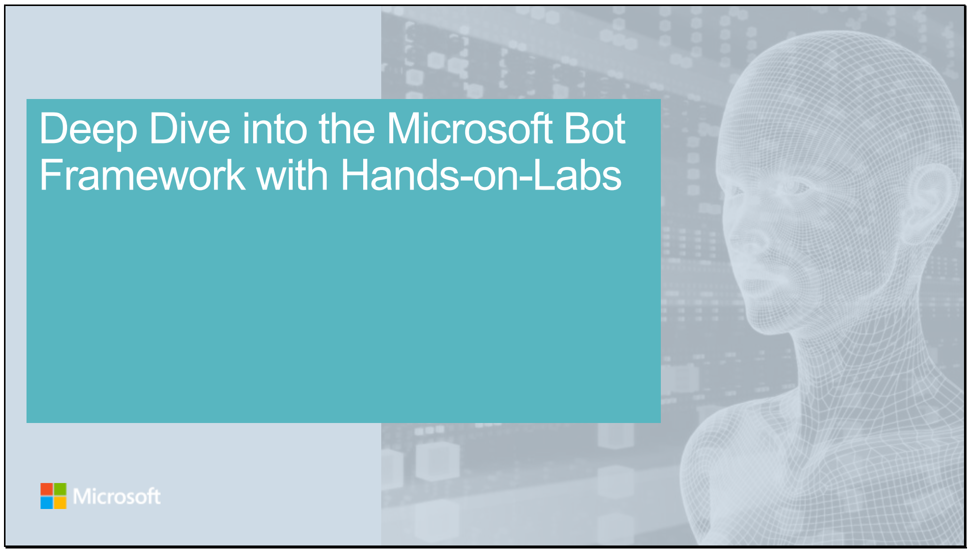
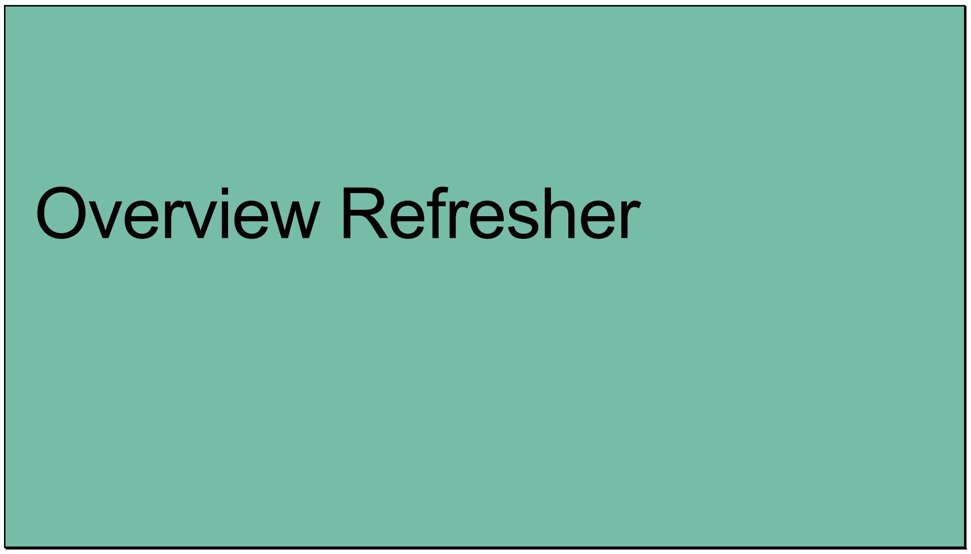


Deep Dive into the Microsoft Bot Framework with Hands-on-Labs

Microsoft

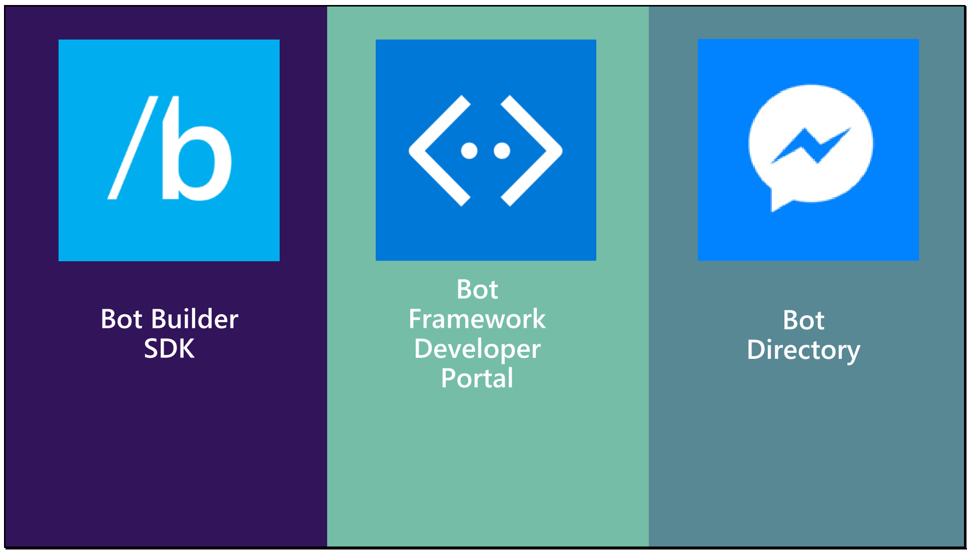
If the links in this deck are broken please let us know (mailto: michhar@Microsoft.com). Thanks in advance and enjoy learning about bots and the Microsoft Bot Framework.

Slide 2



Overview Refresher

Slide 3



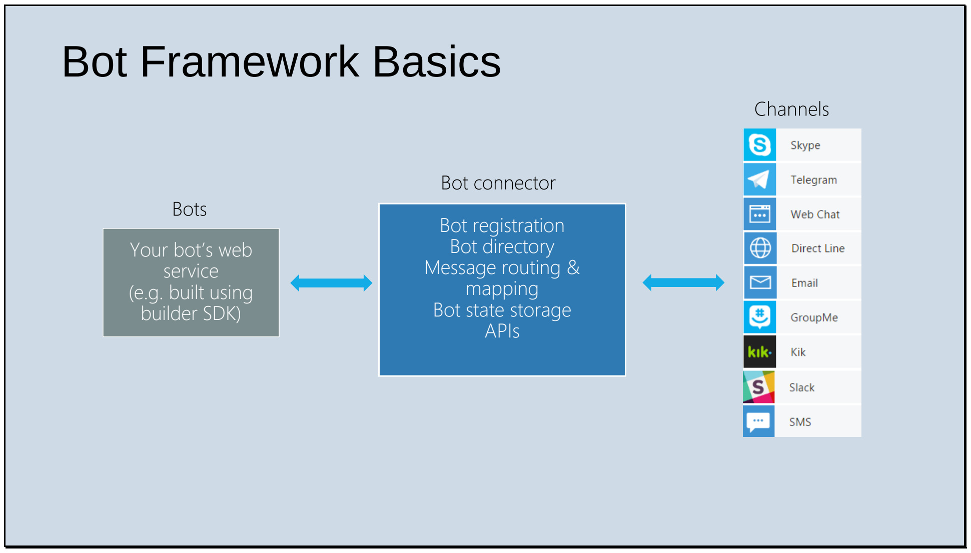
The Bot Builder SDK is an open source SDK hosted on GitHub that provides everything you need to build great dialogs within your Node.js-, .NET- or REST API-based bot. *

The Bot Framework Developer Portal lets you connect your bot(s) seamlessly text/sms to Skype, Slack, Facebook Messenger, Kik, Office 365 mail and other popular services. Register, configure and publish.

The Bot Directory is a public directory of all reviewed bots registered through the Developer Portal.

* Bot builder and bot connector SDK now one in V3 of framework: http://docs.botframework.com/en-us/support/upgrade-to-v3/#botbuilder-and-connector-are-now-one-sdk

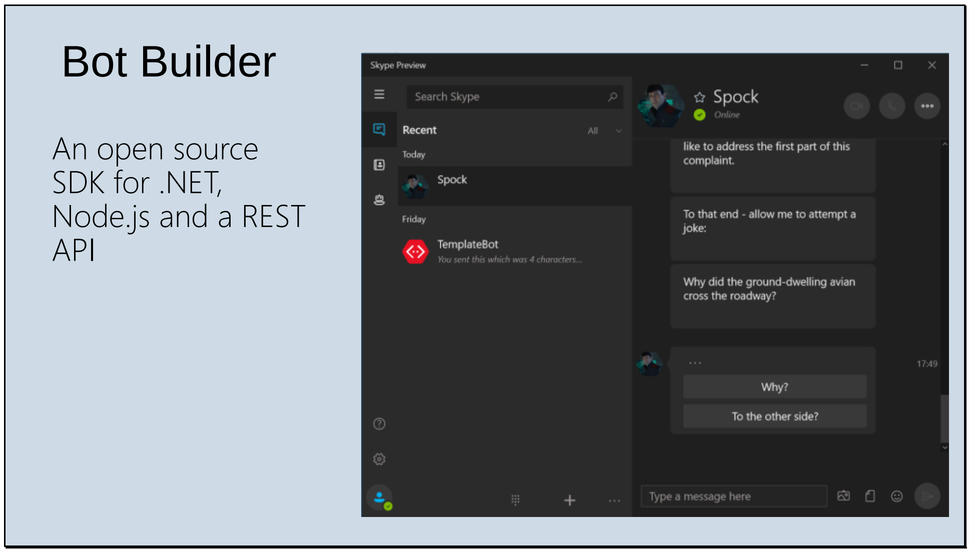
Slide 4



Bot Framework Basics

The Connector library is the exposition of the REST API. The Builder library adds the conversational dialog programming model and other features (e.g. prompts, waterfalls, chains, guided form filling) and access to cognitive services (e.g. LUIS). (see https://docs.botframework.com/en-us/technical-faq)

Slide 5



Bot Builder

An open source SDK for .NET, Node.js and a REST API

Skype Preview

Search Skype

Recent

Today

Spock

Friday

TemplateBot

You sent this which was 4 characters...

Spock

Online

like to address the first part of this complaint.

To that end - allow me to attempt a joke:

Why did the ground-dwelling avian cross the roadway?

17:49

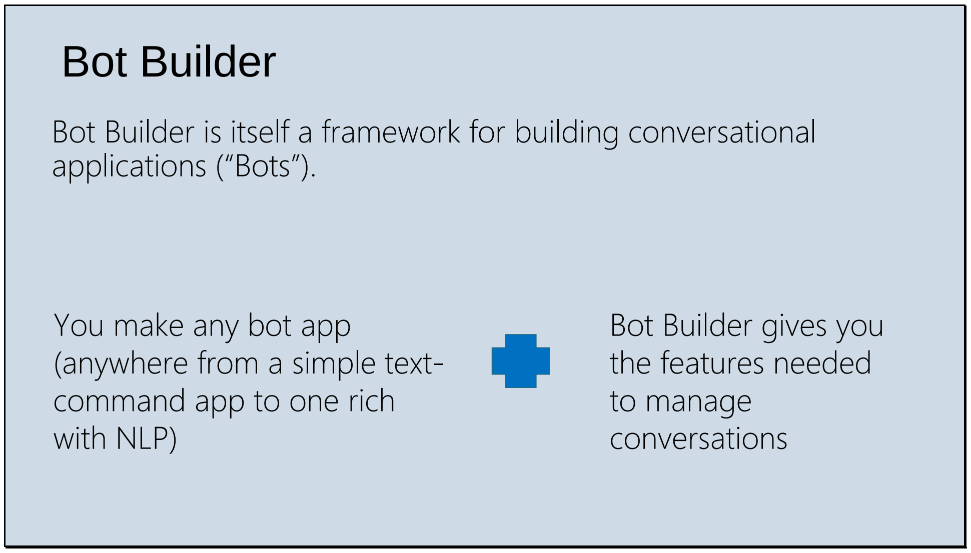
Why?

To the other side?

Type a message here

showing a conversation with a publicly registered bot called Spock. It has dialogs and attachments with buttons as a menu as we'll see later.

Slide 6



Bot Builder

Bot Builder is itself a framework for building conversational applications (“Bots”).

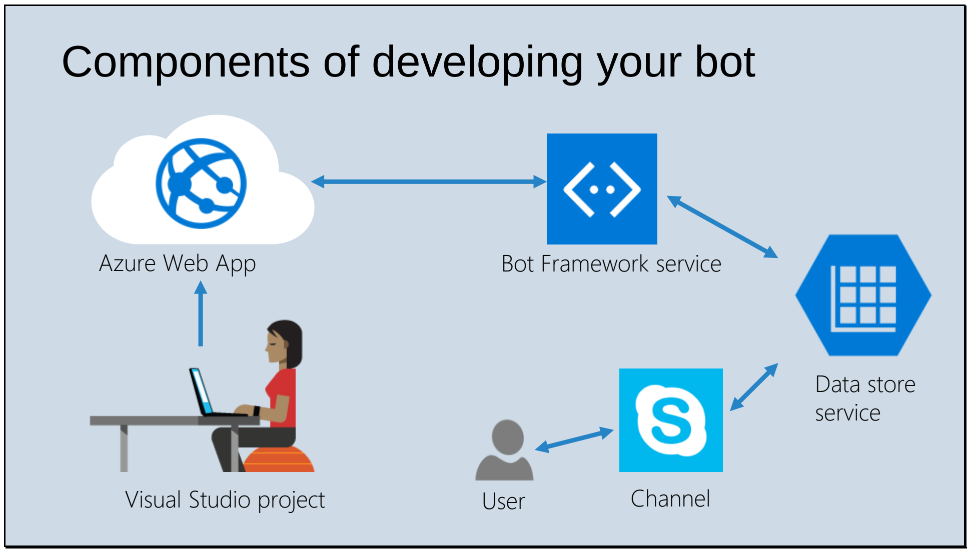
You make any bot app (anywhere from a simple text-command app to one rich with NLP)

➕

Bot Builder gives you the features needed to manage conversations

a bot is just an application that addresses the huge effort involved in repetitive tasks (a bot can generalize a problem – provide a fun interface to a game or experience for instance, or collect votes over a choice of opinions – as a bot it’s a deployable and repeatable solutions, plus hopefully an enjoyable user experience)

Slide 7



Components of developing your bot

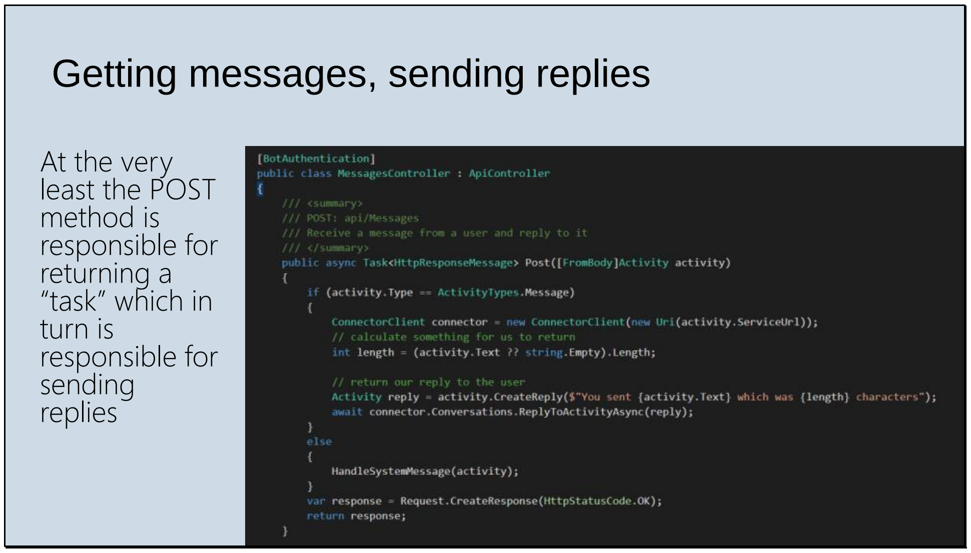
Steps:
The developer writes their bot code in Visual Studio, leveraging the BF application template and using the Bot Builder SDKs plus any extra librarie, SDKs or APIs available
Publish as a Web App to create an endpoint for the bot to talk to the Azure service in the cloud
Connect to the Bot Framework
Pass user and conversation data between channel and Framework (data store in the state service, managed by the BF)
The user interacts with the bot on a channel of their choosing

Slide 8

Basics

You’ve seen some code so far, so let’s dive in to the .NET Bot Builder again.

Slide 9



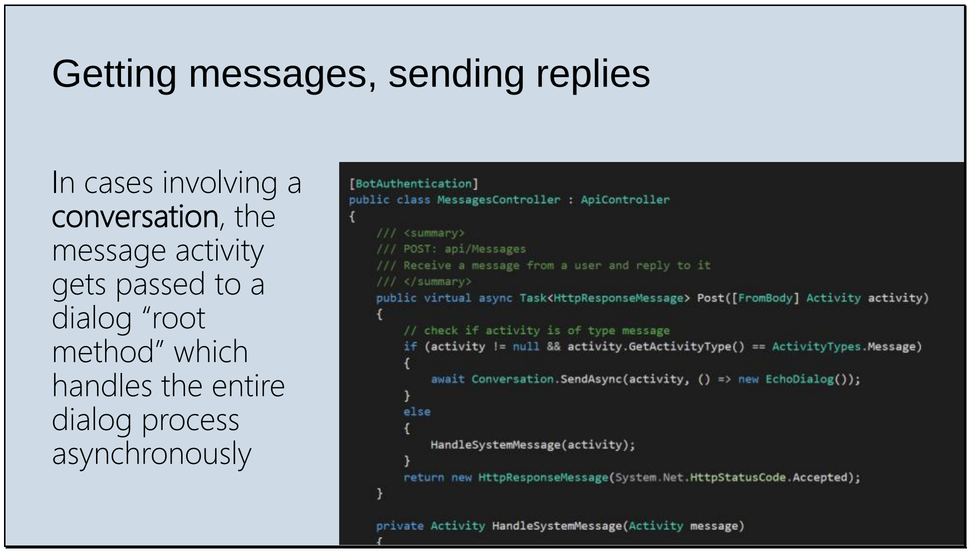
Getting messages, sending replies

At the very least the POST method is responsible for returning a "task" which in turn is responsible for sending replies

```
[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)
        {
            ConnectorClient connector = new ConnectorClient(new Uri(activity.ServiceUrl));
            // calculate something for us to return
            int length = (activity.Text ?? string.Empty).Length;

            // return our reply to the user
            Activity reply = activity.CreateReply($"You sent {activity.Text} which was {length} characters");
            await connector.Conversations.ReplyToActivityAsync(reply);
        }
        else
        {
            HandleSystemMessage(activity);
        }
        var response = Request.CreateResponse(HttpStatusCode.OK);
        return response;
    }
```

e.g. activities – message, typing, delete user data
Note the explicit creation of the connector client in the Bot Application Template C# code

Slide 10



Getting messages, sending replies

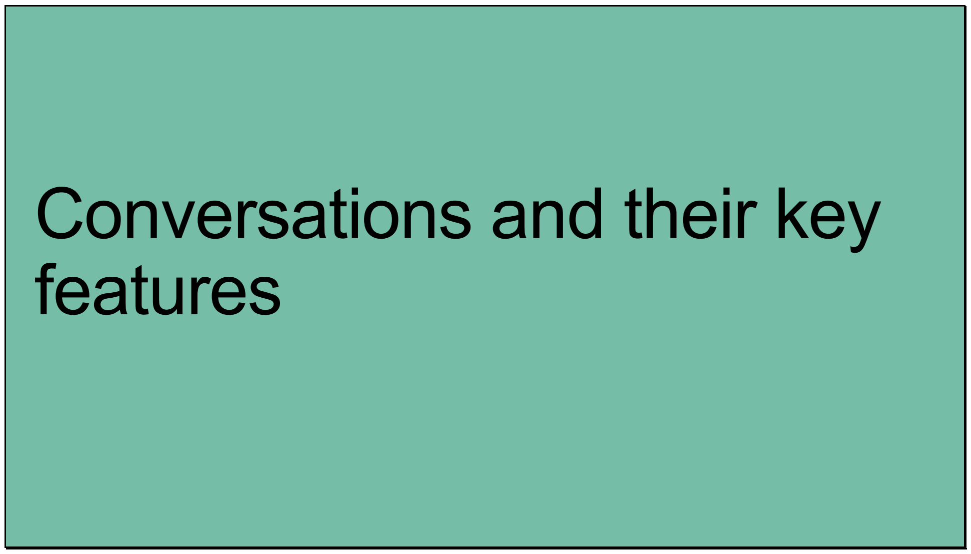
In cases involving a **conversation**, the message activity gets passed to a dialog “root method” which handles the entire dialog process asynchronously

```
[BotAuthentication]
public class MessagesController : ApiController
{
    /// <summary>
    /// POST: api/Messages
    /// Receive a message from a user and reply to it
    /// </summary>
    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);
    }

    private Activity HandleSystemMessage(Activity message)
    {
```

The dialog “root method” is called SendAsync in C# and it controls the following process:

- Instantiate the required components.
- Deserialize the dialog state (the dialog stack and each dialog's state) from IBotDataStore (the default implementation uses the Bot Connector state API as backing IBotDataStore).
- Resume the conversation processes where the Bot decided to suspend and wait for a message.
- Send the replies.
- Serialize the updated dialog state and persist it back to IBotDataStore.

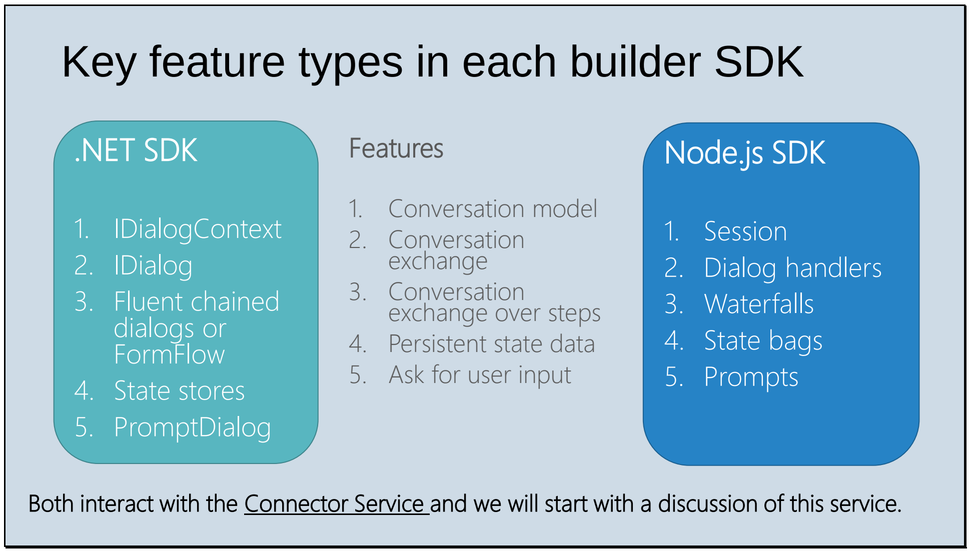
The dialog process can involve sub-dialogs
In between 2 and 3 a dialog class object gets made and the message processed, possibly sub-dialogs spawned and further messages obtained from the user, message waiting occurs, dialog state updated, and more.



Conversations and their key features

Conversation – dialog or stack of dialogs handled by a model

Slide 12



Key feature types in each builder SDK

.NET SDK

1. IDialogContext
2. IDialog
3. Fluent chained dialogs or FormFlow
4. State stores
5. PromptDialog

Features

1. Conversation model
2. Conversation exchange
3. Conversation exchange over steps
4. Persistent state data
5. Ask for user input

Node.js SDK

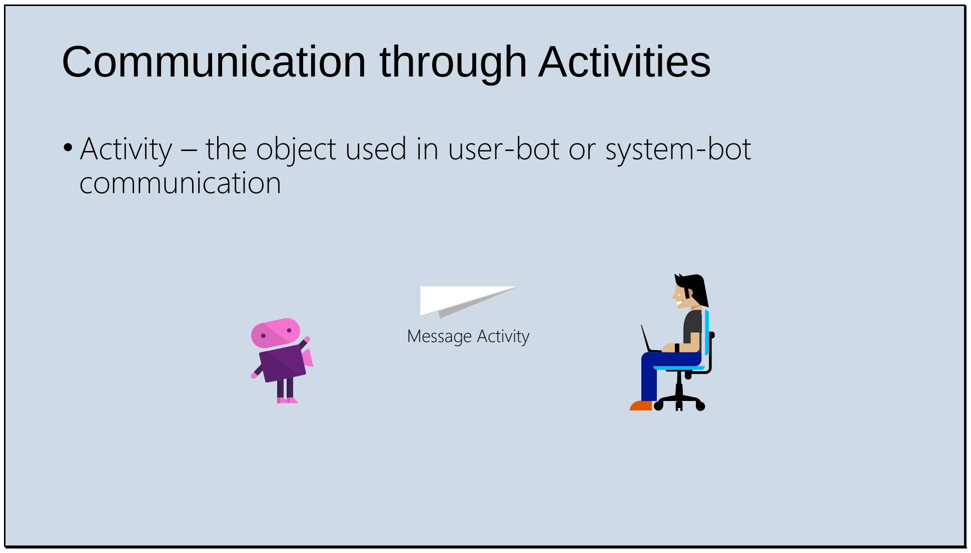
1. Session
2. Dialog handlers
3. Waterfalls
4. State bags
5. Prompts

Both interact with the Connector Service and we will start with a discussion of this service.

They may use different names and different implementations, but have essentially all of the same key features.

The Connector Service

Slide 14



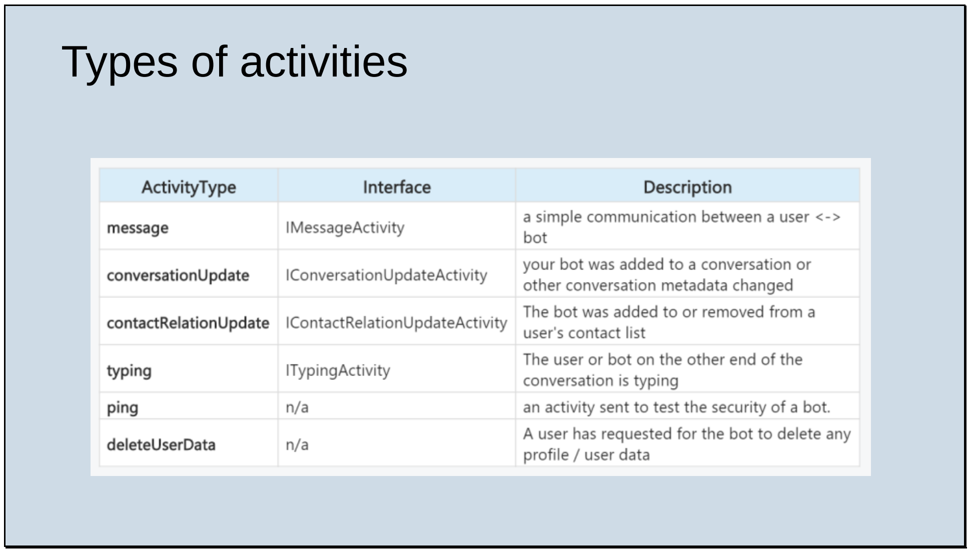
Communication through Activities

- Activity – the object used in user-bot or system-bot communication

Message Activity

for example, a Message activity could be as simple as the text exchange (the “object”) or as complex as a multi-card carousel with buttons and actions (also, the “object”) – see how the players are the same, but the objects are different. activities both encapsulate and abstract interactions between a user or the system and a bot. The activity type dictates how the communication will appear and what properties and controls it will have.

Slide 15

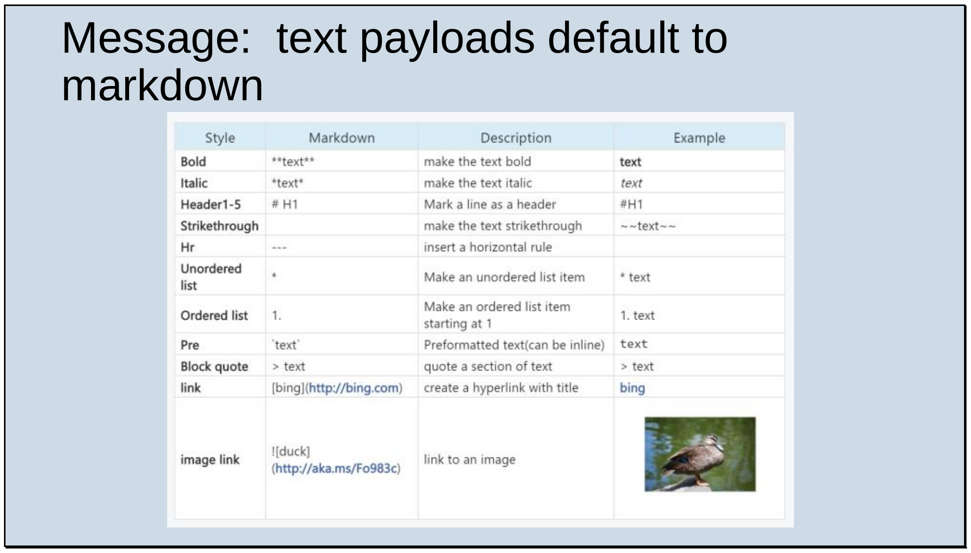


Types of activities

ActivityType	Interface	Description
message	IMessageActivity	a simple communication between a user <-> bot
conversationUpdate	IConversationUpdateActivity	your bot was added to a conversation or other conversation metadata changed
contactRelationUpdate	IContactRelationUpdateActivity	The bot was added to or removed from a user's contact list
typing	ITypingActivity	The user or bot on the other end of the conversation is typing
ping	n/a	an activity sent to test the security of a bot.
deleteUserData	n/a	A user has requested for the bot to delete any profile / user data

message: text "payload" is in markdown syntax by default and is rendered as appropriate for a certain channel.

Slide 16

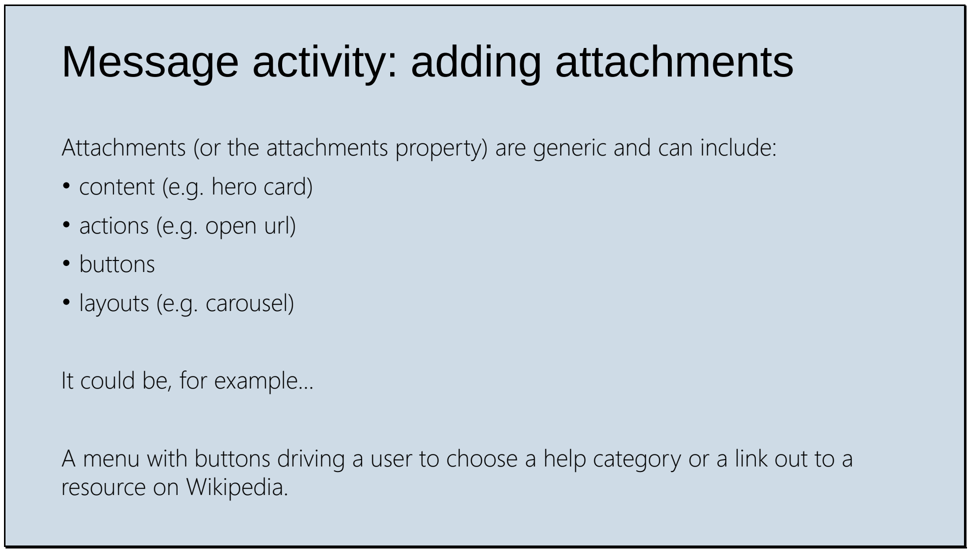


Message: text payloads default to markdown

Style	Markdown	Description	Example
Bold	**text**	make the text bold	**text**
Italic	*text*	make the text italic	*text*
Header1-5	# H1	Mark a line as a header	#H1
Strikethrough		make the text strikethrough	~~text~~
Hr	---	insert a horizontal rule	
Unordered list	*	Make an unordered list item	* text
Ordered list	1.	Make an ordered list item starting at 1	1. text
Pre	`text`	Preformatted text(can be inline)	`text`
Block quote	> text	quote a section of text	> text
link	[bing](http://bing.com)	create a hyperlink with title	bing
image link	![duck](http://aka.ms/Fo983c)	link to an image	

Nice markdown guide: https://daringfireball.net/projects/markdown/syntax

Slide 17



Message activity: adding attachments

Attachments (or the attachments property) are generic and can include:

• content (e.g. hero card)
• actions (e.g. open url)
• buttons
• layouts (e.g. carousel)

It could be, for example…

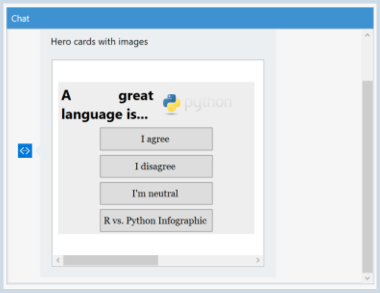
A menu with buttons driving a user to choose a help category or a link out to a resource on Wikipedia.

Attachments are just a construct and part of the object that is getting passed back and forth between a user and bot. See the AttachmentBot on bot-education github site.

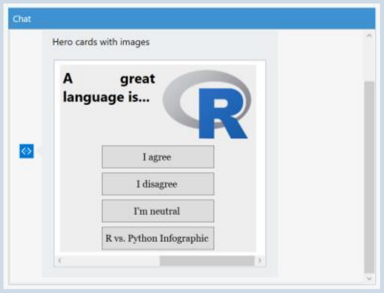
Slide 18

Message activity: adding attachments – an example

Scroll to the left – do you agree python is a great language? Here's a link to something…

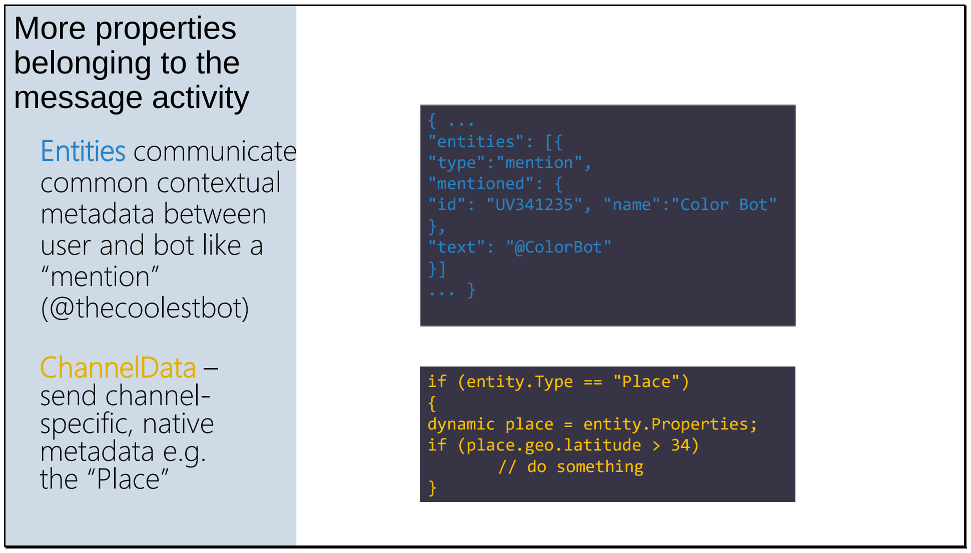


Scroll to the right – do you agree R is a great language? Here's a link to something…



A bot which adds to message activity, a carousel with cards, some buttons (one of which links out to an article). This bot code can be found on the bot-education repository: https://github.com/michhar/bot-education/tree/master/BOTs/Samples/CSharp/AttachmentBot which shows the use of the Attachments property in message activities by highlighting cards with buttons and associated actions.

Slide 19



More properties belonging to the message activity

Entities communicate common contextual metadata between user and bot like a “mention” (@thecoolestbot)

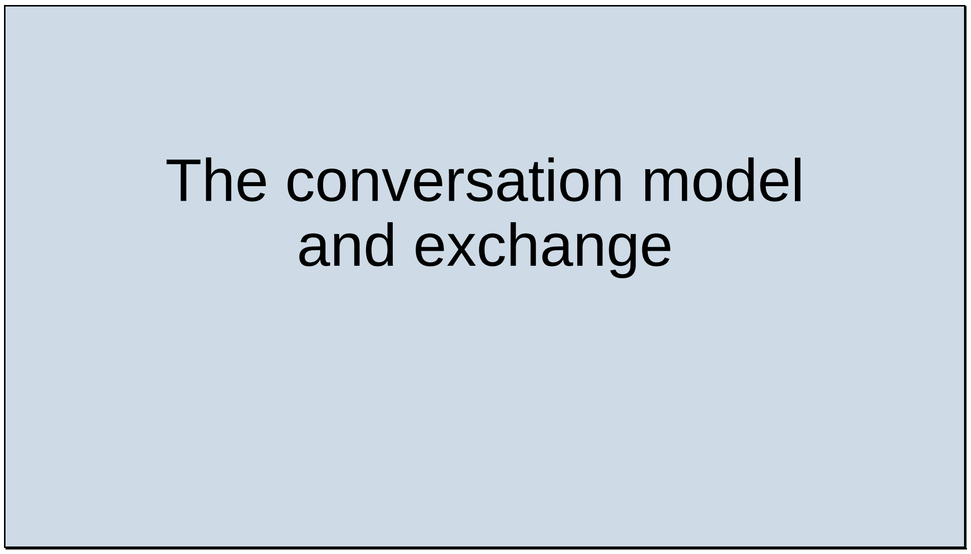
ChannelData – send channel-specific, native metadata e.g. the “Place”

```
{ ...
"entities": [{
"type":"mention",
"mentioned": {
"id": "UV341235", "name":"Color Bot"
},
"text": "@ColorBot"
}]
... }
```

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

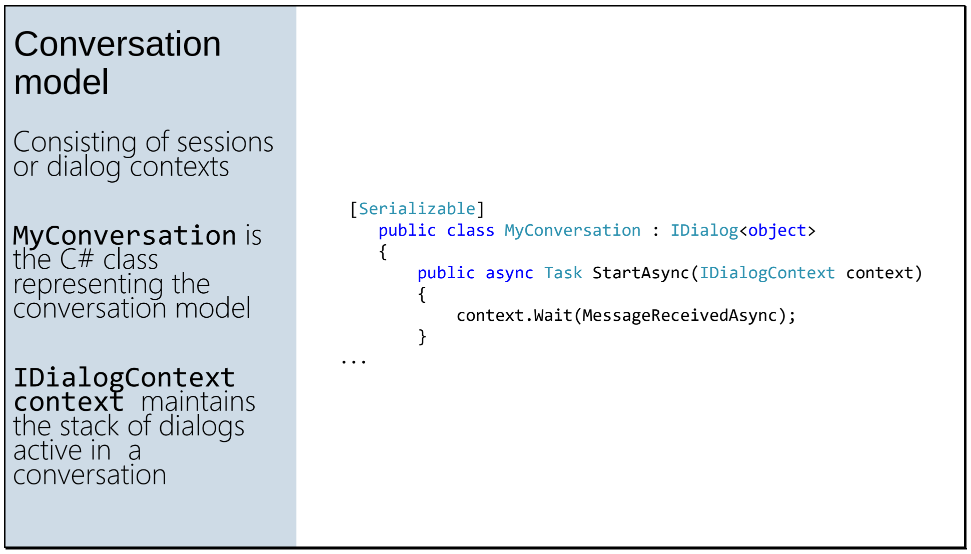
Entities use the standards found on schema.org and include entities and channel data properties. ChannelData allow the bot and user to leverage and enable functionalities provided by that particular channel. The “Place” entity makes it such that all messages always contain address and geo location in a channel specific way.

Slide 20



The conversation model and exchange

Slide 21



Conversation model

Consisting of sessions or dialog contexts

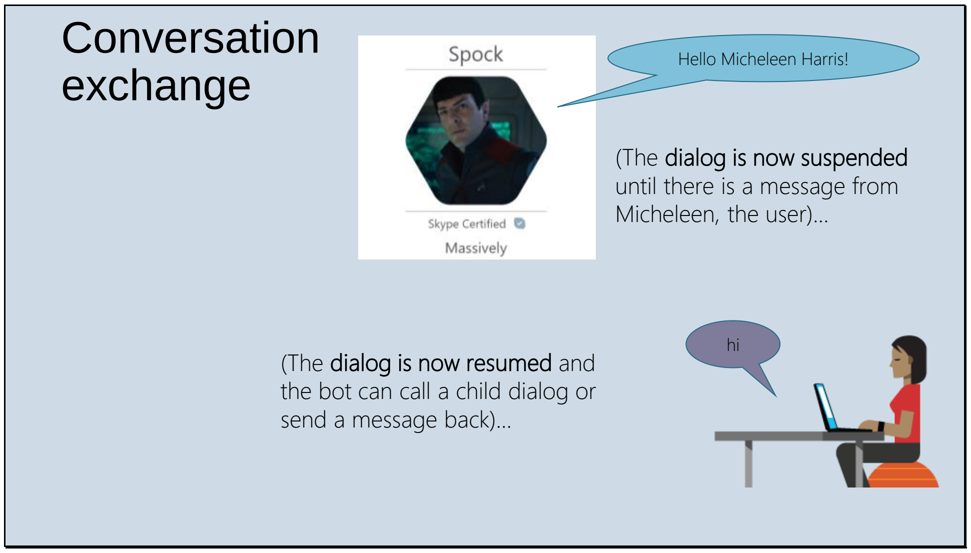
`MyConversation` is the C# class representing the conversation model

`IDialogContext context` maintains the stack of dialogs active in a conversation

```
[Serializable]
    public class MyConversation : IDialog<object>
    {
        public async Task StartAsync(IDialogContext context)
        {
            context.Wait(MessageReceivedAsync);
        }
...
```

aka session (Node) or dialog context (.NET)

Slide 22



Conversation exchange

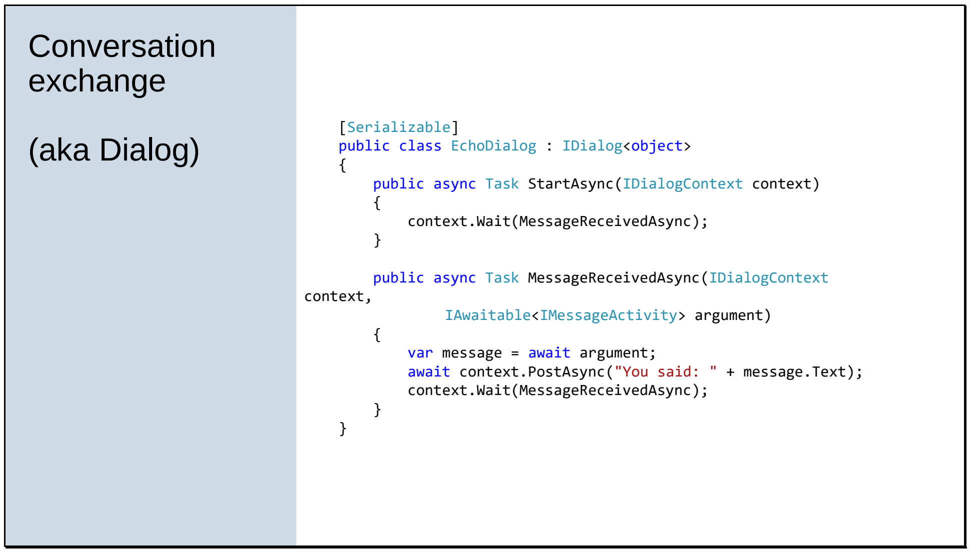
Spock
Skype Certified
Massively
Hello Micheleen Harris!
hi

(The **dialog is now suspended** until there is a message from Micheleen, the user)…

(The **dialog is now resumed** and the bot can call a child dialog or send a message back)…

Bot Builder lets you break your bots' conversation with a user into parts called dialogs. You can chain dialogs together to have sub-conversations with the user or to accomplish some micro task. node.js – prompts and even waterfalls are implemented internally as dialogs

Slide 23



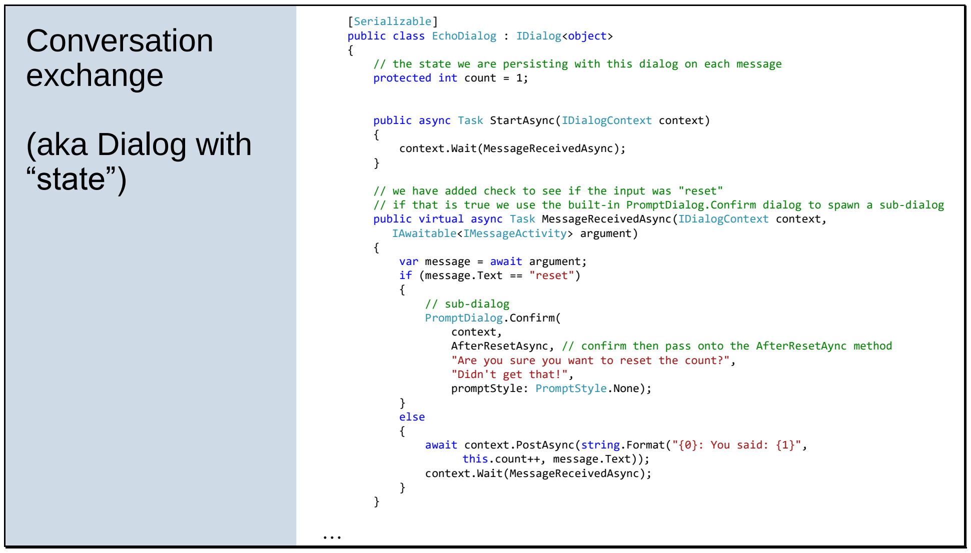
Conversation exchange

(aka Dialog)

```
    [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);
        }
    }
```

The context maintains stack of dialogs active in conversation.
Conversation state is the stack of active dialogs and each dialog’s state.
IDialog is the abstraction for the encapsulated dialog with it’s own state.

Slide 24



Conversation exchange

(aka Dialog with “state”)

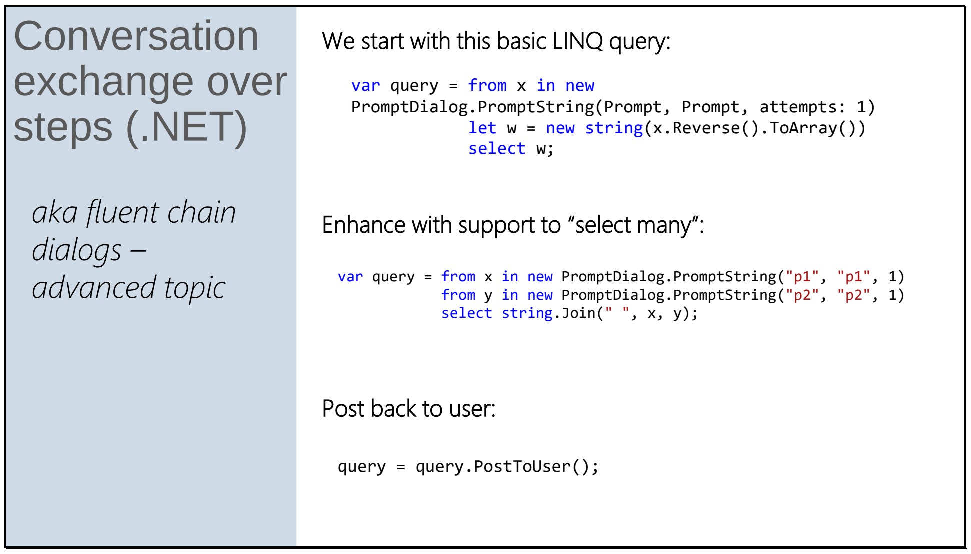
```
[Serializable]
public class EchoDialog : IDialog<object>
{
    // the state we are persisting with this dialog on each message
    protected int count = 1;


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

    // we have added check to see if the input was "reset"
    // if that is true we use the built-in PromptDialog.Confirm dialog to spawn a sub-dialog
    public virtual async Task MessageReceivedAsync(IDialogContext context,
       IAwaitable<IMessageActivity> argument)
    {
        var message = await argument;
        if (message.Text == "reset")
        {
            // sub-dialog
            PromptDialog.Confirm(
                context,
                AfterResetAsync, // confirm then pass onto the AfterResetAync method
                "Are you sure you want to reset the count?",
                "Didn't get that!",
                promptStyle: PromptStyle.None);
        }
        else
        {
            await context.PostAsync(string.Format("{0}: You said: {1}",
                  this.count++, message.Text));
            context.Wait(MessageReceivedAsync);
        }
    }

...
```

The variable, count, is persisted from one message to the next.

Good description of the conversation process from https://docs.botframework.com/en-us/csharp/builder/sdkreference/dialogs.html#echoBot:

In MessageReceivedAsync we have added check to see if the input was "reset" and if that is true we use the built-in Prompts.Confirm dialog to spawn a sub-dialog that asks the user if they are sure about resetting the count. The sub-dialog has its own private state and does not need to worry about interfering with the parent dialog. When the sub dialog is done, it's result is then passed onto the AfterResetAync method. In AfterResetAsync we check on the response and perform the action including sending a message back to the user. The final step is to do IDialogContext.Wait with a continuation back to MessageReceivedAsync on the next message.

Slide 25



Conversation exchange over steps (.NET)

aka fluent chain dialogs – advanced topic

We start with this basic LINQ query:

```
var query = from x in new
PromptDialog.PromptString(Prompt, Prompt, attempts: 1)
            let w = new string(x.Reverse().ToArray())
            select w;
```

Enhance with support to “select many”:

```
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);
```

Post back to user:

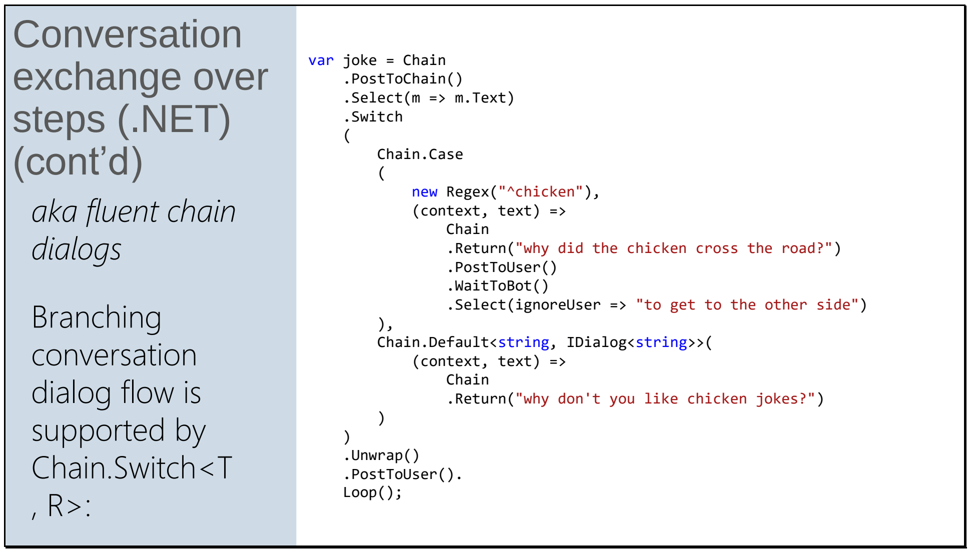
```
query = query.PostToUser();
```

It is also possible to implicitly manage the stack of active dialogs through the fluent Chain methods. (https://docs.botframework.com/en-us/csharp/builder/sdkreference/de/dab/class_microsoft_1_1_bot_1_1_builder_1_1_dialogs_1_1_chain.html)

Supported LINQ queries: “select”, “let”, “from”, “where”

.NET: (other option for “chaining”) Explicit management of the stack of active dialogs is possible through IDialogStack.Call<R> and IDialogStack.Done<R>, explicitly composing dialogs into a larger conversation.

Slide 26



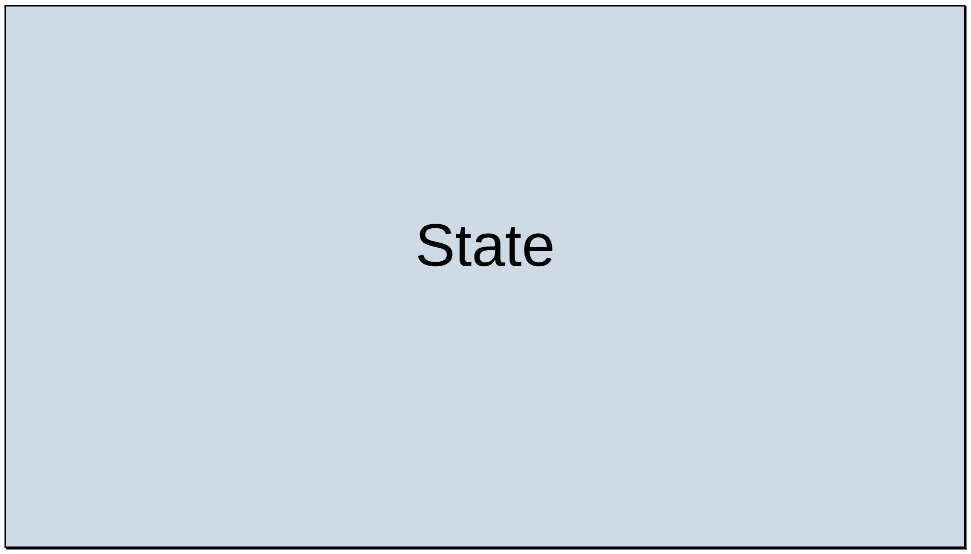
Conversation exchange over steps (.NET) (cont'd)

aka fluent chain dialogs

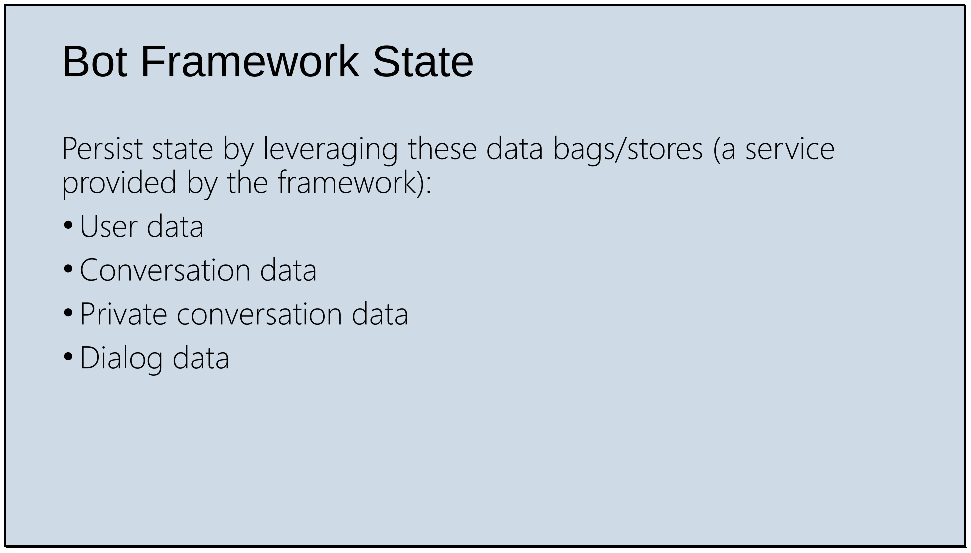
Branching conversation dialog flow is supported by Chain.Switch<T , R>:

```
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();
```

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 for branching in conversations to different paths of chained dialogs, possibly of unequal length. Here is a more complete example of branching dialogs written in the fluent chain style with implicit stack management.



State



Bot Framework State

Persist state by leveraging these data bags/stores (a service provided by the framework):

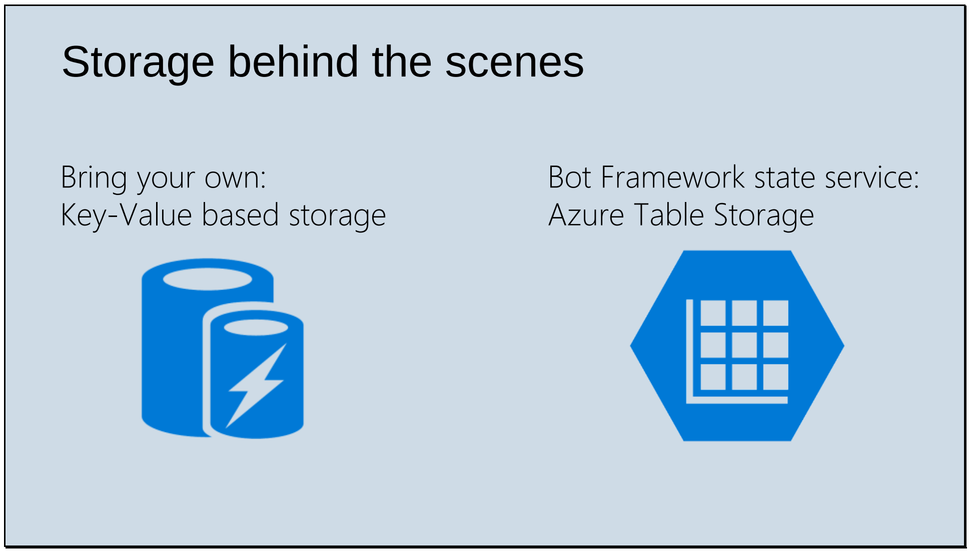
- User data
- Conversation data
- Private conversation data
- Dialog data

If I have a bot that plays Blackjack with me, my stats would be stored in user data (would follow me around from game to game), the deck information and stats in the conversation data (i.e. other players could use the same deck), and my hand in a game would be in user-conversation data (my immediate game's data).

The dialog data persistence ensures that the dialogs state is properly maintained between each turn of the conversation. Dialog data also ensures that a conversation can be picked up later and even on a different machine.

Anything can be stored in these data stores or bags, however it should be limited to data types that are serializable like primitives.

Slide 29

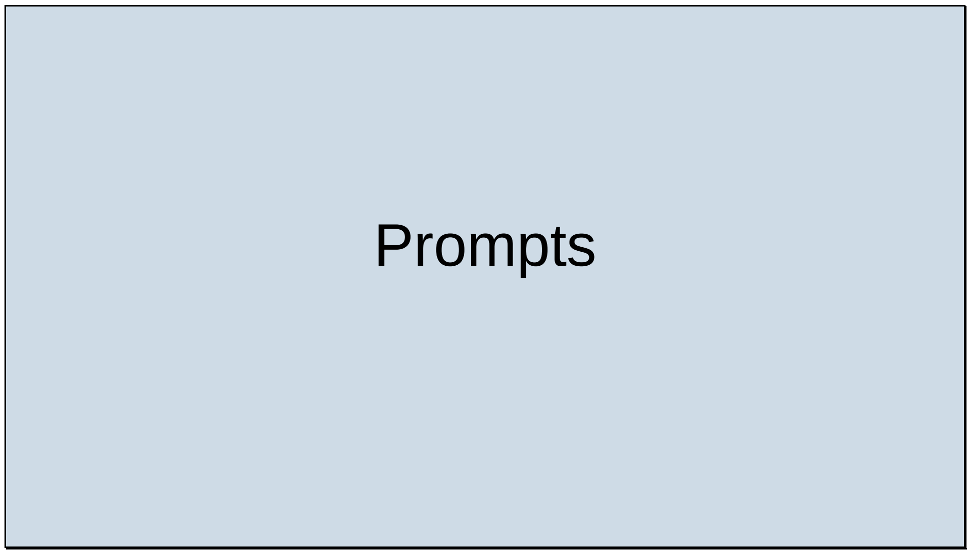


Storage behind the scenes

Bring your own:
Key-Value based storage

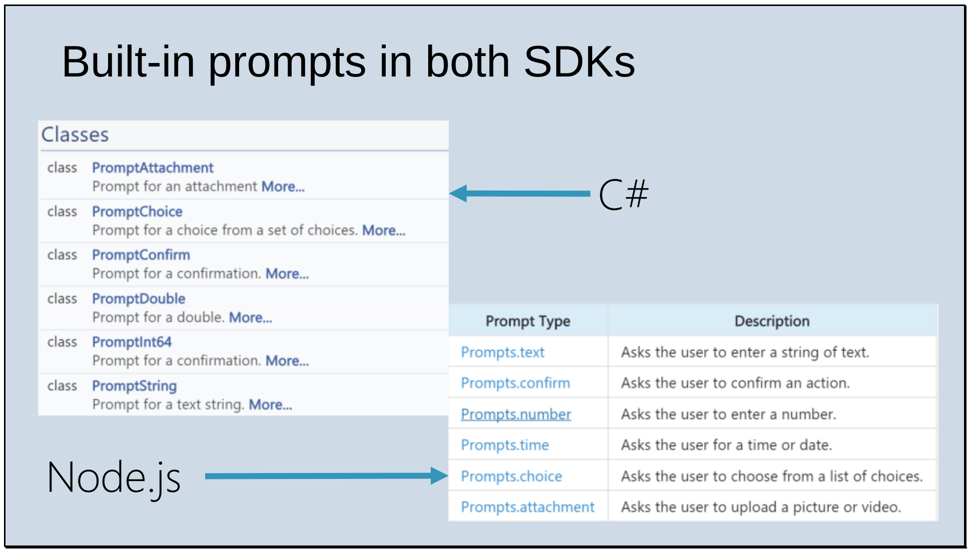
Bot Framework state service:
Azure Table Storage

The State REST API has wrappers built around Azure Table Storage. NB, you can bring your own storage in the form of a key-value store like Redis Cache. You don't really need to worry about the rest, just take note that table storage is used and you can bring your own if need-be. The Bot Framework manages this default storage for you so you can maintain a stateless bot experience.



Prompts

Slide 31



Built-in prompts in both SDKs

Classes

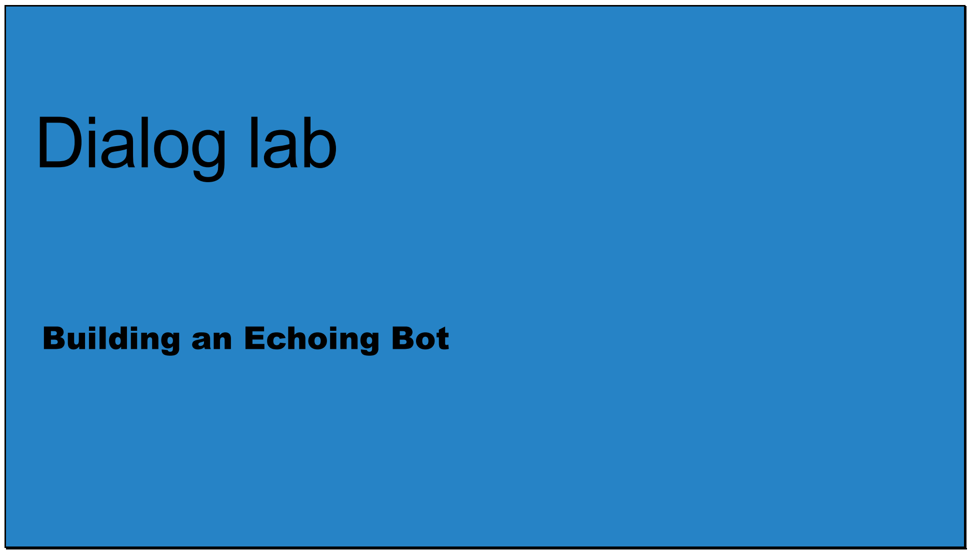
class	PromptAttachment Prompt for an attachment More...
class	PromptChoice Prompt for a choice from a set of choices. More...
class	PromptConfirm Prompt for a confirmation. More...
class	PromptDouble Prompt for a double. More...
class	PromptInt64 Prompt for a confirmation. More...
class	PromptString Prompt for a text string. More...

C#

Node.js

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.
Prompts.choice	Asks the user to choose from a list of choices.
Prompts.attachment	Asks the user to upload a picture or video.

A “dialog factory” for simple prompts to make obtaining user input easier.

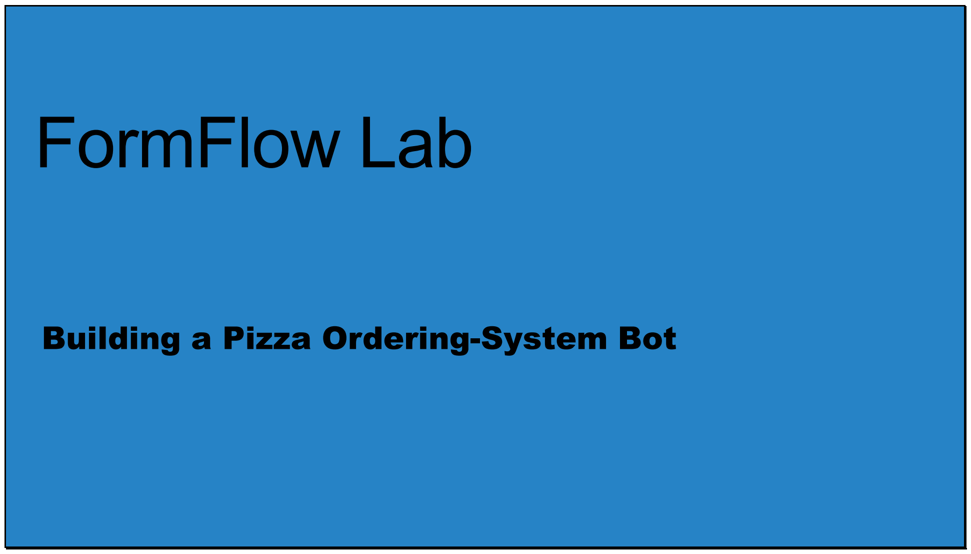


Dialog lab

Building an Echoing Bot

Lab link: https://github.com/michhar/bot-education/blob/master/labs/Lab_Getting_Started.md

Slide 33

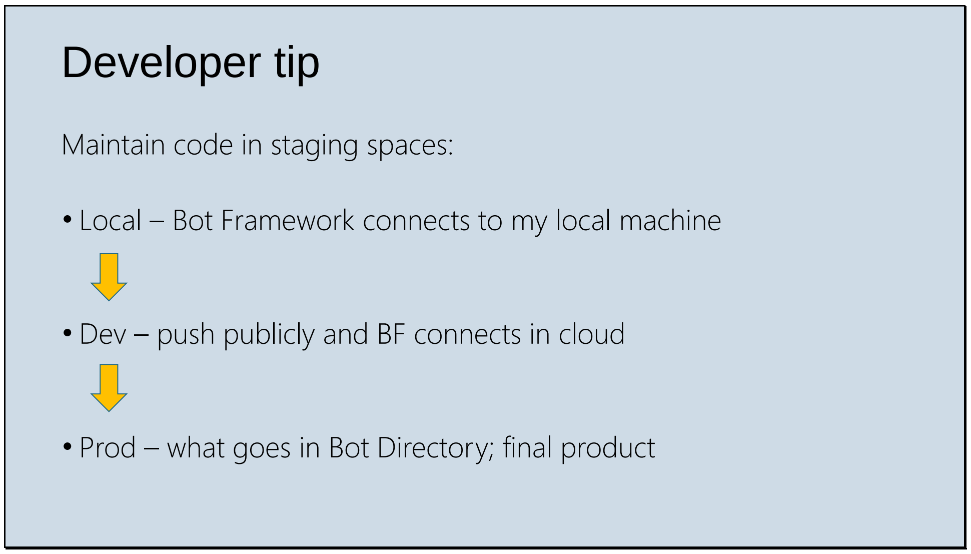


FormFlow Lab

Building a Pizza Ordering-System Bot

Lab link: https://github.com/michhar/bot-education/blob/master/labs/Lab_FormFlow.md

Slide 34



Developer tip

Maintain code in staging spaces:

- Local – Bot Framework connects to my local machine

⇩

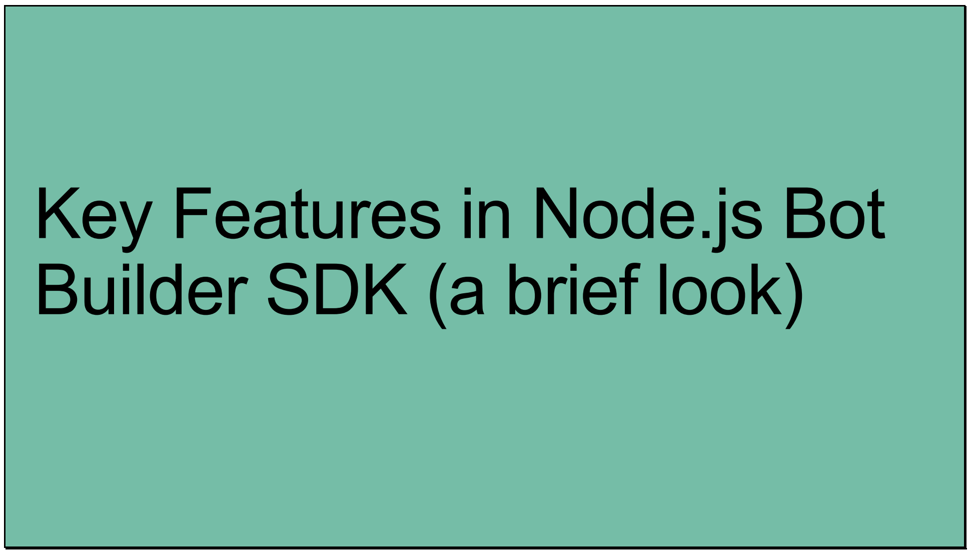
- Dev – push publicly and BF connects in cloud

⇩

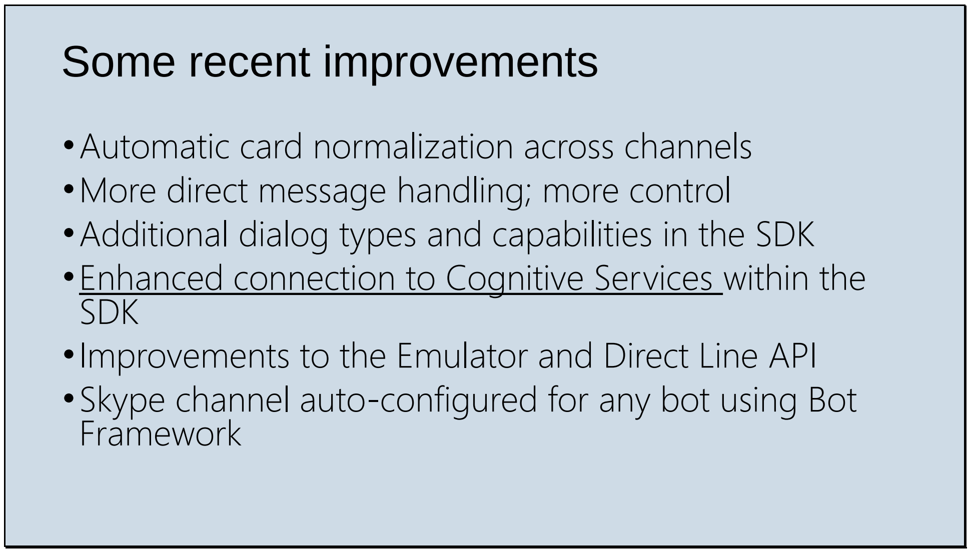
- Prod – what goes in Bot Directory; final product

Local – e.g. in Node.js can use ngrok as a local server tunnel service (get a free account if wish to ngrok https://dashboard.ngrok.com/user/signup)

Slide 35



Key Features in Node.js Bot Builder SDK (a brief look)

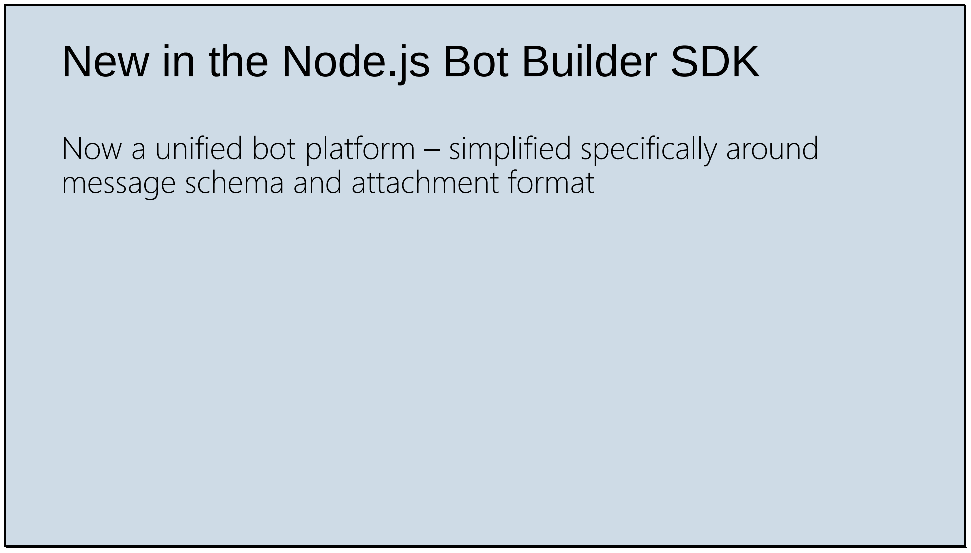


Some recent improvements

- Automatic card normalization across channels
- More direct message handling; more control
- Additional dialog types and capabilities in the SDK
- Enhanced connection to Cognitive Services within the SDK
- Improvements to the Emulator and Direct Line API
- Skype channel auto-configured for any bot using Bot Framework

For Node.js: https://docs.botframework.com/en-us/node/builder/whats-new/

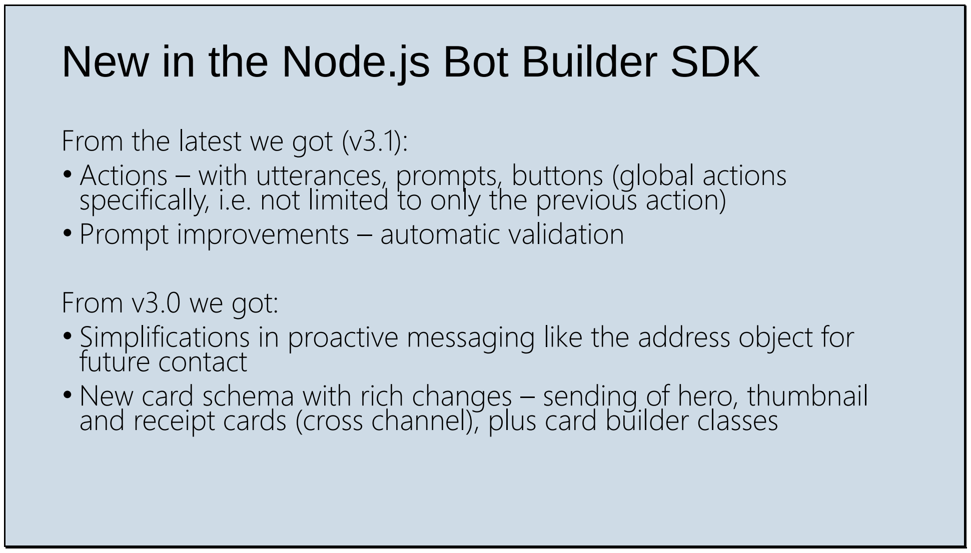
Direct Line API:



New in the Node.js Bot Builder SDK

Now a unified bot platform – simplified specifically around message schema and attachment format

message schema + attachment format jointly called v3 schema, hence v3.0



New in the Node.js Bot Builder SDK

From the latest we got (v3.1):

- Actions – with utterances, prompts, buttons (global actions specifically, i.e. not limited to only the previous action)
- Prompt improvements – automatic validation

From v3.0 we got:

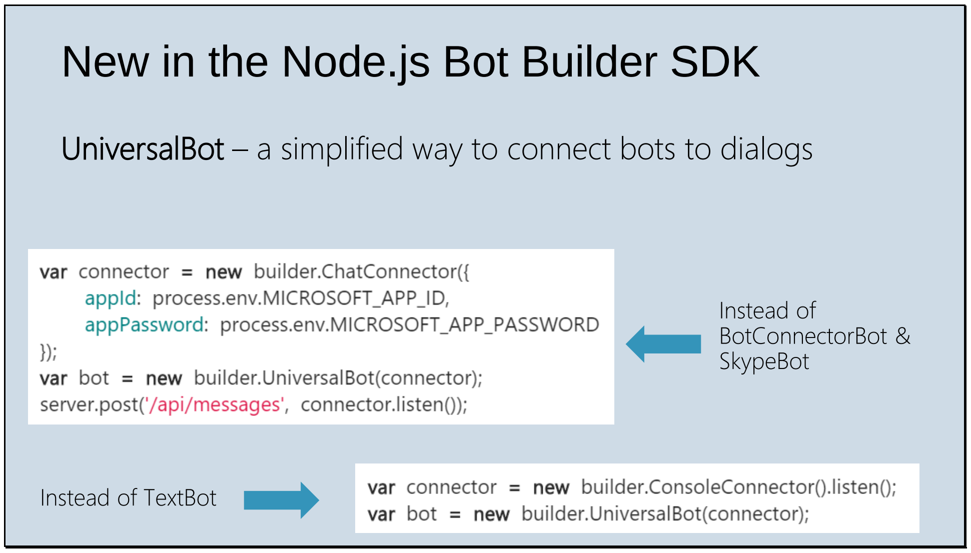
- Simplifications in proactive messaging like the address object for future contact
- New card schema with rich changes – sending of hero, thumbnail and receipt cards (cross channel), plus card builder classes

Actions
simplifies things like starting a new dialog, sending a message, a new prompt with validation and other common tasks
global actions introduced – e.g. saying “help” or “good bye” anywhere in the conversation; tagging a complex order as cancellable; custom button triggers and utterences

- updates and changes from https://docs.botframework.com/en-us/node/builder/whats-new/

Prompt Improvements

Slide 39



New in the Node.js Bot Builder SDK

UniversalBot – a simplified way to connect bots to dialogs

```
var connector = new builder.ChatConnector({
    appId: process.env.MICROSOFT_APP_ID,
    appPassword: process.env.MICROSOFT_APP_PASSWORD
});
var bot = new builder.UniversalBot(connector);
server.post('/api/messages', connector.listen());
```

Instead of BotConnectorBot & SkypeBot

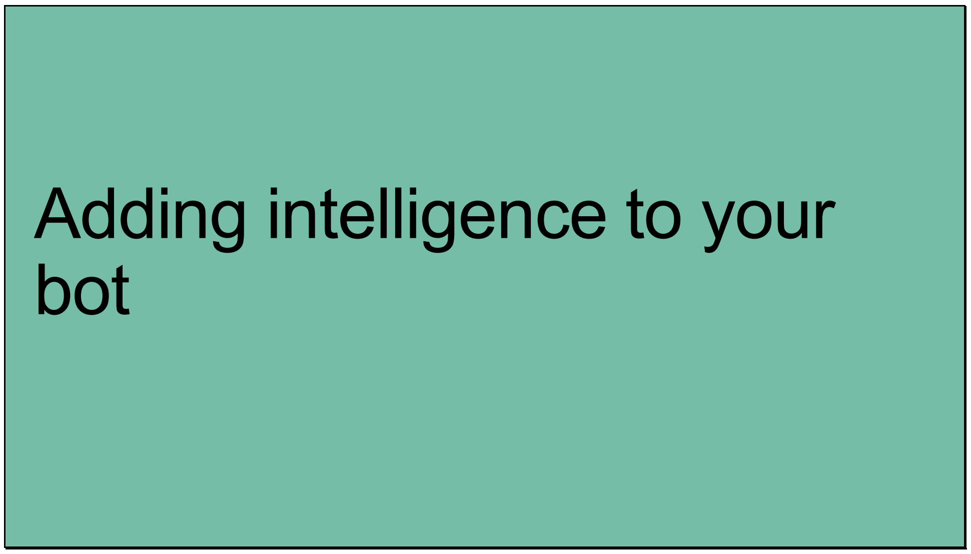
Instead of TextBot

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

UniversalBot (https://docs.botframework.com/en-us/node/builder/chat-reference/classes/_botbuilder_d_.universalbot)

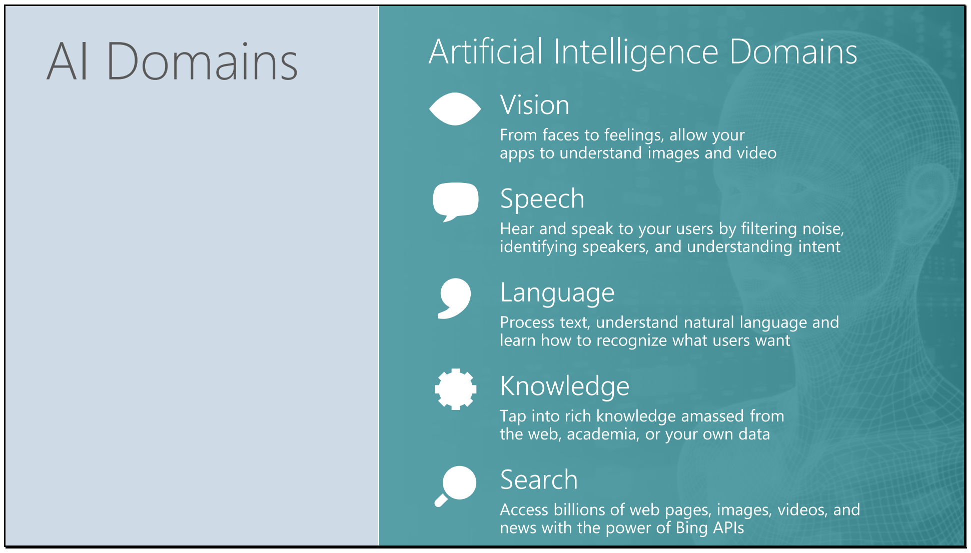
- has a lightweight connector model and includes ChatConnector and ConsoleConnector classes
- your bot can even utilize both the ChatConnector and ConsoleConnector and others at the same time if so desired
- replaces and unifies old classes like BotConnectorBot and TextBot

- updates and changes from https://docs.botframework.com/en-us/node/builder/whats-new/



Adding intelligence to your bot

Slide 41



AI Domains

Artificial Intelligence Domains

Vision
From faces to feelings, allow your apps to understand images and video

Speech
Hear and speak to your users by filtering noise, identifying speakers, and understanding intent

Language
Process text, understand natural language and learn how to recognize what users want

Knowledge
Tap into rich knowledge amassed from the web, academia, or your own data

Search
Access billions of web pages, images, videos, and news with the power of Bing APIs

What are Cognitive Services? Microsoft Cognitive Services are a new collection of intelligence and knowledge APIs that enable developers to ultimately build smarter apps.

NOTES: *key concepts we are trying to convey in this above statement:*

- *That Microsoft Cognitive Services collection is new, not a rebrand of Project Oxford, as we are bringing together Bing, Oxford, AzureML, and Translator APIs.*
- *That we are bringing together Intelligence (Oxford) and Knowledge from the corpus of the web (Bing)*
- *That cognitive = human perception and understanding, enabling your apps to see the world around them, to hear and talk back with the users—to have a human side.*

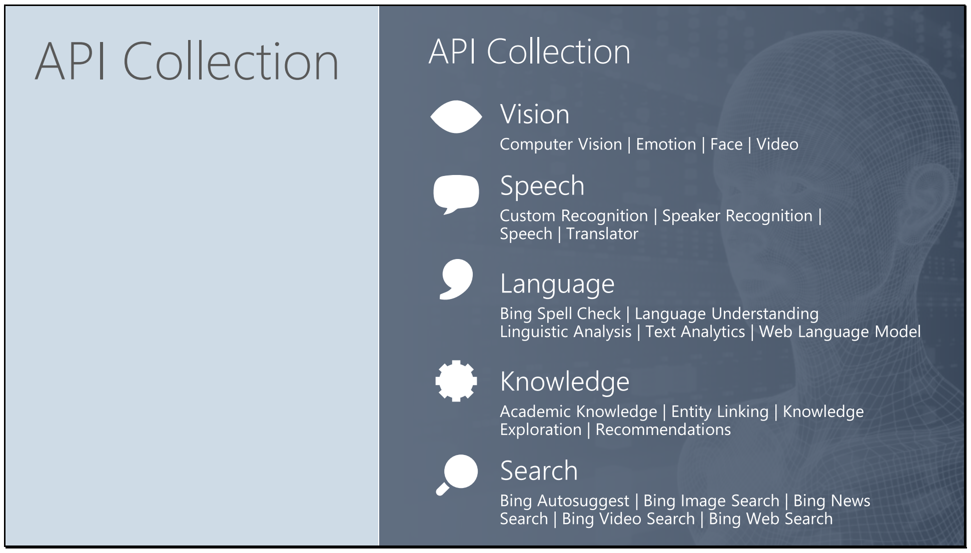
What are Microsoft Cognitive Services?
Microsoft Cognitive Services is a new collection of intelligent APIs that allow systems to see, hear, speak, understand and interpret our needs using natural methods of communication. Developers can use these APIs to make their applications more intelligent, engaging and discoverable. To try Cognitive Services for free, visit www.microsoft.com/cognitive.

With Cognitive Services, developers can easily add intelligent features – such as emotion and sentiment detection, vision and speech recognition, knowledge, search and language understanding – into their applications. The collection will continuously improve, adding new APIs and updating existing ones.

Cognitive Services includes:
Vision: From faces to feelings, allow apps to understand images and video

Speech: Hear and speak to users by filtering noise, identifying speakers, and understanding intent
Language: Process text and learn how to recognize what users want
Knowledge: Tap into rich knowledge amassed from the web, academia, or your own data
Search: Access billions of web pages, images, videos, and news with the power of Bing APIs

Slide 42



API Collection

API Collection

Vision

Computer Vision | Emotion | Face | Video

Speech

Custom Recognition | Speaker Recognition | Speech | Translator

Language

Bing Spell Check | Language Understanding Linguistic Analysis | Text Analytics | Web Language Model

Knowledge

Academic Knowledge | Entity Linking | Knowledge Exploration | Recommendations

Search

Bing Autosuggest | Bing Image Search | Bing News Search | Bing Video Search | Bing Web Search

At Microsoft, we've been offering APIs for a very long time across the company. In delivering Microsoft Cognitive Services API, we started with 4 last year at /build (2015); added 7 more last December, and today we have 22 APIs in our collection.

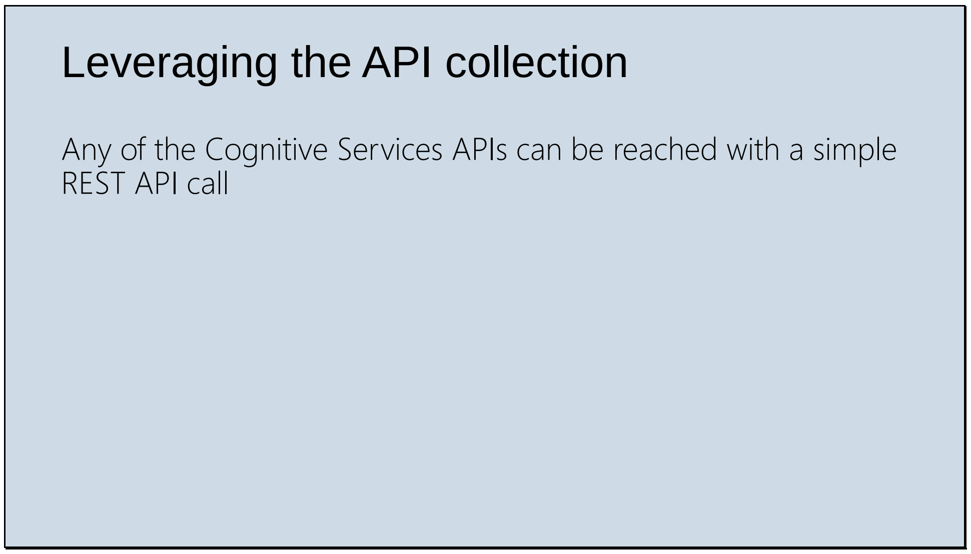
Cognitive Services are available individually or as a part of the Cortana Intelligence Suite, formerly known as Cortana Analytics, which provides a comprehensive collection of services powered by cutting-edge research into machine learning, perception, analytics and social bots.

These APIs are powered by Microsoft Azure.

Cognitive Services code name was Project Oxford and was officially moved over as of /build 2016.

Developers and businesses can use this suite of services and tools to create apps that learn about our world and interact with people and customers in personalized, intelligent ways.

Slide 43



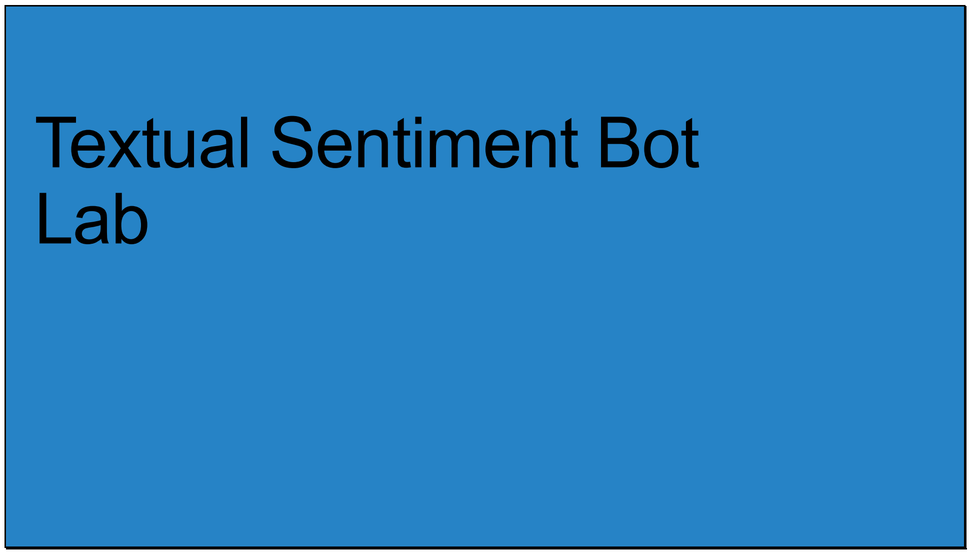
Leveraging the API collection

Any of the Cognitive Services APIs can be reached with a simple REST API call

Language integration with LUIS (for NLP based in intents) is easiest as a NuGet package exists within the bot builder for easy integration, however any of the Cognitive Services APIs can be reached with a simple REST API call.

my notes:
Let’s go straight to an example of implementing a Cognitive Services API call from a bot.

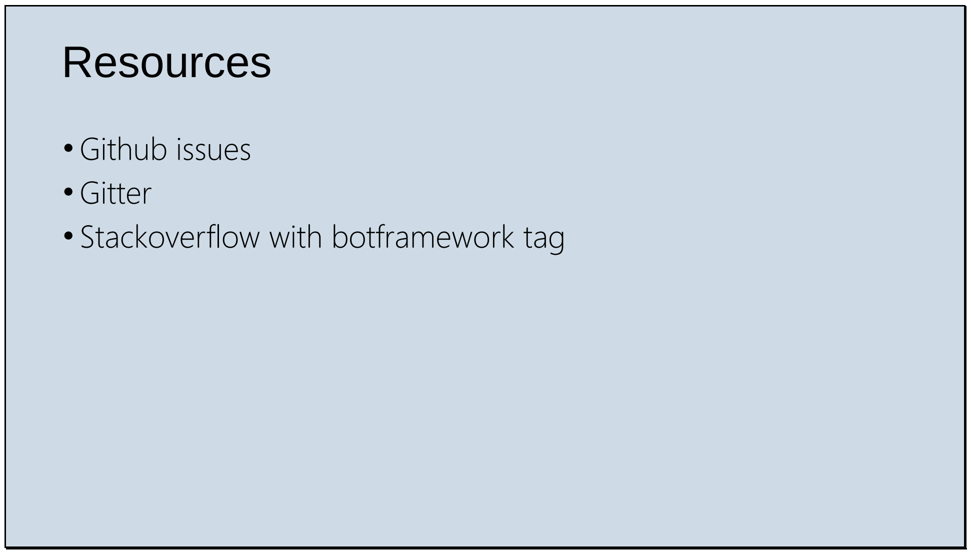
Slide 44



Textual Sentiment Bot Lab

Lab link: https://github.com/michhar/bot-education/blob/master/labs/Lab_Text_Analytics_Bot.md

Slide 45

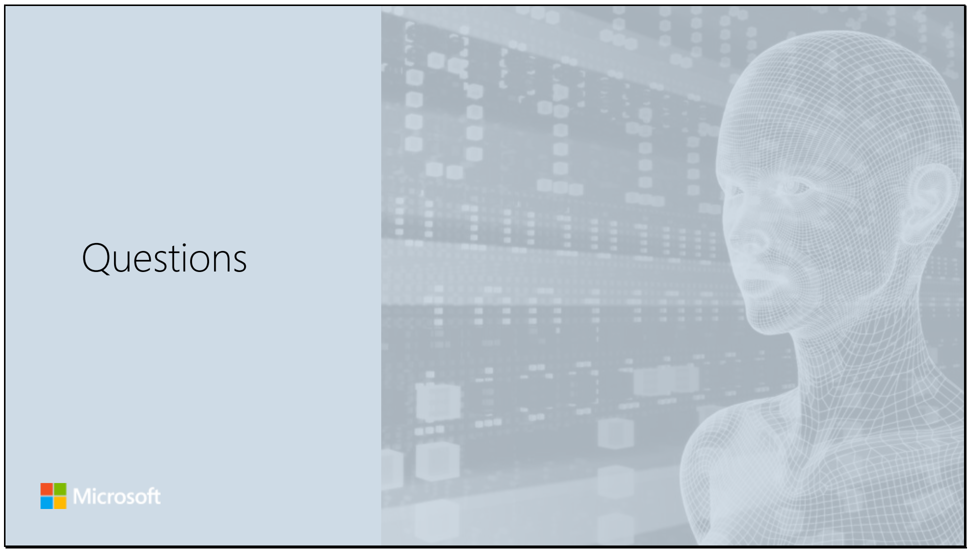


Resources

- Github issues
- Gitter
- Stackoverflow with botframework tag

https://github.com/Microsoft/BotBuilder/issues
https://gitter.im/Microsoft/BotBuilder
http://stackoverflow.com/questions/tagged/botframework

Slide 46



Questions

Microsoft