, with potentially complex relationships between messages, replies, and actions.

To work with full conversations, especially "guided" conversations, the Microsoft Bot Framework uses the notion of **dialogs and forms**.

Quick Check (Multiple Choice)

1.0/1.0 point (graded)

Which of the following card types is designed to display animated GIFs or short videos? (Choose one)

- Audio
- Multimedia
- Video
- [✓ Animation] ✓

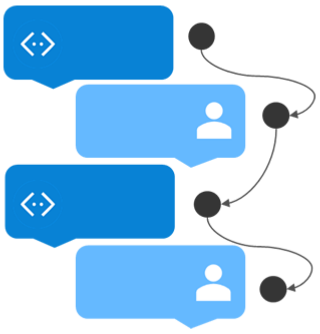
Submit You have used 1 of 2 attempts

Learn About Verified Certificates

Dialog Concepts

Dialog Concepts

In the Microsoft Bot Framework, dialogs enable developers to “model” conversations and manage conversation flow. Although bots send and receive messages, a bot actually communicates with a user via a full conversation. Based on this idea, **every conversation is organized into a series of dialogs**.



Dialog Concepts

Using the Bot Builder SDK, bot dialog steps can follow a “waterfall model”, as well as integrate a series of prompts, wherein a 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.

The concept of a “waterfall model” generally means a relatively linear sequential design or flow.

In more familiar terms, dialogs are a way of wrapping an entire “experience” into an easily managed interaction based on a “chained” and “conversational” paradigm.

- Send information to a user
- Prompt a user for more information or confirmation
- Provide conditional logic
- Provide “as you need it” content
- Can contain or forward to other dialogs.

Spinning up a dialog is as easy as working with messages and attachments. Here's an example of creating a dialog using Node.js:

```
bot.dialog('greetings', [
    function (session) {
        builder.Prompts.text(session, 'Hi! What is your name?');
    },
    function (session, results) {
        session.endDialog(`Hello ${results.response}!`);
    }
]);
```

Or a more complex “confirmation” example using C#:

```
PromptDialog.Confirm(
                context,
                AfterResetAsync,
                "Are you sure you want to reset the count?",
                "Didn't get that!",
                promptStyle: PromptStyle.None);
```

In fact, using the Bot Builder .NET SDK, extremely complex scenarios, including dialog chaining, can be accomplished with just a few lines of code:

```
new Regex("^chicken"),
            (context, text) =>
                Chain
                .Return("why did the chicken cross the road?")
                .PostToUser()
                .WaitToBot()
                .Select(ignoreUser => "to get to the other side")
```

As you can tell, dialogs are powerful and flexible, but handling a guided conversation, such as ordering a sandwich, or making an airline reservation, could require quite a bit effort, as there are potentially a large number of possibilities for what could happen next. To work with guided conversations, the Bot Builder SDK introduces the concept of **forms**.

Form Concepts

Form Concepts

Although Microsoft Bot Framework dialogs are the basic building block of a bot conversation, it's difficult to create a "guided" conversation, with dialogs. By leveraging the concept of **forms**, users can be guided through the relevant steps of a full conversation—similar to the process of filling out a traditional form—and your bot logic can easily provide help and guidance along the way.

Form Concepts

Understanding Forms in a Bot Experience

Start of transcript. Skip to the end.

0:00 / 1:41 1.0x

Pretty much if you live in any country or area of the world,

you have had to fill out forms and

understand the concept of forms.

In the Microsoft Bot Framework the concept of a form is very

similar to the concept of a traditional paper or

web-based form.

Where it contains fields, prompts, and actions.

In a Microsoft Bot Framework, a form can contain text inputs,

Video
Download video file

Transcripts
Download SubRip (.srt) file
Download Text (.txt) file

Creating bot conversation forms is driven by your choice of development language. For .NET developers, the Bot Builder SDK enables forms via strongly typed and attributed C# classes. For Node.js scenarios, form definition occurs via a well-formatted JSON schema.

For example, in C#, a form option could look like this:

```
public enum SauceOptions
{
    ChipotleSouthwest, HoneyMustard, LightMayonnaise, RegularMayonnaise,
    Mustard, Oil, Pepper, Ranch, SweetOnion, Vinegar
};
```

And for Node.js, this same form option could be similar to the following:

```
"Sauces": {
    "type": [
    "array",
    "null"
    ],
        "items": {
        "type": "string",
        "enum": [
            "ChipotleSouthwest",
            "HoneyMustard",
            "LightMayonnaise",
            "RegularMayonnaise",
            "Mustard",
            "Oil",
            "Pepper",
            "Ranch",
            "SweetOnion",
            "Vinegar"
            ]
    }
}
```

Although this process is pretty straightforward, full implementation of a form-based guided experience would require a combination of prompts, choices, and options. To combine these types of form elements into a single, cohesive flow, the Bot Builder SDK uses the notion of **FormFlow**.

Using FormFlow

Using FormFlow

Enhancing the power of forms with Microsoft Bot Builder .NET SDK FormFlow can greatly simplify the process of managing guided bot conversations. As a core feature, FormFlow **automatically generates the dialogs that are necessary to manage a guided conversation**, based on guidelines and requirements you specify via code. 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**.

As a bonus, your bot conversation can use a combination of FormFlow-generated dialogs and other types of dialogs, such as LUIS-injected dialogs, to help evaluate user input and determine intent, based on natural language understanding.

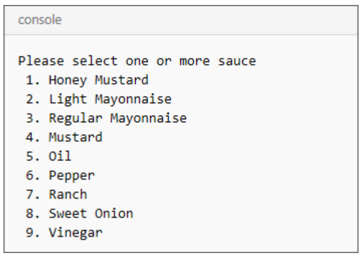
💡Microsoft Bot Builder FormFlow features are only available "out of the box" when using the .NET version of the Bot Builder SDK, however there are FormFlow implementations available for Node.js via third parties, most of which are open source.

Field-based forms

When creating a guided experience using FormFlow, developers need to define a form by creating a C# class containing one or more public properties representing the data that the bot will collect from the user, and then the FormFlow intelligence will know exactly what to do with it. For example, in C#, using the previous scenario defining `SauceOptions`:

```
public enum SauceOptions
{
    ChipotleSouthwest, HoneyMustard, LightMayonnaise, RegularMayonnaise,
    Mustard, Oil, Pepper, Ranch, SweetOnion, Vinegar
};
```

When FormFlow sees this enumeration as part of the form, it will **automatically render formatted versions of the options for a user**. Notice the automatically generated phrase *"Please select one or more sauce"*, and well as the properly formatted display names. FormFlow was smart enough to convert *"HoneyMustard"* to *"Honey Mustard"*, based on enumeration casing.



console

```
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
```

Form Concepts

You have to admit that's pretty remarkable, and it only gets better. FormFlow still delivers a fairly generic experience, and strictly follows the logic of your classes. To further customize a guided experience, the Bot Builder .NET SDK uses **FormBuilder**.

Quick Check (True or False)

1.0/1.0 point (graded)

When FormFlow sees an enumeration as part of the form, it will automatically render formatted versions of the options for a user. (True or False)

[✓ True] ✓

False

Submit   You have used 1 of 2 attempts

Using FormBuilder

Using FormBuilder

By using Bot Builder .NET SDK FormFlow, developers can customize a bot experience by using strongly-typed business rules and business logic. Additionally, by combining FormFlow with **FormBuilder**, developers can customize a guided bot experience even further—**by specifying the sequence in which the form executes steps and dynamically defining field values, confirmations, and messages**.

Creating a custom experience using FormBuilder is as simple as ordering the execution of elements in code. For example, ordering the steps for a klobásníky order process might look like this, in C#:

```
    return new FormBuilder<SandwichOrder>()
        .Message("Welcome to the klobásníky order bot!")
        .Field(nameof(Klobasniky))
        .Field(nameof(Length))
        .Field(nameof(Bread))
        .Field(nameof(Cheese))
        .Field(nameof(Fillings),
```

In this example, the bot is performing the following steps, in this order:

- Displays a welcome message.
- Prompts for a klobásníky type selection
- Prompts for a klobásníky length selection
- Prompts for a klobásníky bread type selection
- Prompts for a cheese selection
- Prompts for additional fillings

> Not familiar with the delicious klobásníky? A klobásníky is a savory finger food of Czech origin, and is often misunderstood to be a variation of the kolache. The truth is, kolache are only filled with non-meat fillings. Unlike kolache, which came to the United States with Czech immigrants, klobasniky were first made by Czechs that settled in Texas, use a sweeter dough, and are always filled with some type of meat.

This process cleanly builds a strongly-typed definition for a bot form, with all dialogs created as desired, and can even include step-specific validation. For example, this block of code would validate filling selections, add multiple fillings, and alternatively add all fillings:

```
.Field(nameof(Fillings),
            validate: async (state, value) =>
            {
            var values = ((List<object>)value).OfType<FillingOptions>();
            var result = new ValidateResult { IsValid = true, Value = values };
            if (values != null && values.Contains(FillingOptions.Everything))
            {
            result.Value = (from FillingOptions filling in Enum.GetValues(typeof(FillingOptions))
            where filling != FillingOptions.Everything && !values.Contains(filling)
            select filling).ToList();
            }
            return result;
            })
```

Finally, submission of the bot form to a user:

```
.Confirm("Do you want to order your {Length} {Klobasniky} using {Bread} {&Bread} with {[{Cheese} {Fillings} {Sa
        .AddRemainingFields()
        .Message("Thanks for ordering a klobásníky!")
        .OnCompletion(processOrder)
        .Build();
```

This may seem like quite a bit of code, however imagine juggling all these processes and options via “roll your own” conditional code!

As another bonus, Microsoft Bot Framework **Bot Builder forms and dialogs manage their own state**, however various mechanisms also exist for managing storage and state for other values, such as third-party tokens, user names, and connection information to services. Selection of storage and state methods really depends on basic **bot state concepts**.

Quick Check (True or False)

1.0/1.0 point (graded)

By using FormBuilder, a developer can specify the sequence in which the form executes steps and dynamically defining field values, confirmations, and messages. (True or False)

[✓ True] ✓

False

Submit You have used 1 of 2 attempts

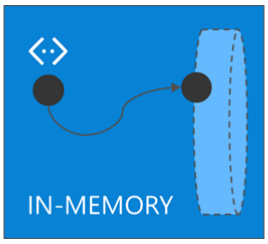
Learn About Verified Certificates

Bot State Options

Bot State Options

The Bot Builder Framework enables your bot to store and retrieve state data that is associated with all users, a single conversation, or even 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 testing and prototyping purposes, you can use the Bot Builder in-memory data storage. For production bots, you can implement your own storage adapter or use one of Azure Extensions.



In-memory data storage

The Microsoft Bot Framework Azure Extensions support the following storage locations:

- Azure Table Storage
- Azure Cosmos DB
- Azure SQL

💡Although used quite frequently in both the documentation and SDK examples, the Bot Framework State Service API is not recommended for production environments, and has been deprecated. It is recommended that you update your bot code to use the in-memory storage adapter for testing purposes or use one of the **Azure Extensions** for production bots.

Integrating in-memory data storage occurs by spinning up a new `MemoryBotStorage` object, which might look like this in Node.js:

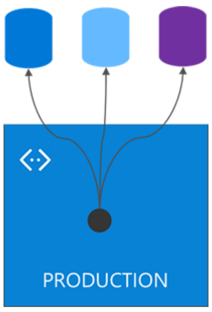
```
var store = new builder.MemoryBotStorage();
```

The process would be similar in C#, only using the .NET `InMemoryDataStore` object instead:

```
var store = new InMemoryDataStore();
```

Production Storage

Since in-memory data storage is not recommended for production scenarios, it's recommended (and convenient) to use an Azure-based storage mechanism via the Bot Builder Azure Extensions. Transitioning development storage to production storage is really just a matter of using the target storage.



Production Storage

For example, using **Azure Table Storage** would only change a single line of C# code:

```
var store = new TableBotDataStore(['YOUR TABLE STORAGE CONNECTION STRING']);
```

Using Azure Cosmos DB is a little more work, however it's really quite simple:

```
var uri = new Uri(ConfigurationManager.AppSettings['YOUR DOCUMENTDB URI']);
var key = ConfigurationManager.AppSettings['YOUR DOCUMENTDB KEY'];
var store = new DocumentDbBotDataStore(uri, key)
```

With a bot storage method selected, your code can easily **manage state data**.

Quick Check (Multiple Selection)

1.0/1.0 point (graded)

Which of the following storage locations would be a good fit for a bot in production? (Choose all that apply)

- ☐ In-memory data storage
- ☑ Azure Table Storage
- ☑ Azure Cosmos DB
- ☑ Azure SQL

✓

Submit You have used 1 of 2 attempts

Learn About Verified Certificates

Managing State Data

Managing State Data

Unlike application-based storage mechanisms, where datatypes and values are "wide open", bot storage is designed to work against specific types of bot information. The Bot Builder SDK allows storage for the following storage containers:

Working with State

Start of transcript. Skip to the end.

When we talk about managing state in a bot with
Microsoft Bot Framework,
we're really talking about three different things.
First, we're talking about a storage mechanism
such as in memory storage or Azure table storage,
we're talking about storage containers, and then we're
talking about setting values on the storage containers.
In a Microsoft Bot Framework, we're really working with four

Video
Download video file

Transcripts
Download SubRip (.srt) file
Download Text (.txt) file

- **User data**: Contains data that is **saved for the user on the specified channel**.

  This data will persist across multiple conversations.

- **Private conversation data**: 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.

- **Conversation data**: 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.

- **Dialog data**: Contains data that is **saved for the current dialog only**.

  Each dialog maintains its own copy of this property.

Using these storage containers makes it easier to persist data based on bot context and scope, for example, storing a current user's information might look like this, in Node.js:

```
session.userData.userName = "Geddy Lee";
session.userData.userAge = 57;
session.userData.hasChildren = true;
```

Retrieving the data is obviously just a standard reversal of values:

```
var userName = session.userData.userName;
var userAge = session.userData.userAge;
var hasChildren = session.userData.hasChildren;
```

Clearing or resetting data, in Node.js, makes sense, too:

```
session.userData = {};
session.privateConversationData = {};
session.conversationData = {};
session.dialogData = {};
```

💡Notice the use of an empty array ({}) as the clear “setter”. **Never** set the container value to `null` (or remove the object from the session) as doing so will create errors the next time you try to access the specific container.

With a deep understanding (and some great examples) of using the Microsoft Bot Framework to create conversation bots, it's time to test your knowledge and write some code.

Learn About Verified Certificates