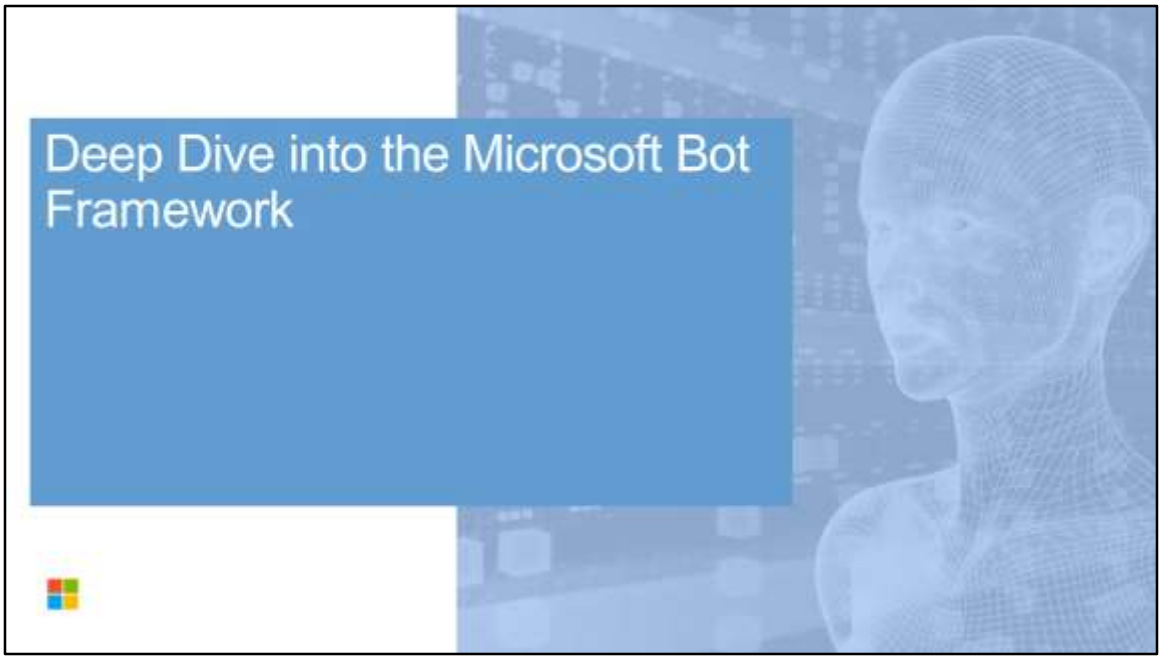


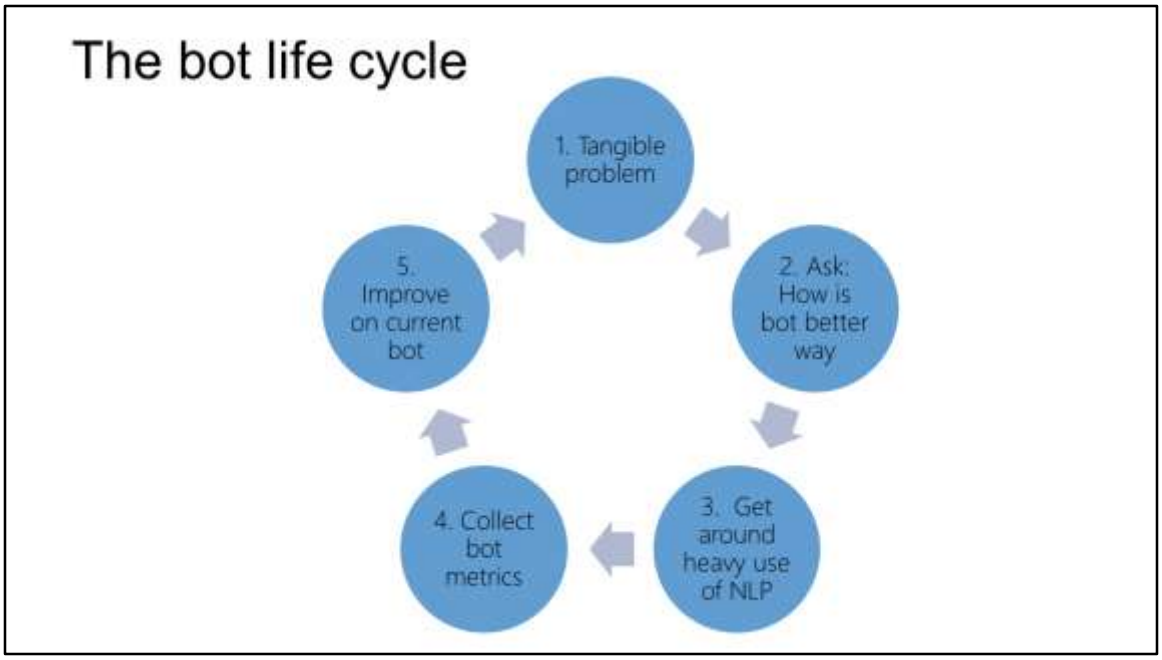
If the links in this deck are broken please let us know (mailto: michhar@Microsoft.com). Thanks in advance and enjoy learning about bots and the Microsoft Bot Framework.

Session outline

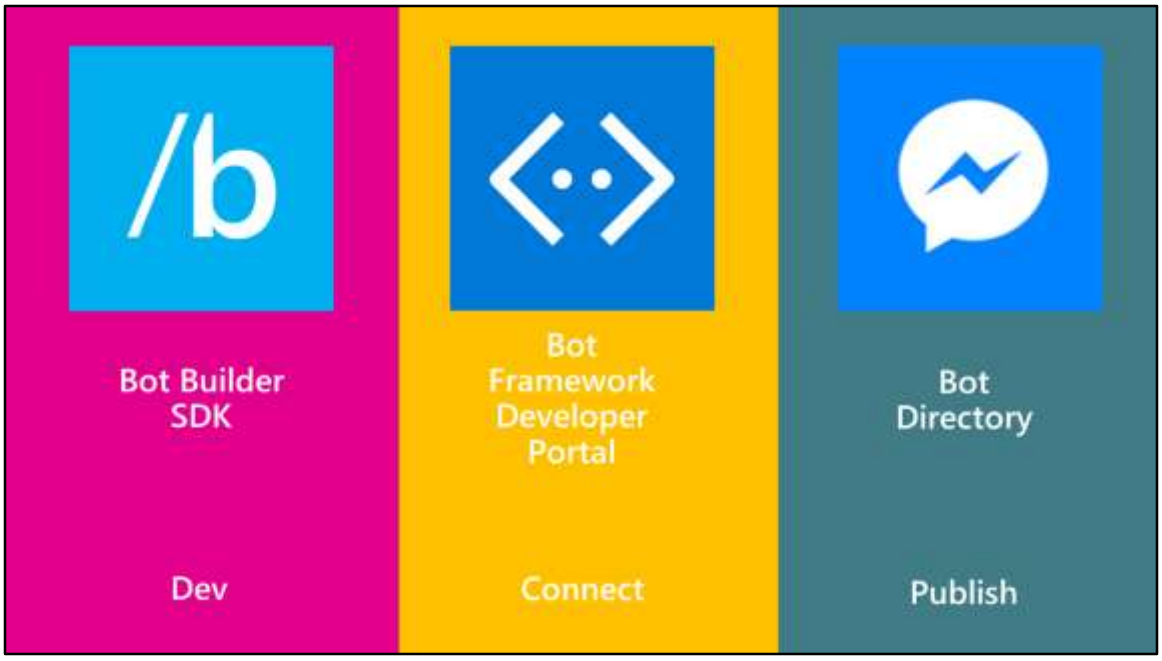
- Overview refresher
- Basics
- *QnA bot lab*
- Conversations and their features
- Conversation model
- *Dialog lab*
- *Attachments lab*
- State storage
- *Nonlinear waterfall lab*
- *Intents lab part 1*
- Adding intelligence
- *Intents lab part 2*

aka.ms/botedu aka.ms/bot-discuss

Overview Refresher



1. Start by asking what problem are we trying to solve. Refine until it looks like a tangible problem and not "magic"
2. Ask how a bot will be a better experience. User experience is EVERYTHING
3. Avoid too much natural language. Careful with unrealistic expectations. Natural language recognition is limited. Menus work great. Commands work great. Buttons, etc.
4. You can only analyze and improve your bot if you're collecting metrics for it
5. Iterate, improve



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

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.

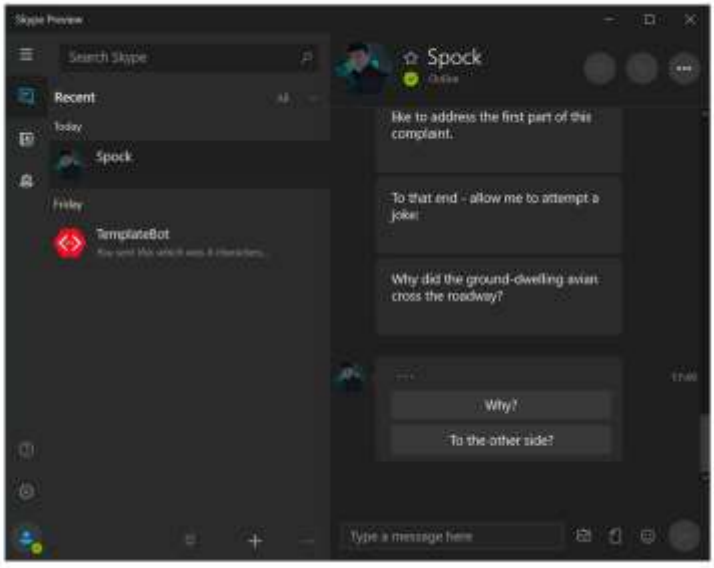
The Bot Framework Developer Portal lets you connect your bot(s) seamlessly text/sms to Skype, Slack, Facebook Messenger, Kik, Office 365 mail and other popular services. Register, configure and publish.

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

NB: 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

Bot Builder

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



my notes:
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.

Bot Builder

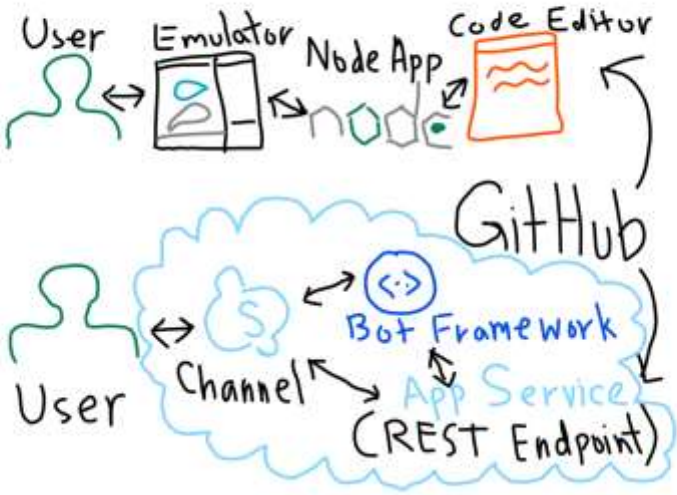
Bot Builder is itself a framework for building conversational applications ("Bots").

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

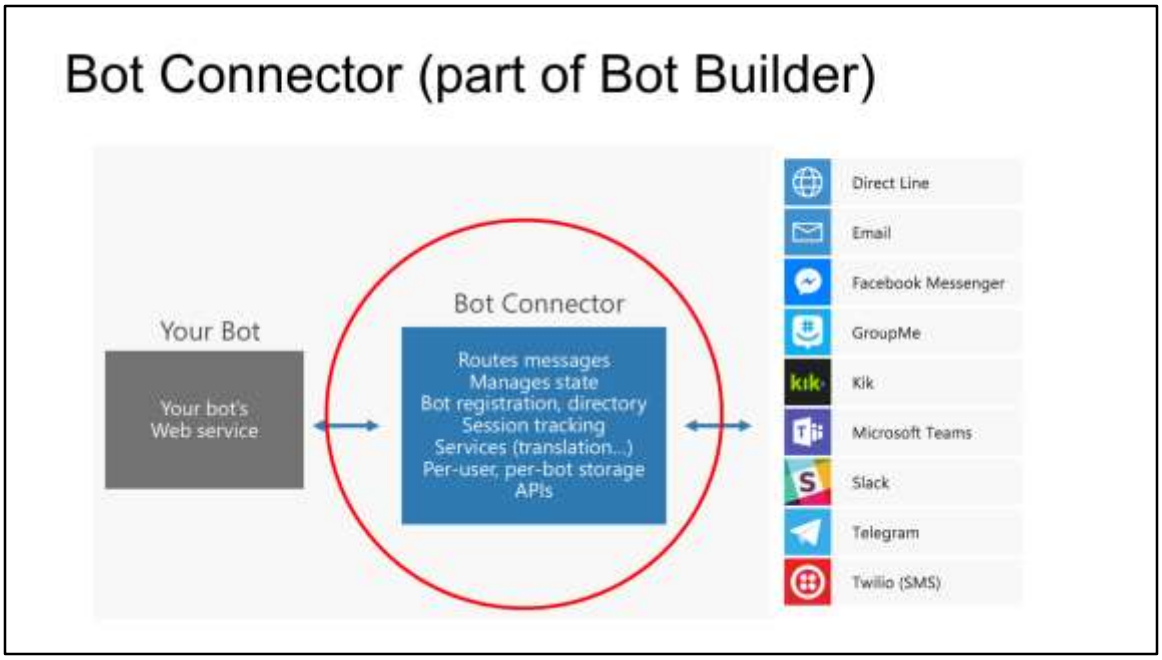
my notes:
a bot is just an application that addresses the huge effort involved in repetitive tasks (a bot can generalize a problem – provide a fun interface to a game or experience for instance, or collect votes over a choice of opinions – as a bot it's a deployable and repeatable solutions, plus hopefully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

Dev process/Toolbox



Releasing a bot:

1. Develop
2. Deploy
3. Register
4. Publish
5. Test

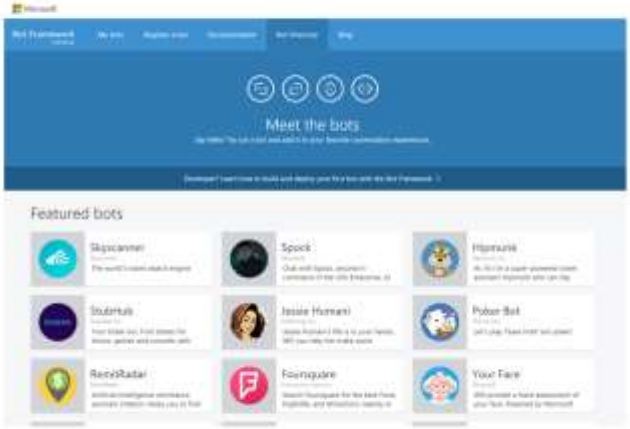


It's part of the Bot Builder SDK
http://docs.botframework.com/en-us/csharp/builder/sdkreference/gettingstarted.html#channels

Bot Directory

Public Directory of Bot Framework Bots

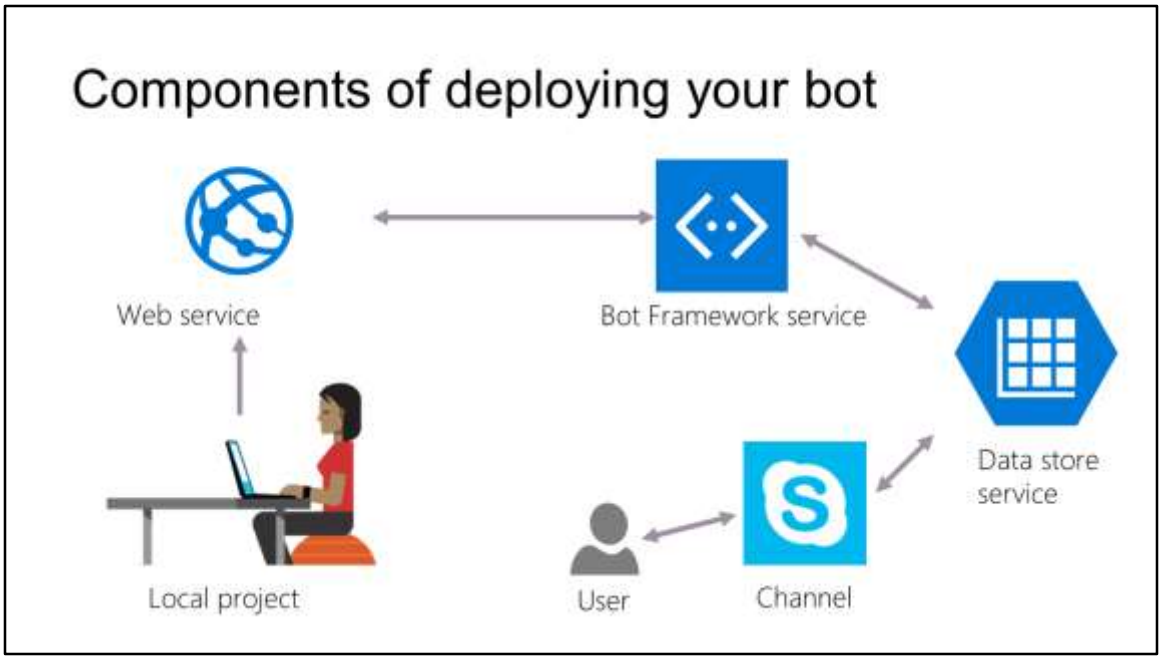
- Discover, try, and add bots from here with no added configuration
- Bots are public at developer discretion; must be reviewed
- Searchable here



Bots must be submitted for review and approved in order to appear in the directory

As of Nov. 17, 2016 there are 94 bots in the directory

from MS and other companies



Steps:
The developer writes their bot code, leveraging the BF libraries in the SDKs and APIs available
Create an endpoint for the bot to talk to in the cloud
Connect to the Bot Framework service (Connector and State services)
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

Basics

Code snippets

my notes:
You've seen some code so far, so let's dive in to the .NET Bot Builder again.

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

```
// Instatiate the chat connector to route messages
var connector = new builder.ChatConnector(botConnectorOptions);

// Handle Bot Framework messages with a restify server
var server = restify.createServer();
server.post('/api/messages', connector.listen());
server.listen(process.env.port || process.env.PORT || 3978, function () {
    console.log('%s listening to %s', server.name, server.url);
});
```

Note the explicit creation of the connector client

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 (with "bot.dialog")

"session" is our conversation model

"bot.dialog" is the dialog handler

```
// Think of a sandwich (the bread is the root dialog and meat/cheese the 'askName' dialog)
bot.dialog('/', [
    function (session) {
        // "push" onto the dialog stack
        session.beginDialog('/askName');
    },
    function (session, results) {
        session.send('Hello %s!', results.response);
    }
]);

bot.dialog('/askName', [ // this is a small waterfall
    function (session) {
        builder.Prompts.text(session, 'Hi! What is your name?');
    },
    function (session, results) {
        // "pop" off of the dialog stack and return the results
        session.endDialogWithResult(results);
    }
]);
```

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

1. Instantiate the required components.
2. Deserialize the dialog state (the dialog stack in the storage "bags" and each dialog's state)
3. Resume the conversation processes where the Bot decided to suspend and wait for a message.
4. Send the replies.
5. Serialize the updated dialog state and storage "bags" and persist it back to the state store.

Notes:
The dialog process can involve sub-dialogs

In between 2 and 3 a dialog 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.

Conversation exchange

The **waterfall** is an *array of functions* that chain together to form steps in a dialog

```
// What's different about this bot setup? (this is the new style, btw, for v3.5.0)
var bot = new builder.UniversalBot(connector, [
    function (session) {
        // Introducing a builtin prompt
        builder.Prompts.text(session, 'Hi! What is your name?');
    },
    function (session, results) {
        // Step 2 in the waterfall
        session.send('Hello %s!', results.response);
    }
]);
```

By passing an array of functions for our dialog handler a waterfall is setup where the results of the first function are passed to the input of the second function. We can chain together a series of these functions into steps that create waterfalls of any length.

Wait for user input

Use a **prompt** to *wait* and obtain user input.

Use a **built-in prompt** for expected types.

```
// Setup bot and root waterfall - as this old or new style?
var bot = new builder.UniversalBot(connector, [
    function (session) {
        builder.Prompts.text(session, "Hello... What's your name?");
    },
    function (session, results) {
        session.userData.name = results.response;
        builder.Prompts.number(session, "Hi " + results.response + ". How many years have you been coding?");
    },
    function (session, results) {
        session.userData.coding = results.response;
        builder.Prompts.choice(session, "What language do you code Node using?", ["JavaScript", "CoffeeScript",
            "TypeScript"]);
    },
    function (session, results) {
        session.userData.language = results.response.entity;
        session.send("Got it... " + session.userData.name +
                     " you've been programming for " + session.userData.coding +
                     " years and use " + session.userData.language + ".");
    }
]);
```

To actually wait for the users input we're using one of the SDK's built in prompts. We're using a simple text prompt which will capture anything the user types but the SDK a wide range of built-in prompt types.

Persistent state data

State storage, managed by State API

Also, called storage "bags"

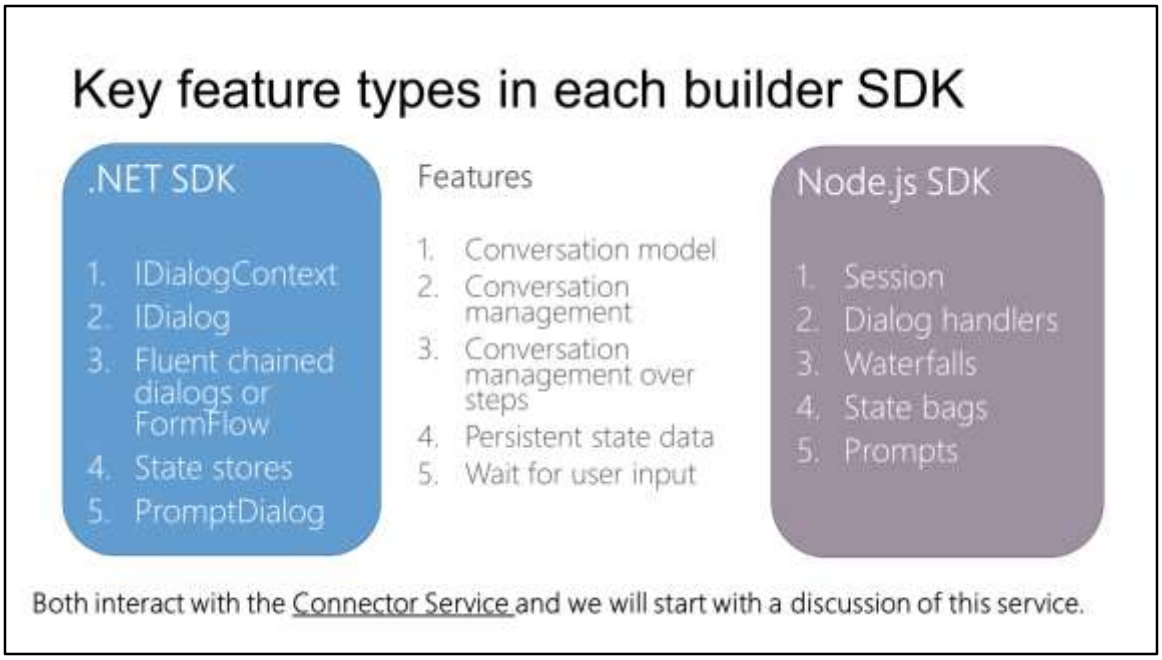
```
// Setup bot and root waterfall - is this old or new style?
var bot = new builder.UniversalBot(connector, [
    function (session) {
        builder.Prompts.text(session, "Hello... What's your name?");
    },
    function (session, results) {
        session.userData.name = results.response;
        builder.Prompts.number(session, "Hi " + results.response + ", How many years have you been coding?");
    },
    function (session, results) {
        session.userData.coding = results.response;
        builder.Prompts.choice(session, "What language do you code Node using?", ["JavaScript", "CoffeeScript",
            "TypeScript"]);
    },
    function (session, results) {
        session.userData.language = results.response.entity;
        session.send("Got it... " + session.userData.name +
                     " you've been programming for " + session.userData.coding +
                     " years and use " + session.userData.language + ".");
    }
]);
```

Remember our:
User data
Conversation data
Private conversation data
Dialog data

Conversations and their key features

Overview

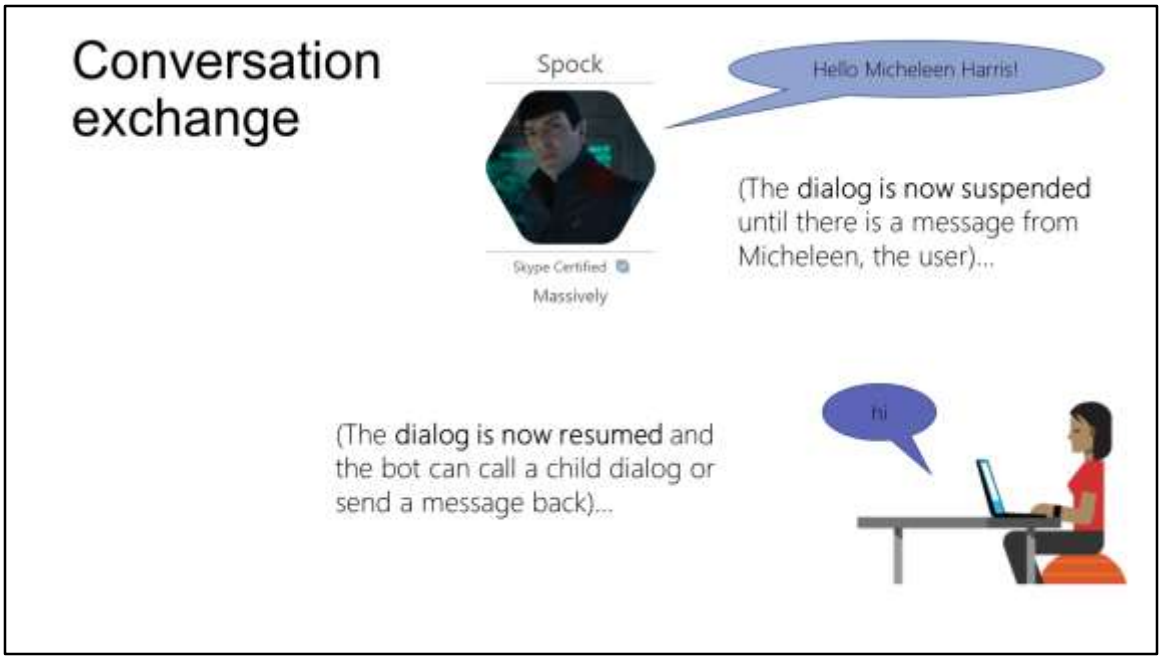
Conversation – dialog or stack of dialogs handled by a model



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

Conversation management

Dialog handlers



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

UniversalBot

Manages all conversations between a bot and user(s)

- Connectors
- Proactive messaging

https://docs.botframework.com/en-us/node/builder/chat/UniversalBot/#navtitle

Connecting the bot to a framework

Two methods

1) Connect to the **local console** (e.g. `ConsoleConnector` class)

2) Connect to the **Bot Framework (or emulate it at least)** (e.g. `ChatConnector` class)

https://docs.botframework.com/en-us/node/builder/chat/UniversalBot/#connectors

Modes of message delivery

Two types

1) **Reactive** – in response to a user message (e.g. `session.send()`)

2) **Proactive** – in response to some external event (e.g. `bot.send()` or `bot.beginDialog()`)

Reactive is most common method delivery. A "pull" bot. (We've seen this type of code earlier – it's what we were doing)

But there's the option of "pushing" a message to a user based on an external event like if a product has shipped or there's a weather alert. We first need the "address" of the user so we can find them later.

https://docs.botframework.com/en-us/node/builder/chat/UniversalBot/#proactive-messaging

Starting conversations – the push bot

Options

1) You don't need to wait for a reply (e.g. `bot.send()`)
2) You do need to wait for a reply (e.g. `bot.beginDialog()`)

* Notice, we are not dealing with the session, but rather the bot object

Are we waiting for a reply or not here?

```
server.post('/api/notify', function (req, res) {
    // Process posted notification
    var address = JSON.parse(req.body.address);
    var notification = req.body.notification;

    // Send notification as a proactive message
    var msg = new builder.Message().address(address).text(notification);
    bot.send(msg, function (err) {
        // Return success/failure
        res.status(err ? 500 : 200);
        res.end();
    });
});
```

The differences between **bot.send()** and **bot.beginDialog()** are subtle.

Calling **bot.send()** with a message won't affect the state of any existing conversation between the bot & user so it's generally safe to call at any time.

Calling **bot.beginDialog()** will end any existing conversation between the bot & user and start a new conversation using the specified dialog.

As a general rule you should call **bot.send()** if you don't need to wait for a reply from the user and **bot.beginDialog()** if you do.

https://docs.botframework.com/en-us/node/builder/chat/UniversalBot/#starting-conversations

Dialogs and handlers

Conversational exchange elements

- Dialog object
- Closure
- Waterfalls
- other forms (e.g. graph extension as in bot-trees)

Dialog object – conversational exchange object
Closure – passed in single functions to create a 1-step waterfall which loops

Waterfalls – conversational exchange over steps
other forms (e.g. graph extension as in bot-trees)

Dialog handler example

bot.dialog object handles the message

A child dialog is made

```
// think of a sandwich (the bread is the root dialog and meat/cheese the 'askName' dialog)
bot.dialog('/', [
    function (session) {
        // "push" onto the dialog stack
        session.beginDialog('/askName');
    },
    function (session, results) {
        session.send('Hello %s!', results.response);
    }
]);

bot.dialog('/askName', [ // this is a small waterfall
    function (session) {
        builder.Prompts.text(session, 'Hi! What is your name?');
    },
    function (session, results) {
        // "pop" off of the dialog stack and return the results
        session.endDialogWithResult(results);
    }
]);
```

To understand dialogs its easiest to think of them as the equivalent of routes for a website.

All bots will have at least one root '/' dialog just like all websites typically have at least one root '/' route.

When the framework receives a message from the user it will be routed to this root '/' dialog for processing.

For many bots this single root '/' dialog is all that's needed but just like websites often have multiple routes, bots will often have multiple dialogs.

https://docs.botframework.com/en-us/node/builder/chat/dialogs/#overview

Closure

Single function (one-step "waterfall") – generally a loop

```
// Create a bot
var bot = new builder.UniversalBot(connector);

// Dialog handling
bot.dialog('/', function (session) {
    session.send('Hello world');
});
```

Waterfall example 1

Waterfalls are the most common form of dialogs

```
// What's different about this bot setup? (this is the new style, btw, for v3.5.0)
var bot = new builder.UniversalBot(connector, [
    function (session) {
        // Introducing a builtin prompt
        builder.Prompts.text(session, 'Hi! What is your name?');
    },
    function (session, results) {
        // Step 2 in the waterfall
        session.send('Hello %s!', results.response);
    }
]);
```

Most common form of dialogs. Fundamental skill.

Bots based on Bot Builder implement something we call "Guided Dialog" meaning that the bot is generally driving (or guiding) the conversation with the user.

Calling a built-in prompt like Prompts.text() moves the conversation along because the users response to the prompt is passed to the input of the next waterfall step.

In .NET FormFlows are used.

Waterfall example 2

Many-step waterfall

```
// Setup bot and root waterfall - is this old or new style?
var bot = new builder.UniversalBot(connector, [
    function (session) {
        builder.Prompts.text(session, "Hello... What's your name!");
    },
    function (session, results) {
        session.userData.name = results.response;
        builder.Prompts.number(session, "Hi " + results.response + ", How many years have you been coding?");
    },
    function (session, results) {
        session.userData.coding = results.response;
        builder.Prompts.choice(session, "What language do you code Node using?", ["JavaScript", "CoffeeScript",
            "TypeScript"]);
    },
    function (session, results) {
        session.userData.language = results.response.entity;
        session.send("Got it... " + session.userData.name +
                     " you've been programming for " + session.userData.coding +
                     " years and use " + session.userData.language + ".");
    }
]);
```

Most common form of dialogs. Fundamental skill.

Hello…What's your name
Hi <results> How many years have you been coding?
"What language do you code Node using?"

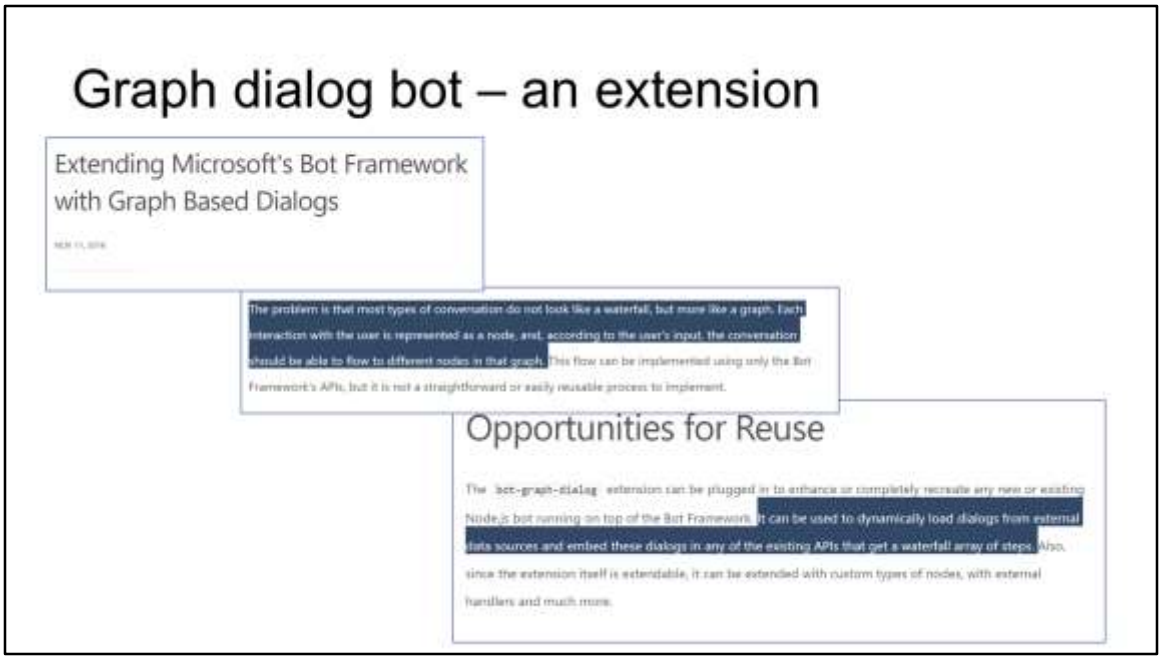
etc.

Bots based on Bot Builder implement something we call "Guided Dialog" meaning that the bot is generally driving (or guiding) the conversation with the user.

With waterfalls you drive the conversation by taking an action that moves the waterfall from one step to the next.

https://docs.botframework.com/en-us/node/builder/chat/dialogs/#waterfall

Graph dialog bot – an extension



https://www.microsoft.com/developerblog/real-life-code/2016/11/11/Bot-Graph-Dialog.html

Conversation model

Sessions or contexts

Sessions

Session contains the stack of active dialogs

- Sending messages
- Dialog stack
- Callbacks

https://docs.botframework.com/en-us/node/builder/chat/session/#navtitle

Sending messages (text, attachments, cards, etc.)
Dialog stack (starting, ending, replacing, etc.)
Callbacks (usage)

Sending messages

A text message is easy. Just `session.send()` for example.

What's more fun is sending attachments.

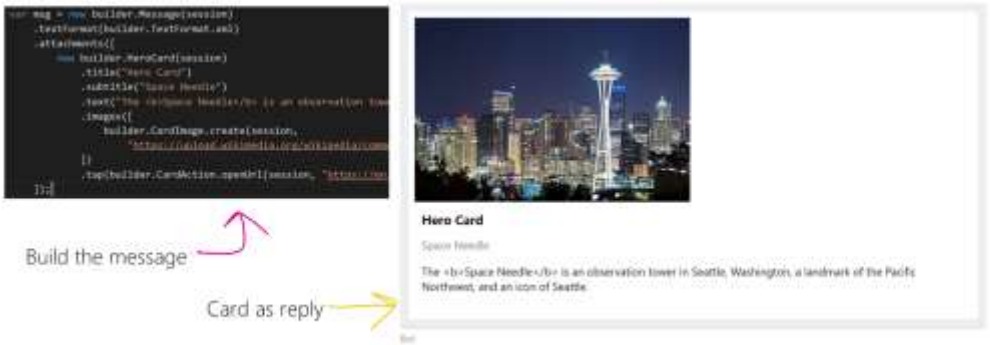
What type of attachment is being sent here?

```
// Send an image attachment to the user
bot.dialog('/picture', [
    function (session) {
        session.send("You can easily send pictures to a user...");
        var msg = new builder.Message(session)
            .attachments([{
                contentType: "image/jpeg",
                contentUrl: "http://www.theoldrobots.com/images62/Bender-18.JPG"
            }]);
        session.endDialog(msg);
    }
]);
```

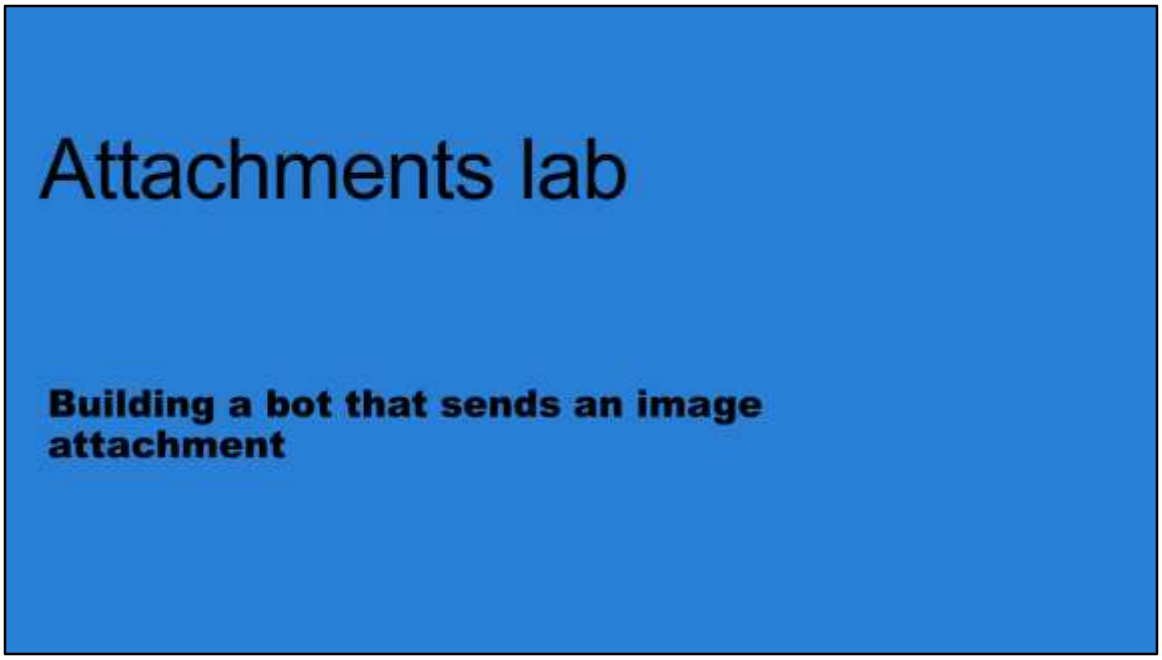
Attachments types allowed: image, video, and file

Sending messages cont'd

Cards – hero and thumbnail cards which can have text, images or buttons even



Follow the instructions for this sample bot: https://github.com/Azure/bot-education/tree/master/Student-Resources/BOTs/Node/bot-playground

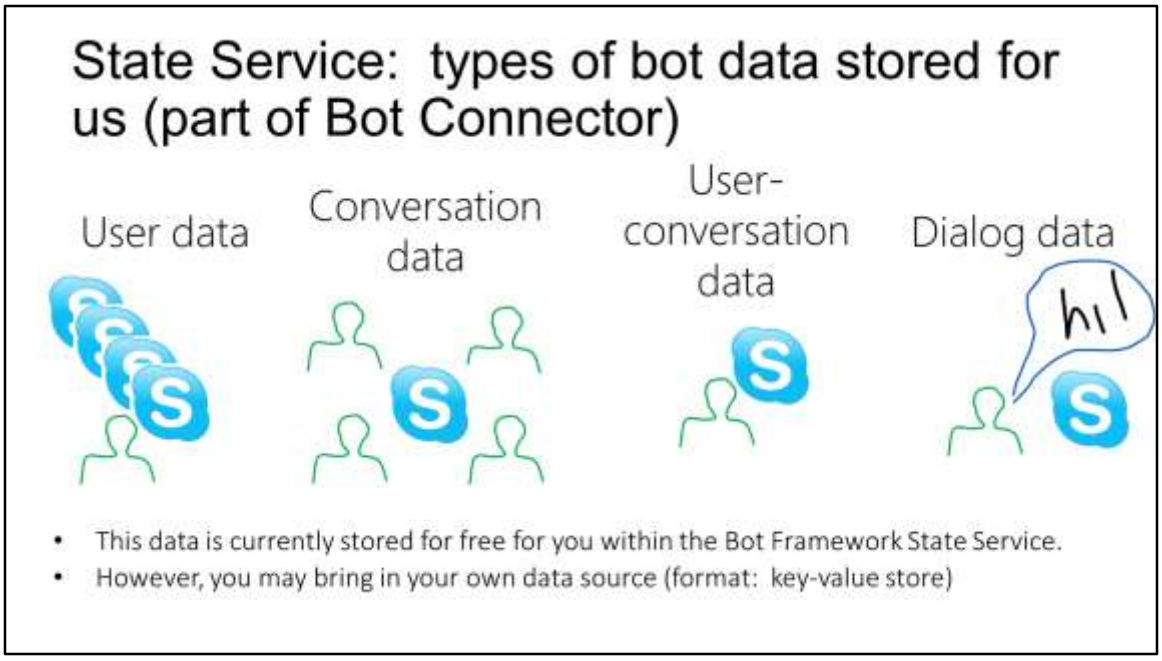


Add an attachment similar to the code in https://docs.botframework.com/en-us/node/builder/chat/session/#attachments to the Hello World or a previous dev bot so that an image is presented after the first message or prompt.

State storage

Managed State API and bags

https://docs.botframework.com/en-us/node/builder/guides/core-concepts/#adding-dialogs-and-memory



User data – globally available for user across all conversations

conversation data – stores globally for a single conversation (many users could be involved)

User-conversation data – stores globally conversation data for a user (But private to just that user)

Dialog data as well – persists for a single dialog (helpful for temp data in a waterfall set of steps)

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.

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.

User input

Prompts
Recognizers

https://docs.botframework.com/en-us/node/builder/chat/prompts/#navtitle

Built-in prompts in both SDKs

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.

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

Additional elements

Localization
Calling
Extensions

https://docs.botframework.com/en-us/node/builder/chat/prompts/#navtitle

Developer tip

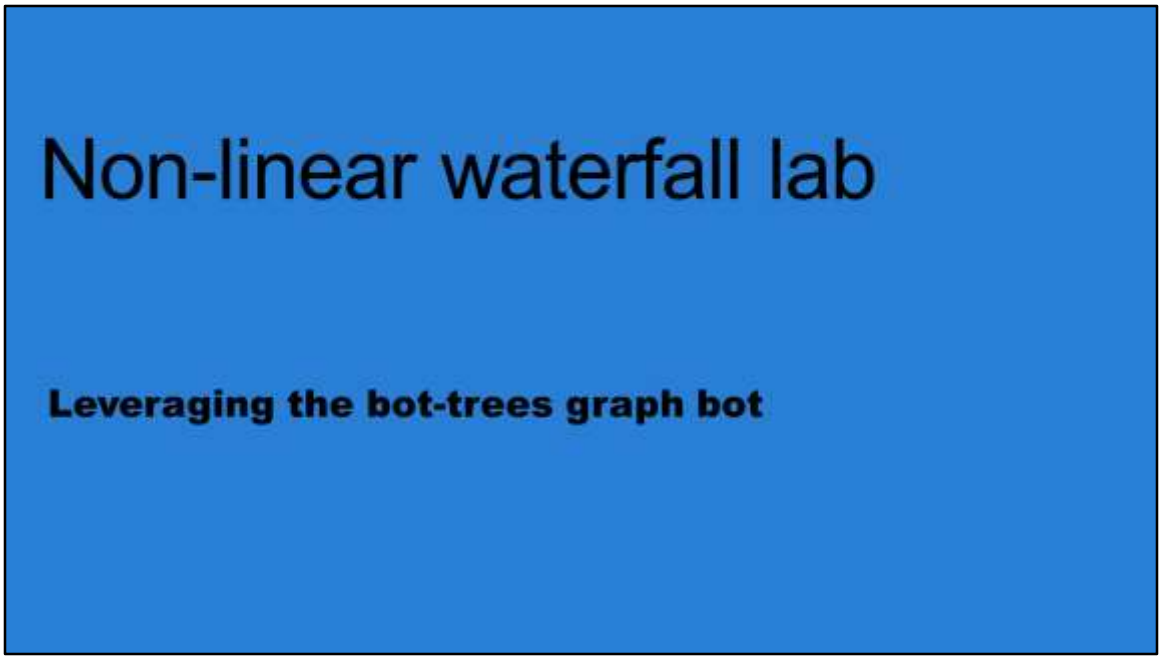
Maintain code in staging spaces:

- Local – Bot Framework connects to my local machine

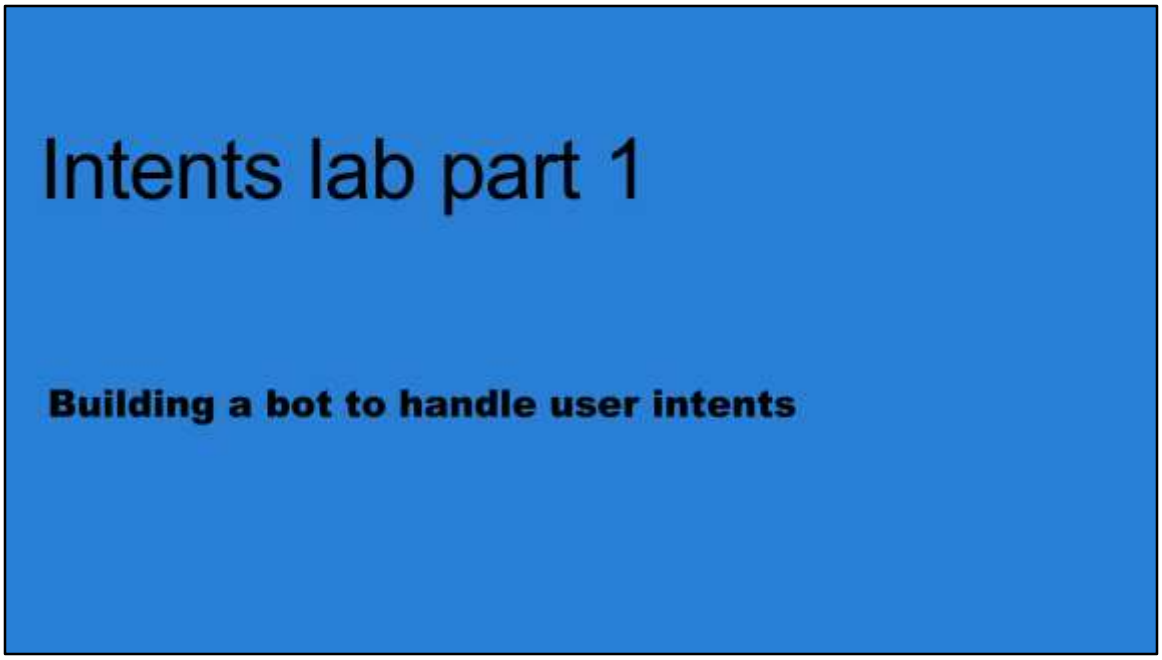
- Dev – push publicly and BF connects in cloud

- 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)



https://github.com/CatalystCode/bot-trees (follow instructions to install extension – this will be part of the Bot Framework in the future)
Set it up for fun as an Azure Bot Service (use the echobot for instance as a template and then host the code for continuous deployment on GitHub and pulled in to Azure) – basically replace the “template” code given by Azure Bot Service with custom code while retaining the structure



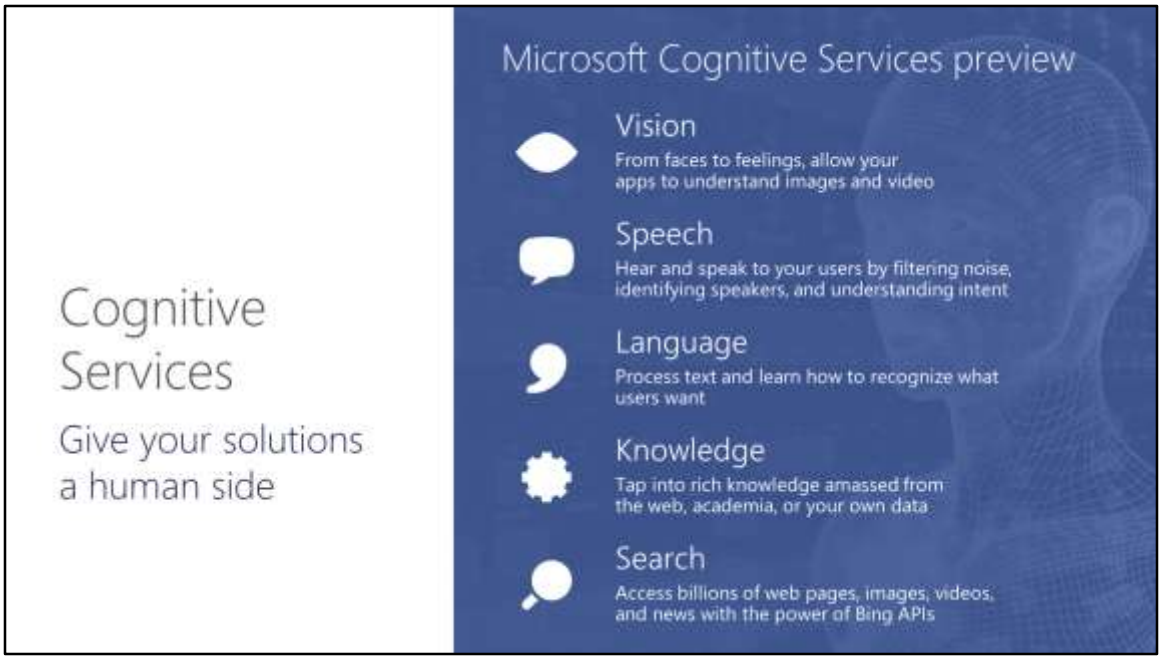
Update this bot:
https://github.com/Microsoft/BotBuilder/blob/master/Node/examples/feature-customRecognizer/app.js for our purposes of testing and to do the following:

- Recognize a user asking for the hours and then the place's name and the date (you'll create a waterfall here)
- Once you've done that, try saving the place's name to userData bag and then use that variable in the next step of the waterfall

See this for a nice example of root dialog waterfall upon which you can base waterfalls:
https://github.com/Microsoft/BotBuilder/blob/master/Node/examples/basics-waterfall/app.js

Advanced:
- Can you deal with the date using Date datetime formats? See https://docs.botframework.com/en-us/node/builder/chat/IntentDialog/#resolving-dates--times

Adding intelligence to your bot



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

So, what are Cognitive Services? Cognitive Services are a collection of artificial intelligence APIs, and we believe in *democratizing* artificial intelligence. So what that means is, regardless of your skill level -- whether you're a high school student running your first program or working in industry or in a giant enterprise -- that you should be able to use our APIs incredibly quickly in a matter of minutes.

And regardless of your platform -- whether you're on Android or IOS or Windows, or making a website -- all of our APIs are rest APIs, which means you can call them as long as you have an Internet connection. And so that's pretty huge because what we're doing is making it so that everyone can build these smarter, more context-aware applications.

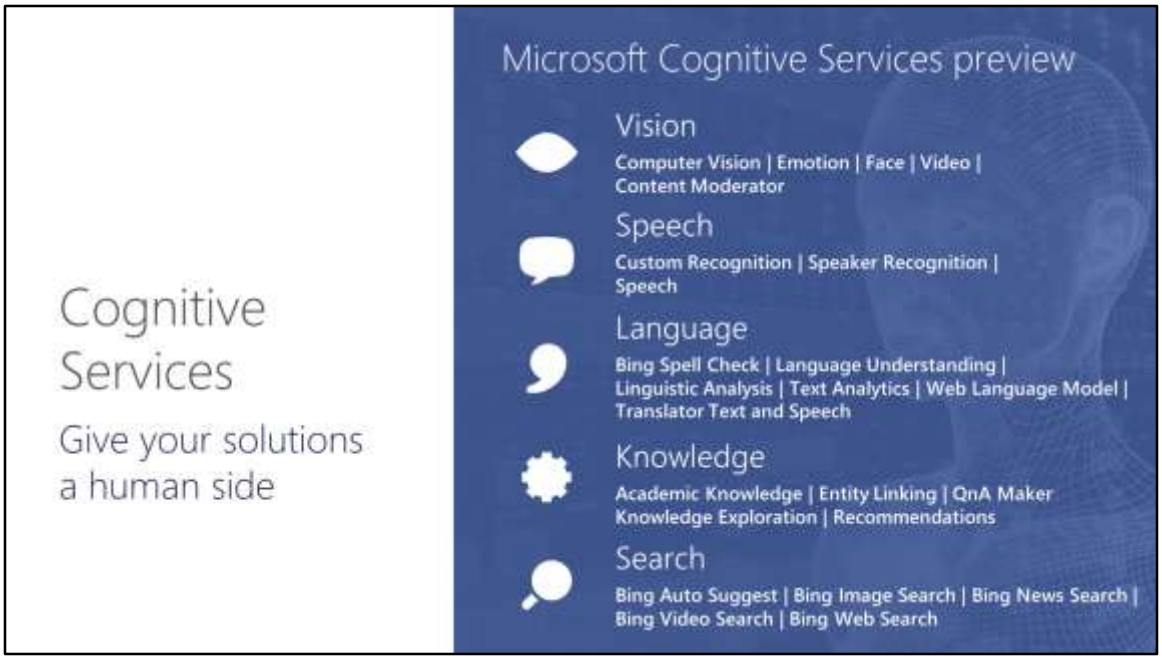
The technology used in our APIs is the same technology that powers our products today. And so, when you think of things like the Bing search APIs, it's the same technology from Bing.

Today I'm going to talk with you about the entire collection spanning vision, speech, language, knowledge, and search.

The other things that I want to point out is that you can get started for free with all of the APIs, but we do have pricing available for a number of them, which are in public preview on Azure.

The other piece is the developer resources. So, all of our documentation is on the website and actually in GitHub as well, so we do welcome community submissions. We have a set of SDKs that are also available on GitHub where we welcome pull requests and post everything on there. The SDKs vary from API to API, but they are all included in this one repository for people to see.

And then we have three different communities that we support. We have our MSDN forums, our Stack Overflow, and we have User Voice that we use for feedback requests.



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 25 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.

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.

Leveraging the API collection

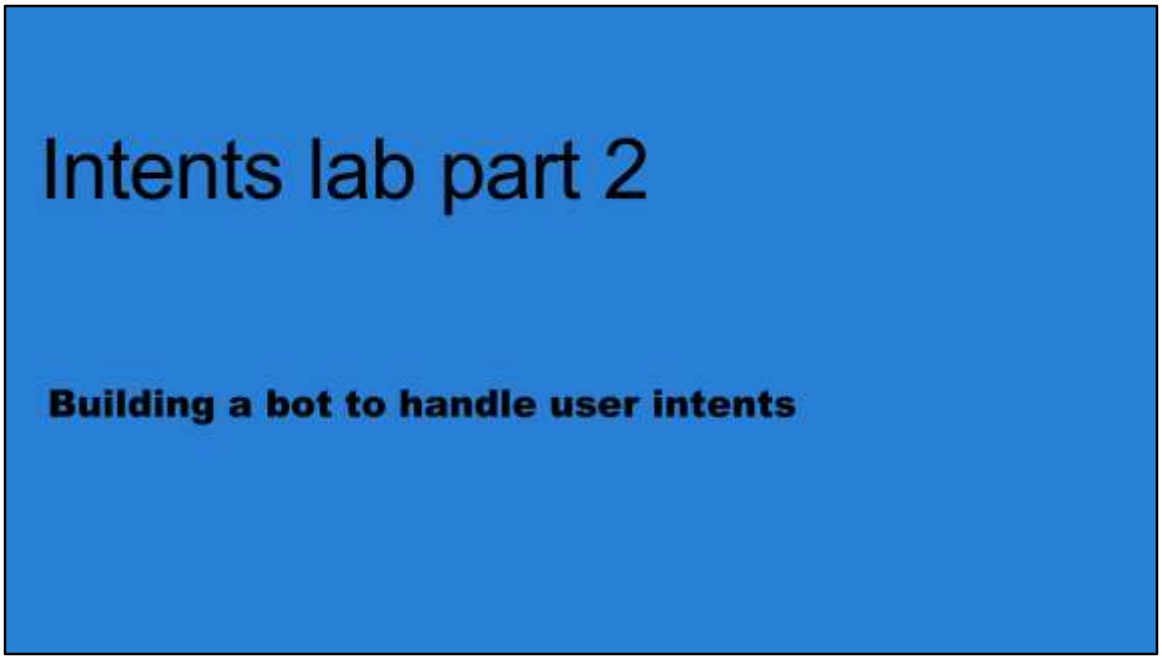
Any of the Cognitive Services APIs can be reached with a simple REST API call

```
var request = require("request");

var readImageText = function _readImageText(url, callback) {

    var options = {
        method: 'POST',
        url: config.CONFIGURATIONS.COMPUTER_VISION_SERVICE.API_URL + "ocr/",
        headers: {
            'ocp-apim-subscription-key': config.CONFIGURATIONS.COMPUTER_VISION_SERVICE.API_KEY,
            'content-type': 'application/json'
        },
        body: {url: url, language: "en"},
        json: true
    };

    request(options, callback);

};
```

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.



Go through the LUIS model setup and then modify your “Intents lab part 1” code to work with LUIS.

To set up the model, follow the LUIS, Intents, Entities, and Model Training and Create Your Model sections beginning at https://docs.botframework.com/en-us/node/builder/guides/understanding-natural-language/#luis

Use the Cortana pre-built model from your subscription.

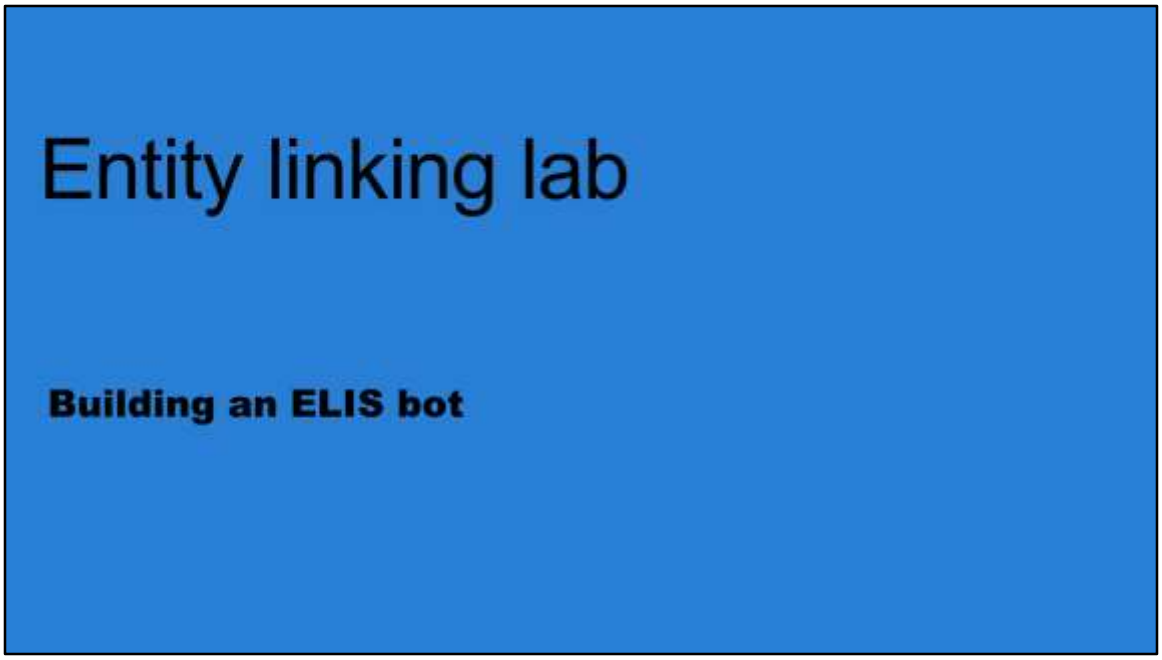
But try with the new style as was begun in “part 1”

You can base your bot loosely on this bot: https://github.com/Microsoft/BotBuilder/blob/master/Node/examples/basics-naturalLanguage/app.js

Note, <model>. Is of the format: https://api.projectoxford.ai/luis/v2.0/apps/[model id goes here]?subscription-key=[key goes here]

Rate this tutorial!

https://aka.ms/botedu-survey



Use this bot as the base for interfacing with cognitive services: https://github.com/michhar/bot-education-samples/tree/master/Node/bot-cognitive-sample

Resources

- Github issues
- Gitter
- Stackoverflow with botframework tag (#botframework)

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

Questions?

Also, try the course gitter chatroom at aka.ms/botedu-discuss