Getting Started

What is Bot Builder for Node.js and why should I use it?

Bot Builder for Node.js is a powerful framework for constructing bots that can handle both freeform interactions and more guided ones where the possibilities are explicitly shown to the user. It is easy to use and models frameworks like Express & Restify to provide developers with a familiar way to write Bots.

High Level Features:

- Powerful dialog system with dialogs that are isolated and composable.
- Built-in prompts for simple things like Yes/No, strings, numbers, enumerations.
- Built-in dialogs that utilize powerful AI frameworks like LUIS.
- Bots are stateless which helps them scale.
- Bots can run on almost any bot platform like the Microsoft Bot Framework, Skype, and Slack.

Build a bot

Create a folder for your bot, cd into it, and run npm init.

```
npm init
```

Get the Bot Builder and Restify modules using npm.

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

Make a file named app.js and say hello in a few lines of code.

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

//=========================================================
// Bot Setup
//=========================================================

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

// Create chat bot
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());

//=========================================================
// Bots Dialogs
//=========================================================

bot.dialog('/', function (session) {
    session.send("Hello World");
});
```

Test your bot

Use the Bot Framework Emulator to test your bot on localhost.

Install the emulator from here and then start your bot in a console window.

```
node app.js
```

Start the emulator and say “hello” to your bot.

Publish your bot

Deploy your bot to the cloud and then register it with the Microsoft Bot Framework. If you’re deploying your bot to Microsoft Azure you can use this great guide for Publishing a Node.js app to Azure using Continuous Integration.

NOTE: When you register your bot with the Bot Framework you'll want to update the appId & appSecret for both your bot and the emulator with the values assigned to you by the portal.

Dive deeper

Learn how to build great bots.

- Core Concepts Guide
- Chat SDK Reference
- Calling SDK Reference
- Bot Builder on GitHub

What's new or changed in v3.3

v3.3

Version 3.3 includes general bug fixes & improvments along with improved localization support.

Localization

Bot Builder now supports a rich file based localization system for building bots that support multiple languages. This new system also provides a way for English only bots to re-skin the SDK's built-in messages so should prove generally useful to all bot developers.

v3.2

Version 3.2 of the Node SDK includes bug fixes as well as a few new convenience methods like Session.sendTyping() & IntentDialog.matchesAny() and also includes support for g-zip `userData` & `conversationData` stored on the Bot Frameworks Bot State service.

v3.1

Version 3.1 of the Node SDK fixes a few bugs, makes a general improvement to the way your waterfalls interact with the built-in prompts, and adds a major new concept called Actions.

Actions

Bot Builder includes a DialogAction which is a small set of *static* functions that can be used to simplify implementing more coming tasks like sending a message, starting a dialog, or creating a new prompt with a built-in validation routine. In version 3.1 we're extending Bot Builders action

concept to include a new set of Named Dialog Actions. These new actions let you define code that should be run should some user initiated action occur like saying “help” or clicking a button on a Persistent Menu in Facebook.

Up to this point, bots based on Bot Builder were limited to conducting a single guided conversation. If the bot shows a card with a `postBack` or `imBack` button the user needs to click the button immediately because once it scrolls up the feed the bot is no longer listening for the button to be clicked and therefore can’t take any action in response to the click. There was also no clean way for a bot to listen for commands from the user like “help” or “cancel” as again the bot can generally only listen for responses to the last questioned it asked the user.

With the addition of Actions all of that changes. Bots can now define global actions which can be triggered regardless of where the user is within the conversation. A global “help” action can easily be added that can be triggered from anywhere to provide the user with help. A global “goodbye” action can be used to let the end the conversation with the bot from anywhere. You can also define actions at the dialog level which will only be evaluated if the dialog is on the stack. So tagging a complex order taking dialog as cancelable is now a simple one-line operation. All of these actions can either be bound to buttons so that they’ll be triggered anytime the

button is clicked (regardless of where it is in the feed) or you can associate a regular expression with the action which will cause it to be triggered anytime the user says a given utterance.

Prompt Improvements

To go along with the new Actions feature, the built-in prompts have received some improvements. In previous versions of Bot Builder you needed to be pretty defensive when coding your waterfall steps as there were a number of things that could cause a call to one of the built-in prompts to return a `result.response` that was null. The user could have asked to cancel the prompt or they could have simply failed to enter a valid response within the appropriate number of retries. Both issues can lead to very buggy waterfall code if you don't add a whole series of defensive checks to every single step of your waterfall.

To try and simplify building less buggy waterfalls you can now generally code them without having to first check to ensure that `result.response` has a value before using it. With two exceptions, if you now call a built-in prompt your waterfall will only advance to the next step once a user successfully enters a valid response. That means you can now just assume you got a value from the user and remove all of those nasty extra checks. The changes the made this possible is the prompts no longer listen for the user to say "cancel", there's an action for that. And the prompts now by

default re-prompt the user to enter a valid prompt before advancing. The two exceptions to that rule are when you manually advance the waterfall by calling `next()` from a step or you explicitly set the `maxRetries` setting when you call the prompt. In both cases you're in control of when this happens and therefore you can code to protext against a null value being returned for `result.response`.

v3.0

There are numerous changes to Bot Builder for Node.js in the 3.0 release. Before we get into the changes let's answer "Why version 3.0 and not version 2.0?". For the past few months the Bot Framework team has been working with Skype to create one unified bot platform for all of Microsoft. This resulted in a new unified message schema and attachment format that we jointly called the v3 schema. All of the new API's from the Bot Framework are annotated with v3 so to avoid confusion on the SDK side we decided to adopt v3 for the SDK version number as well.

To go along with the new message schema being introduced we also took the opportunity to overhaul a number of elements of the SDK. We've learned a lot over the last couple of months (this bot building stuff is new to us too) and so the SDK was definitely in need of a little restructuring to better position it moving forward.

The big question that most of you probably have is “Do I need to re-write my bot?” For most bots the answers “no”. Your old bot should keep working although you may see a few warnings about using deprecated classes and attachment schemas logged to the console. But you’ll want to update your bot as soon as you can because all of the new stuff is richer and better. It should be relatively straight forward to switch to the new stuff so it should probably only take a few minutes at most. In a few cases, like if your bot is sending “proactive messages” or you’re using the `SlackBot` class you will need to rework your bot because we’ve significantly changed the way proactive messaging works (it’s way simpler now) and in the case of the SlackBot it’s unfortunately no longer supported Options for that will be outlined below.

New Schema

The message schema in v3 has changed quite a bit but this should have little effect on most existing bots as the fields that have changed aren’t ones that you’d normally touch. The exceptions to that are the old `channelData` field is now called sourceEvent and all of the `from`, `to`, and `conversationId` fields have been moved onto a new IAddress object that hangs off session.message.address. This new address object dramatically simplifies sending proactive messages from your bot as now all you have to do is save the address object for some user you want to notify at a future point in time and then when you’re ready to contact them you

read that in, compose a new message with that object for the address, and then call either bot.send() or bot.beginDialog().

The bigger schema change is around attachments. There's a whole new card schema that lets you send rich hero cards, thumbnail cards, and receipt cards in a cross channel way. The SDK now includes a whole new set of card builder classes to make it trivial to build carousels and lists of cards that work across multiple channels. To keep from breaking a large number of bots the SDK still supports the old attachment format and will automatically convert your old style attachments to the new format but it will load your console up with warnings every time it does so you'll want to convert to the new attachment schema as soon as you can.

UniversalBot

The 1.x version of the SDK had a number of bot classes that would connect your bots dialogs to a range of sources. The issue with that model is that it you can really only easily connect your bots dialogs to one source at a time and there's a lot of code duplication across these various bot classes. The new SDK has a single UniversalBot class that you connect your dialogs to. This new class has a much lighter weight connector model for connecting your bot to different sources. The SDK comes out of the box with a ChatConnector & ConsoleConnector class and you could connect your bot to both simultaneously if you wanted to.

The old BotConnectorBot, SkypeBot, and TextBot classes are deprecated but largely still functional. You'll receive a warning when you use them and an exception will be thrown if you try to use a feature that's no longer supported. Converting to using the new UniversalBot class should be fairly straightforward. For the BotConnectorBot & SkypeBot classes switch to using the UniversalBot with the ChatConnector.

```
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());
```

For the TextBot class switch to using the UniversalBot with a ConsoleConnector:

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

The old `SlackBot` class for talking natively to Slack is unfortunately no longer supported and has been removed from the SDK all together. The reasoning for this is that Bot Framework already has a slack channel so over the long run you'll get better support for cross channel messaging by going through the ChatConnector when talking to Slack. Someone in the

community way elect to build a native connector for Slack and we'll certainly point everyone to that if they do.

IntentDialog

A lot of users have asked for more flexibility with regards to the LuisDialog class. They'd like the ability to chain multiple models together or the ability to add CommandDialog regular expression based matching over the top of LUIS's built-in Crotona Model. In the new SDK we've completely reworked the IntentDialog class that the LuisDialog class derived from this. This new IntentDialog class is now the only class you need to perform intent recognition. It lets you mix together regular expression based intent matching with a new system of pluggable cloud based intent recognizers. Out of the box the SDK includes a new LuisRecognizer plugin but the community could easily add recognizers for other services. You can configure the IntentDialog to use multiple recognizers and then there are options to control whether the recognizers are evaluated in parallel or series. Extending LUIS's built-in models is now a breeze.

The old CommandDialog & LuisDialog classes have been deprecated but are still completely functional. You'll just get a warning logged to the console on startup. Migrating to use the new IntentDialog class is super straight forward:

```
var recognizer = new builder.LuisRecognizer('<your models url>');
var intents = new builder.IntentDialog({ recognizers: [recognizer] });
bot.dialog('/', intents);
```

Message and Card Builders

The 1.x version of the SDK included a Message builder class that was useful for composing messages when you wanted your bot to return attachments. In light of recent developments in the bot space we envision more and more bots will want to take advantage of all the various buttons, cards, and keyboard features being added by messaging clients. We want to make it as easy as possible to build rich replies from your bot so we've dramatically revamped the existing Message builder class and added a whole new suite of Card builder classes. The existing methods on the Message class still work but are mostly deprecated. The old attachment format still works but is also deprecated in favor of the new card formats. Pretty much everything in the SDK that can send a reply to the user has been updated to take either strings or a Message object so even all of the built-in prompts let you prompt the user using cards that automatically adapt to the channel they're being rendered to.

Middleware

Bot Builders existing middleware system has also received an overhaul. Middleware can now install multiple hooks to capture incoming & outgoing

messages as well as a hook that lets you manipulate Bot Builders dialog system. We're working with other SDK developers in the Node community to create standard interfaces and schema that let bot related middleware work across multiple SDK's. So very soon middleware written for Bot Builder will work in an SDK like Botkit and vice versa.

To go along with this the SDK now includes a couple of pieces of built-in middleware. There's a new dialogVersion() middleware that lets you version your dialogs and automatically reset active conversation with your bot when you deploy a major change to your bot. The new firstRun middleware lets you create a really clean firstrun experience for your bot. You can run new users through a set of first run dialogs to collect things like profile information and accept your bots terms of use. Then when your terms of use change you can automatically run them back through just the terms of use part to have them re-accept the new terms. These are the first two pieces of middleware we're delivering but a lot more will be coming soon.

Libraries

This is still a work in progress but there's a whole new Library system design to let you package up parts of your bot into Node modules that can be easily reused within other bots. We're working up examples that show this in action and provide a recipe for how to build your own libraries. On top of that we have quite a few add-on libraries we'd like to ship ourselves.

Error Handling

The new SDK contains several general improvements around error handling. Situations that would get users stuck in your bot for unknown reasons should now be resolved and there's a new customizable error message that gets sent to the user anytime we detect an error like an exception being thrown in your bot.

Calling SDK

Last but certainly not least we're shipping a whole new Calling SDK that lets you build bots exclusively for skype that you can call and have conversations with using voice. These bots borrow a lot of concepts from their chat counterparts so if you're familiar with developing chat bots, building calling bots will feel like a natural extension of those skills. You can even directly share code between your calling and chat bots in some cases.

All

< Documentation Home

Bot Builder for Node.js

Getting Started

What's new or changed in v3.3

Guides

Core Concepts

Understanding Natural Language

Debug Locally with VSCode

Deploying to Azure

Examples

Chat Bots

UniversalBot

Dialogs

Session

Prompts

IntentDialog

Localization

Calling Bots

UniversalCallBot

Core Concepts

Overview

Bot Builder is a framework for building conversational applications (“Bots”) using Node.js. From simple command based bots to rich natural language bots the framework provides all of the features needed to manage the conversational aspects of a bot. You can easily connect bots built using the framework to your users wherever they converse, from SMS to Skype to Slack and more...

Installation

To get started either install the Bot Builder module via NPM:

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

Or clone our GitHub repository using Git. This may be preferable over NPM as it will provide you with numerous example code fragments and bots:

```
git clone https://github.com/Microsoft/BotBuilder.git
cd BotBuilder/Node
npm install
```

Hello World

Once the Bot Builder module is installed we can get things started by building our first "Hello World" bot called HelloBot. The first decision we need to make is what kind of bot do we want to build? Bot Builder lets you build bots for a variety of platforms but for our HelloBot we're just going to interact with it though the command line so we're going to create a UniversalBot bound to an instance of a ConsoleConnector:

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

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

The UniversalBot class implements all of logic to manage the bots' conversations with users. You can bind the UniversalBot to a variety of channels using connectors. For this guide we'll just chat with the bot from a console window so we'll use the ConsoleConnector class. In the future when you're ready to deploy your bot to real channels you'll want to swap out the ConsoleConnector for a ChatConnector configured with your bots App ID & Password from the Bot Framework portal.

Now that we have our bot & connector setup, we need to add a dialog to our newly created bot object. Bot Builder breaks conversational applications up into components called dialogs. If you think about building a conversational application in the way you'd think about building a web application, each dialog can be thought of as route within the conversational application. As users send messages to your bot the framework tracks which dialog is currently active and will automatically route the incoming message to the active dialog. For our HelloBot we'll just add single root '/' dialog that responds to any message with "Hello World".

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

var connector = new builder.ConsoleConnector().listen();
var bot = new builder.UniversalBot(connector);
bot.dialog('/', function (session) {
    session.send('Hello World');
});
```

We can now run our bot and interact with it from the command line. So run the bot and type 'hello':

```
node app.js
hello
Hello World
```

Collecting Input

It's likely that you're going to want your bot to be a little smarter than HelloBot currently is so let's give HelloBot the ability to ask the user their name and then provide them with a personalized greeting. To do that we're going to introduce a new concept called a waterfall which will prompt the user for some information and then wait for their response:

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

var connector = new builder.ConsoleConnector().listen();
var bot = new builder.UniversalBot(connector);
bot.dialog('/', [
    function (session) {
        builder.Prompts.text(session, 'Hi! What is your name?');
    },
    function (session, results) {
        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.

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. If we run our updated HelloBot we know see that our bot asks us for our name and then gives us a personalized greeting.

```
node app.js
hello
Hi! What is your name?
John
Hello John!
hi there
Hi! What is your name?
```

The problem now is that if you say hello multiple times the bot doesn't remember our name. Let's fix that.

Adding Dialogs and Memory

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. For HelloBot we're going to add a new `/profile` dialog that guides the user through filling out their profile information. This information needs to be stored somewhere so we can either return it to the caller as the output from our dialog using `session.endDialog({ response: { name: 'John' } })` or we can store it globally using the SDK's built-in storage system. In our case we want to remember this information globally for a user so we're going to store it off session.userData. This also gives us a convenient way to trigger that we should ask the user to fill out their profile.

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

var connector = new builder.ConsoleConnector().listen();
var bot = new builder.UniversalBot(connector);
bot.dialog('/', [
    function (session, args, next) {
        if (!session.userData.name) {
            session.beginDialog('/profile');
        } else {
            next();
        }
    },
    function (session, results) {
        session.send('Hello %s!', session.userData.name);
    }
]);

bot.dialog('/profile', [
    function (session) {
        builder.Prompts.text(session, 'Hi! What is your name?');
```

```
    },
    function (session, results) {
        session.userData.name = results.response;
        session.endDialog();
    }
]);
```

Now when we run HelloBot it will prompt us to enter our name once and it will then remember our name the next time we chat with the bot.

```
node app.js
hello
Hi! What is your name?
John
Hello John!
hi there
Hello John!
```

The SDK includes several ways of persisting data relative to a user or conversation:

- **userData** stores information globally for the user across all conversations.
- **conversationData** stores information globally for a single conversation. This data is visible to everyone within the conversation so care should be used to what's stored there. It's disabled by default and needs to be enabled using the bots persistConversationData setting.
- **privateConversationData** stores information globally for a single conversation but its private data for the current user. This data spans all dialogs so it's useful for storing temporary state that you want cleaned up when the conversation ends.

- **dialogData** persists information for a single dialog instance. This is essential for storing temporary information in between the steps of a waterfall.

Bots built using Bot Builder are designed to be stateless so that they can easily be scaled to run across multiple compute nodes. Because of that you should generally avoid the temptation to save state using a global variable or function closure. Doing so will create issues when you want to scale out your bot. Instead leverage the data bags above to persist temporary and permanent state.

Now that HelloBot can remember things let's try to make it better understand the user.

Determining Intent

One of the keys to building a great bot is effectively determining the users intent when they ask your bot to do something. Bot Builder includes a powerful IntentDialog class designed to assist with that task. The IntentDialog class lets you determine the users intent using a combination of two techniques. You can pass a regular expression to IntentDialog.matches() the users message text will be compared against that RegEx. If it matches then the handler associated with that expression will be triggered.

Regular expressions are nice but for even more powerful intent recognition you can leverage machine learning via LUIS by plugging a LuisRecognizer into your IntentDialog (explore the examples to see this in action.) The IntentDialog also supports a combination of RegEx's and recognizer plugins and will use scoring heuristics to identify the most likely handler to invoke. If no intents are predicted or the score is below a threshold the dialogs onDefault() handler will be invoked.

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

var connector = new builder.ConsoleConnector().listen();
var bot = new builder.UniversalBot(connector);
var intents = new builder.IntentDialog();
bot.dialog('/', intents);

intents.matches(/^change name/i, [
    function (session) {
        session.beginDialog('/profile');
    },
    function (session, results) {
        session.send('Ok... Changed your name to %s', session.userData.r
    }
]);

intents.onDefault([
    function (session, args, next) {
        if (!session.userData.name) {
            session.beginDialog('/profile');
        } else {
            next();
        }
```

```
    },
    function (session, results) {
        session.send('Hello %s!', session.userData.name);
    }
]);

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

Our HelloBot has been updated to use an IntentDialog that will look for the user to say “change name”. As you can see the handler for this intent is just another waterfall which means you can do any sequence of prompts/actions in response to an intent being triggered. We moved our old dialogs waterfall to the intent dialogs onDefault() handler but it otherwise remains unchanged. Now when we run HelloBot we get a flow like this:

```
node app.js
hello
Hi! What is your name?
John
Hello John!
change name
Hi! What is your name?
```

```
Steve
Ok... Changed your name to Steve
Hi
Hello Steve!
```

Publishing to the Bot Framework Service

Now that we have a fairly functional HelloBot (it does an excellent job of greeting users) we should publish it the Bot Framework Service so that we can talk to it across various communication channels. Code wise we'll need to update our bot to use a properly configured ChatConnector:

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

//=========================================================
// Bot Setup
//=========================================================

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

// Create chat bot
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());

//=========================================================
// Bots Dialogs
//=========================================================

var intents = new builder.IntentDialog();
bot.dialog('/', intents);

intents.matches(/^change name/i, [
    function (session) {
        session.beginDialog('/profile');
    },
    function (session, results) {
        session.send('Ok... Changed your name to %s', session.userData.r
    }
]);

intents.onDefault([
    function (session, args, next) {
        if (!session.userData.name) {
            session.beginDialog('/profile');
        } else {
            next();
        }
    },
    function (session, results) {
        session.send('Hello %s!', session.userData.name);
    }
]);

bot.dialog('/profile', [
    function (session) {
        builder.Prompts.text(session, 'Hi! What is your name?');
    },
```

```
    function (session, results) {
        session.userData.name = results.response;
        session.endDialog();
    }
]);
```

Our updated bot now requires the use of Restify so we'll need to install that. From a console window in our bots directory type:

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

While our bot has a bit of new setup code, all of our bots dialogs stay the same. To complete the conversion, you'll need to register your bot with the developer portal for the Bot Framework and then configure the ChatConnector with your bots App ID & Password. You pass these to the bot via environment variables when running locally or via your hosting sites web config when deployed to the cloud.

You can use the Bot Framework Emulator to locally test your changes and verify you have everything properly configured prior to deploying your bot. Make sure you set the App ID & Password within the emulator to match your bots configured App ID & Password.

Debugging locally using ngrok

If you're running on a mac and can't use the emulator, or you just want to debug an issue you're seeing when deployed, you can easily configure your bot to run locally using ngrok. First install ngrok and then from a console window type:

```
ngrok http 3978
```

This will configure an ngrok forwarding link that forwards requests to your bot running on port 3978. You'll then need to set the forwarding link to be the registered endpoint for your bot within the Bot Framework developer portal. The endpoint should look something like `https://0d6c4024.ngrok.io/api/messages` once configured. Just don't forget to include the `/api/messages` at the end of the link.

You're now ready to start your bot in debug mode. If you're using VSCode you'll need to configure your launch `env` with your bots App ID & Password:

```
{
    "version": "0.2.0",
    "configurations": [
        {
            "name": "Launch",
            "type": "node",
            "request": "launch",
            "program": "${workspaceRoot}/app.js",
```

```
            "stopOnEntry": false,
            "args": [],
            "cwd": "${workspaceRoot}",
            "preLaunchTask": null,
            "runtimeExecutable": null,
            "runtimeArgs": [
                "--nolazy"
            ],
            "env": {
                "NODE_ENV": "development",
                "MICROSOFT_APP_ID": "Your bots App ID",
                "MICROSOFT_APP_PASSWORD": "Your Bots Password"
            },
            "externalConsole": false,
            "sourceMaps": false,
            "outDir": null
        },
```

Configure the `.vscode/launch.json` file, press run, and you're ready to debug!

All

< Documentation Home

Bot Builder for Node.js

Getting Started

What's new or changed in v3.3

Guides

Core Concepts

Understanding Natural Language

Debug Locally with VSCode

Deploying to Azure

Examples

Chat Bots

UniversalBot

Dialogs

Session

Prompts

IntentDialog

Localization

Calling Bots

UniversalCallBot

Understanding Natural Language

LUIS

Microsofts Language Understanding Intelligent Service (LUIS) offers a fast and effective way of adding language understanding to applications. With LUIS, you can use pre-existing, world-class, pre-built models from Bing and Cortana whenever they suit your purposes – and when you need specialized models, LUIS guides you through the process of quickly building them.

LUIS draws on technology for interactive machine learning and language understanding from Microsoft Research and Bing, including Microsoft Research's Platform for Interactive Concept Learning (PICL). LUIS is a part of project of Microsoft Project Oxford.

Bot Builder lets you use LUIS to add natural language understanding to your bot via the LuisDialog class. You can add an instance of a LuisDialog that references your published language model and then add intent handlers to take actions in response to users utterances. To see LUIS in action watch the 10 minute tutorial below.

- Microsoft LUIS Tutorial (video)

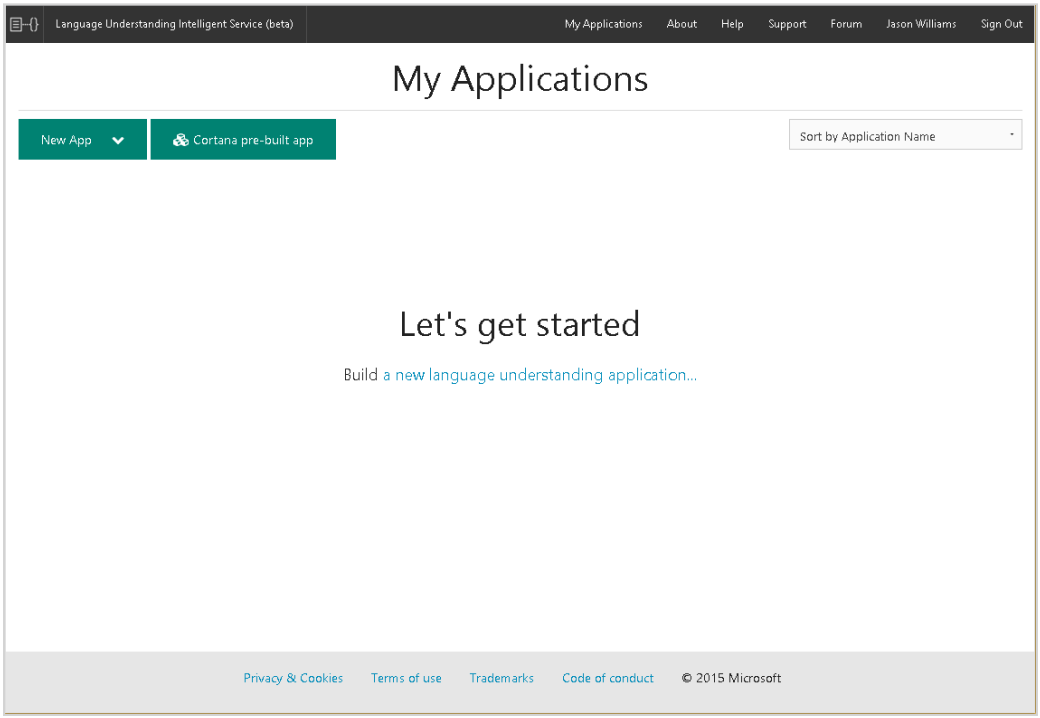
Intents, Entities, and Model Training

One of the key problems in human-computer interactions is the ability of the computer to understand what a person wants, and to find the pieces of information that are relevant to their intent. For example, in a news-browsing app, you might say “Get news about virtual reality companies,” in which case there is the intention to FindNews, and “virtual reality companies” is the topic. LUIS is designed to enable you to very quickly deploy an http endpoint that will take the sentences you send it, and interpret them in terms of the intention they convey, and the key entities like “virtual reality companies” that are present. LUIS lets you custom design the set of intentions and entities that are relevant to the application, and then guides you through the process of building a language understanding system.

Once your application is deployed and traffic starts to flow into the system, LUIS uses active learning to improve itself. In the active learning process, LUIS identifies the interactions that it is relatively unsure of, and asks you to label them according to intent and entities. This has tremendous advantages: LUIS knows what it is unsure of, and asks you to help where you will provide the maximum improvement in system performance. Secondly, by focusing on the important cases, LUIS learns as quickly as possible, and takes the minimum amount of your time.

Create Your Model

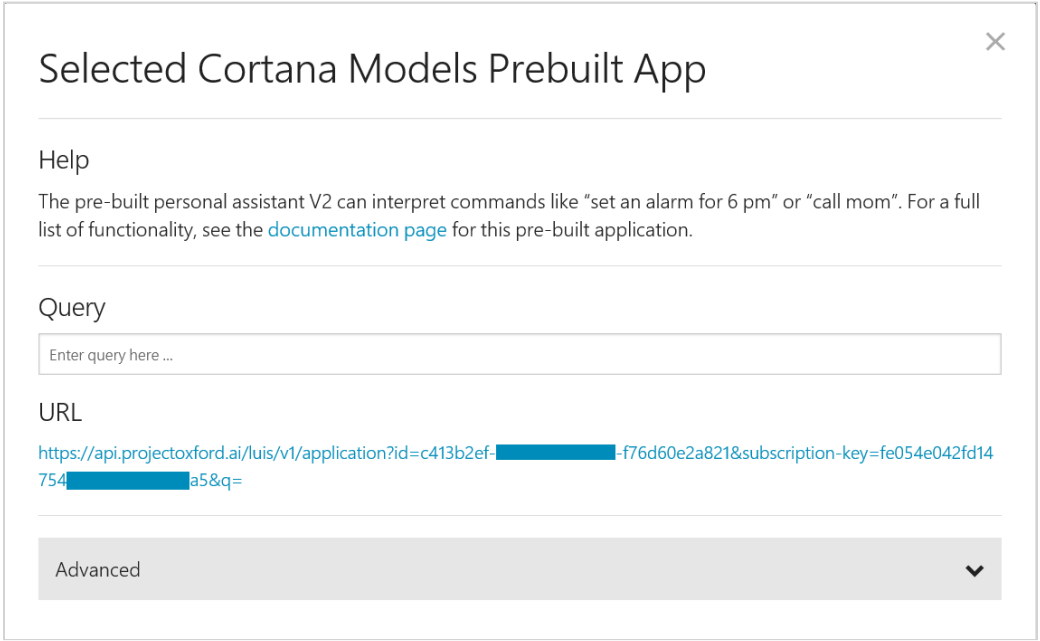
The first step of adding natural language support to your bot is to create your LUIS Model. You do this by logging into LUIS and creating a new LUIS Application for your bot. This application is what you'll use to add the Intents & Entities that LUIS will use to train your bots Model.



In addition to creating a new app you have the option of either importing an existing model (this is what you'll do when working with the Bot Builder examples that use LUIS) or using the prebuilt Cortana app. For the purposes of this tutorial we'll create a bot based on the prebuilt Cortana app. When you select the prebuilt Cortana app for English you'll see a dialog like below.

You'll want to copy the URL listed on the dialog as this is what you'll bind your LuisDialog class to. This URL points to the Model that LUIS published for your bots LUIS app and will be stable for the lifetime of the app. So

once you've trained and published a model for a LUIS app you can update and re-train the model all you want without having to even redeploy your bot. This is very handy in the early stages of building a bot as you'll be re-training your model a lot.



Handle Intents

Once you've deployed a model for your LUIS app we can create a bot that consumes that model. To keep things simple we'll create a UniversalBot that we can interact with from a console window.

Create a folder for your bot, cd into it, and run npm init.

```
npm init
```

Install the Bot Builder module from npm.

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

Create a file named app.js with the code below. You'll need to update the model in the sample code below to use the URL you got from LUIS for your copy of the prebuilt Cortana App.

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

// Create bot and bind to console
var connector = new builder.ConsoleConnector().listen();
var bot = new builder.UniversalBot(connector);

// Create LUIS recognizer that points at our model and add it as the root '/
var model = '<your models url>';
var recognizer = new builder.LuisRecognizer(model);
var dialog = new builder.IntentDialog({ recognizers: [recognizer] });
bot.dialog('/', dialog);

// Add intent handlers
dialog.matches('builtin.intent.alarm.set_alarm', builder.DialogAction.send('
dialog.matches('builtin.intent.alarm.delete_alarm', builder.DialogAction.se
dialog.onDefault(builder.DialogAction.send("I'm sorry I didn't understand.
```

This sample pulls in our Cortana Model and implements two intent handlers, one for setting a new alarm and one for deleting an alarm. Both

handlers for now just use a DialogAction to send a static message when triggered. The Cortana Model can actually trigger a number of intents so we'll also add an onDefault() handler to catch any intents we don't currently support. Running this sample from the command line we get something like this:

```
node app.js
set an alarm in 5 minutes called wakeup
Creating Alarm
snooze the wakeup alarm
I'm sorry I didn't understand. I can only create & delete alarms.
delete the wakeup alarm
Deleting Alarm
```

Process Entities

Now that we have our bot understanding what the users intended action is we can do the work of actually creating and deleting alarms. We'll extend our sample to include logic to handle each intent and a very simple in-memory alarm scheduler.

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

// Create bot and bind to console
var connector = new builder.ConsoleConnector().listen();
var bot = new builder.UniversalBot(connector);

// Create LUIS recognizer that points at our model and add it as the root '/
```

```
var model = '<your models url>';
var recognizer = new builder.LuisRecognizer(model);
var dialog = new builder.IntentDialog({ recognizers: [recognizer] });
bot.dialog('/', dialog);

// Add intent handlers
dialog.matches('builtin.intent.alarm.set_alarm', [
    function (session, args, next) {
        // Resolve and store any entities passed from LUIS.
        var title = builder.EntityRecognizer.findEntity(args.entities, 'buil
        var time = builder.EntityRecognizer.resolveTime(args.entities);
        var alarm = session.dialogData.alarm = {
          title: title ? title.entity : null,
          timestamp: time ? time.getTime() : null
        };

        // Prompt for title
        if (!alarm.title) {
            builder.Prompts.text(session, 'What would you like to call yo
        } else {
            next();
        }
    },
    function (session, results, next) {
        var alarm = session.dialogData.alarm;
        if (results.response) {
            alarm.title = results.response;
        }

        // Prompt for time (title will be blank if the user said cancel)
        if (alarm.title && !alarm.timestamp) {
            builder.Prompts.time(session, 'What time would you like to s
        } else {
            next();
        }
    },
```

```
    function (session, results) {
        var alarm = session.dialogData.alarm;
        if (results.response) {
            var time = builder.EntityRecognizer.resolveTime([results.res
            alarm.timestamp = time ? time.getTime() : null;
        }

        // Set the alarm (if title or timestamp is blank the user said cance
        if (alarm.title && alarm.timestamp) {
            // Save address of who to notify and write to scheduler.
            alarm.address = session.message.address;
            alarms[alarm.title] = alarm;

            // Send confirmation to user
            var date = new Date(alarm.timestamp);
            var isAM = date.getHours() < 12;
            session.send('Creating alarm named "%s" for %d/%d/%d %d
                alarm.title,
                date.getMonth() + 1, date.getDate(), date.getFullYear(
                isAM ? date.getHours() : date.getHours() - 12, date.g
        } else {
            session.send('Ok... no problem.');
        }
    }
]);

dialog.matches('builtin.intent.alarm.delete_alarm', [
    function (session, args, next) {
        // Resolve entities passed from LUIS.
        var title;
        var entity = builder.EntityRecognizer.findEntity(args.entities, 'bu
        if (entity) {
            // Verify its in our set of alarms.
            title = builder.EntityRecognizer.findBestMatch(alarms, entit
        }
```

```
        // Prompt for alarm name
        if (!title) {
            builder.Prompts.choice(session, 'Which alarm would you like
        } else {
            next({ response: title });
        }
    },
    function (session, results) {
        // If response is null the user canceled the task
        if (results.response) {
            delete alarms[results.response.entity];
            session.send("Deleted the '%s' alarm.", results.response.entit
        } else {
            session.send('Ok... no problem.');
        }
    }
]);

dialog.onDefault(builder.DialogAction.send("I'm sorry I didn't understand.

// Very simple alarm scheduler
var alarms = {};
setInterval(function () {
    var now = new Date().getTime();
    for (var key in alarms) {
        var alarm = alarms[key];
        if (now >= alarm.timestamp) {
            var msg = new builder.Message()
                .address(alarm.address)
                .text("Here's your '%s' alarm.", alarm.title);
            bot.send(msg);
            delete alarms[key];
        }
    }
}, 15000);
```

We're using waterfalls for our set_alarm & delete_alarm intent handlers. This is a common pattern that you'll likely use for most of your intent handlers. The way waterfalls work in Bot Builder is the very first step of the waterfall is called when a dialog (or in this case intent handler) is triggered. The step then does some work and continue execution of the waterfall by either calling another dialog (like a built-in prompt) or calling the optional next() function passed in. When a dialog is called in a step, any result returned from the dialog will be passed as input to the results parameter for the next step.

In the case of intent handlers any entities that LUIS recognized will be passed along in the args parameter of that first step of the waterfall. More often than not you need some additional pieces of information before you can fully process the users request. LUIS uses entities to pass that extra data along but you really don't want to require that the user enter every single piece of information up front. So in the set_alarm case we want to support "set alarm in 5 minutes called wakeup", "create an alarm called wakeup", and just "create an alarm". That means we may not always get all of the entities we expect from LUIS and even when we get them we should expect them to be invalid at times. So in all cases we're going to want to be prepared to prompt the user for missing or invalid entities. Bot Builder makes it relatively easy to build that flexibility into your bot by using a combination of waterfalls & built-in prompts.

Looking at the waterfall for the set_alarm intent. We're first going to try and validate & store any entities we received from LUIS. Bot Builder includes an EntityRecognizer class which has useful functions for working with entities. Times, for instance, can come in fairly decomposed and often spanning multiple entities. You can use the EntityRecognizer.resolveTime() function to return you an actual JavaScript Date object based upon the passed in time entities if it's able to calculate one.

Once we've validated and stored our entities we're going to then decide if we need to prompt the user for the name of the alarm. If we got the title passed to us from LUIS we can skip the prompt and proceed to the next step in the waterfall using the next() function. If not we can ask the user for the title using the Prompts.text() built-in prompt. If the user enters a title it will be passed to the next step via the results.response field so in the next step of the waterfall we can store the users response, and then figure out do we have the next missing piece of data and either skip to the next step or prompt. This sequence continues until we've either collected all of the needed entities or the user cancels the task.

The built-in prompts all support letting the user cancel a prompt by saying 'cancel' or 'nevermind'. It's up to you to decide whether that means cancel just the current step or cancel the whole task.

For the delete_alarm intent handler we have a similar waterfall. It's a little simpler because it only needs the title but this waterfall illustrates using two very power features of Bot Builder. You can use EntityRecognizer.findBestMatch() to compare a users utterance against a list of choices and Prompts.choice() to present the user with a list of choices to choose from. Both are very flexible and support fuzzy matching of choices.

Finally, we added a '/notify' dialog to notify the user when their alarm fires. Our simple alarm scheduler triggers this push notification to the user via a call cortanaBot.beginDialog() specifying the address of the user to contact and the name of dialog to start. It can also pass additional arguments to the dialog so in this example we're passing the triggered alarm. The alarm.from & alarm.to aren't that relevant for our simple TextBot based bot but in a real bot you'd need to address the outgoing message with the user you're starting a conversation with so those fields are included here for completeness.

The important thing to note with bot originated dialogs is that they're full dialogs meaning the user can reply to the bots message and that response will get routed to the dialog. This is really powerful because it means that you could notify a user that their alarm trigged and they could reply asking your bot to "snooze it".

If we now run our bot again, we'll get an output similar to:

```
node app.js
set an alarm in 5 minutes called wakeup
Creating alarm named 'wakeup' for 3/24/2016 9:05am
snooze the wakeup alarm
I'm sorry I didn't understand. I can only create & delete alarms.
delete the wakeup alarm
Deleting Alarm
```

All

< Documentation Home

Bot Builder for Node.js

Getting Started

What's new or changed in v3.3

Guides

Core Concepts

Understanding Natural Language

Debug Locally with VSCode

Deploying to Azure

Examples

Chat Bots

UniversalBot

Dialogs

Session

Prompts

IntentDialog

Localization

Calling Bots

UniversalCallBot

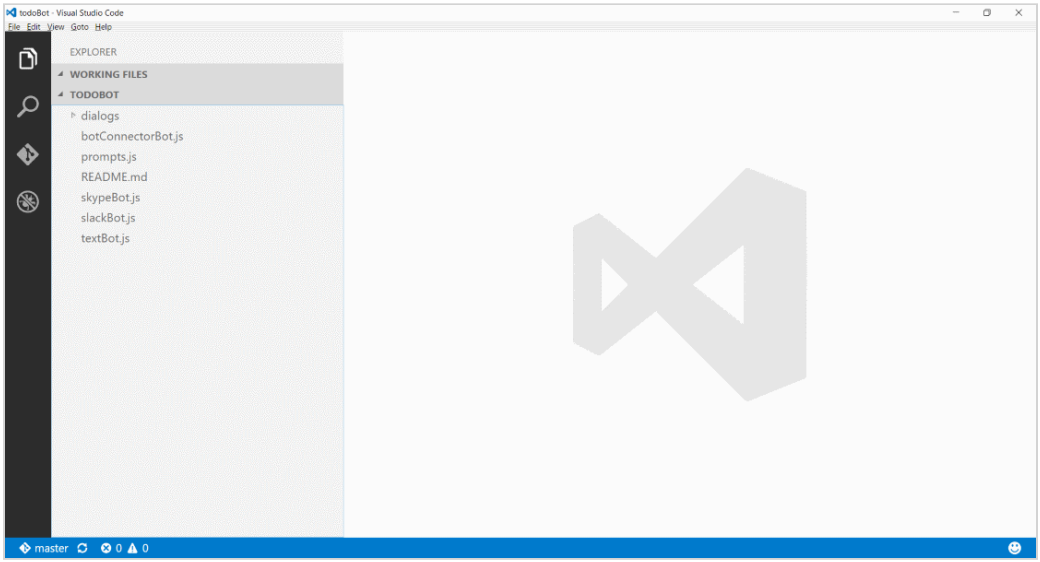
Debug Locally with VSCode

Overview

If you want to debug your bot, you can use the awesome Bot Framework Emulator. One option is to install VSCode and use Bot Builders TextBot class to debug your bot running in a console window. This guide will walk you through doing just that.

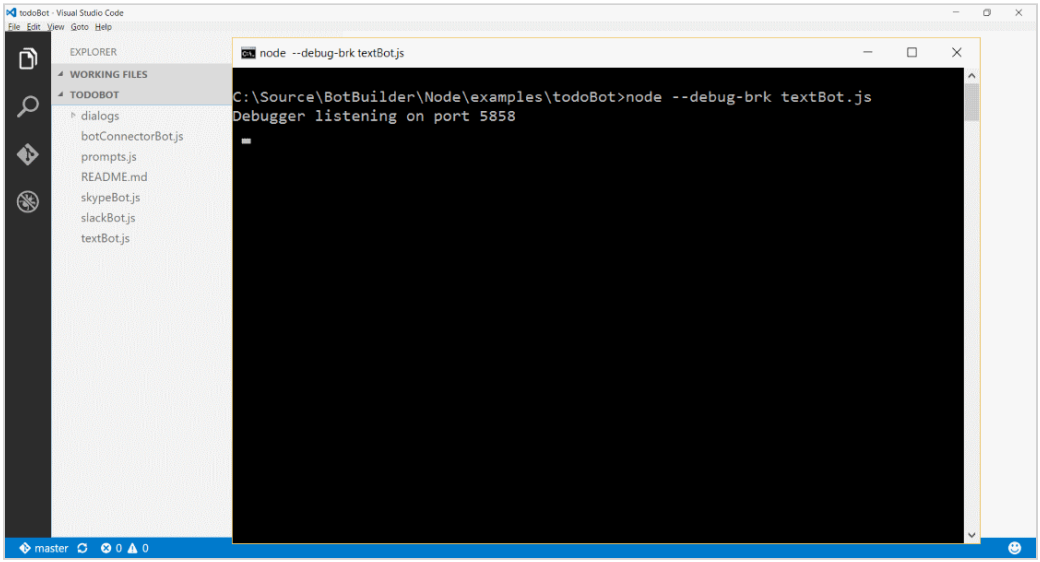
Launch VSCode

For purposes of this walkthrough we'll use Bot Builders TodoBot example. After you install VSCode on your machine you should open your bots project using "open folder".



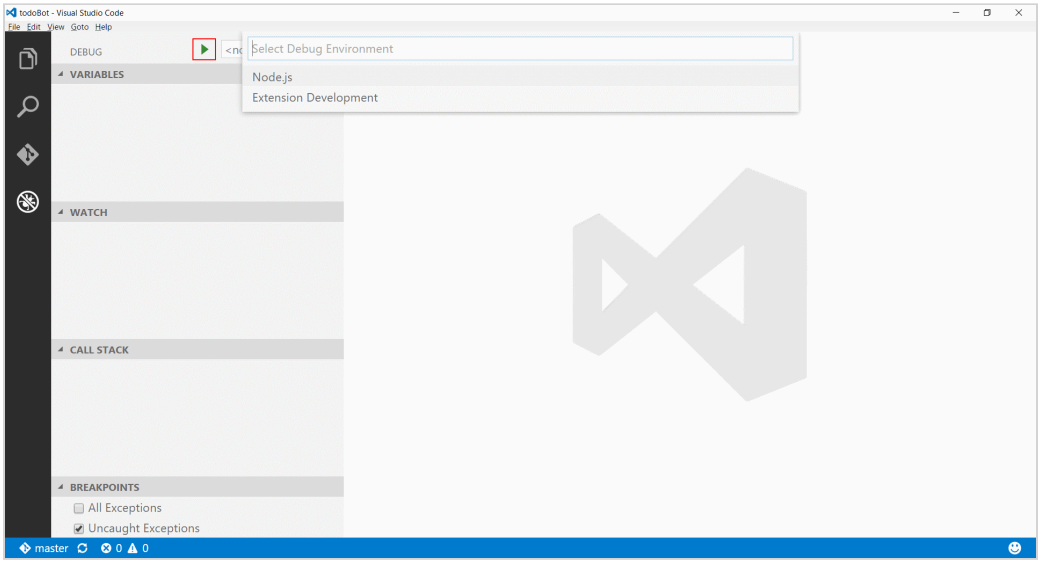
Launch Bot

The TodoBot illustrates running a bot on multiple platforms which is the key to being able to debug your bot locally. To debug locally you need a version of your bot that can run from a console window using the TextBot class. For the TodoBot we can run it locally by launching the textBot.js class. To properly debug this class using VScode we'll want to launch node with the --debug-brk flag which causes it to immediately break. So from a console window type "node --debug-brk textBot.js".



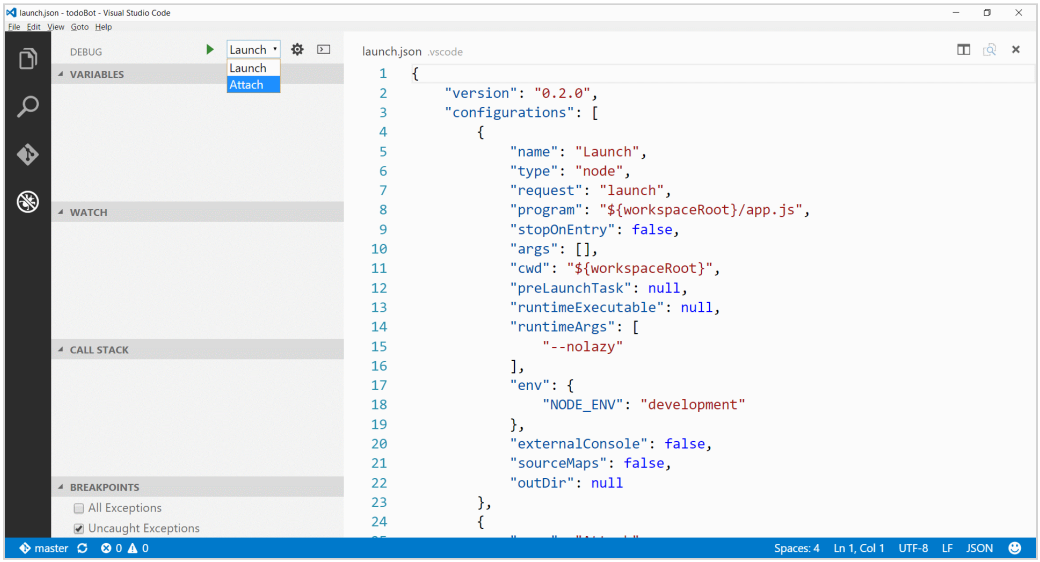
Configure VSCode

Before you can debug your now paused bot you'll need to configure the VSCode node debugger. VSCode knows you project is using node but there are a lot of possible configurations for how you launch node so it wants you to go through a one-time setup for each project (aka folder.) To setup the debugger select the debug tab on the lower left and press the green run button. VSCode will ask you to pick your debug environment and you can just select "node.js". The default settings are fine for our purposes so no need to adjust anything. You should see a .vscode folder added to your project and if you don't want this checked into your Git repository you can add a '/**/.vscode' entry to your .gitignore file.



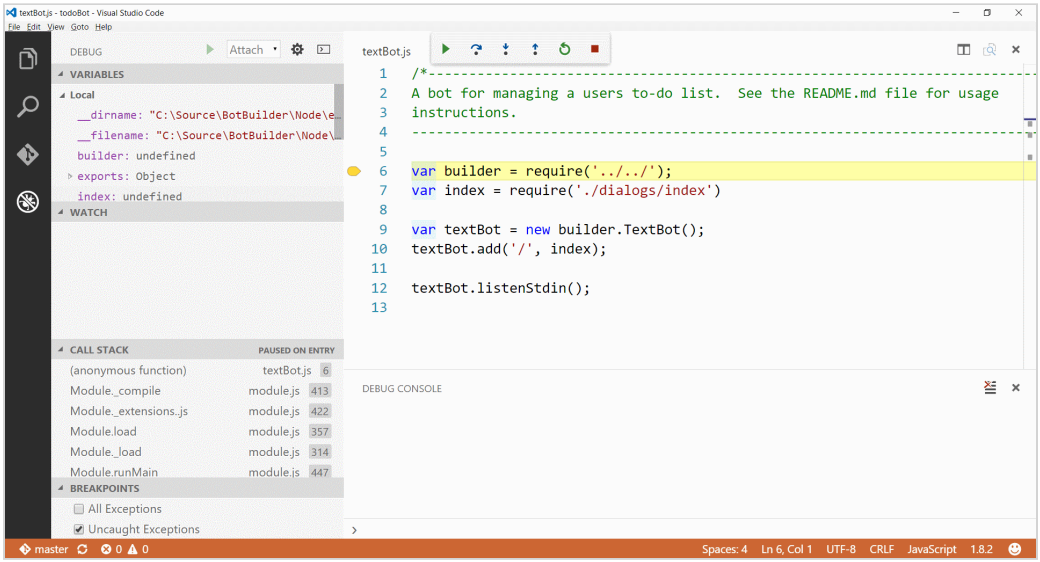
Attach Debugger

Configuring the debugger resulted in two debug modes being added, Launch & Attach. Since our bot is paused in a console window we'll want to select the "Attach" mode and press the green run button again.



Debug Bot

VSCode will attach to your bot paused on the first line of code and now you're ready to set break points and debug your bot! Your bot can be communicated with from the console window so switch back to the console window your bot is running in and say "hello".



All

< Documentation Home

Bot Builder for Node.js

Getting Started

What's new or changed in v3.3

Guides

Core Concepts

Understanding Natural Language

Debug Locally with VSCode

Deploying to Azure

Examples

Chat Bots

UniversalBot

Dialogs

Session

Prompts

IntentDialog

Localization

Calling Bots

UniversalCallBot

Deploying to Azure

There are many ways to deploy your bot to Azure, depending on your situation. We're presenting three ways here that can get you started quickly.

Prerequisites

- In order to complete these steps, you need an Azure Subscription.

I want to setup continuous integration from my local git

This section assumes you have completed the Core Concepts guide.

Step 1: Install the Azure CLI

Open a command prompt, cd into your bot folder, and run the following command:

```
npm install azure-cli -g
```

Step 2: Create an Azure site, and configure it for Node.js/git

First of all, login into your Azure account. Type the following in your command prompt, and follow the instructions.

```
azure login
```

Once logged in, type the following to create a new site, and configure it for Node.js and git

```
azure site create --git <appname>
```

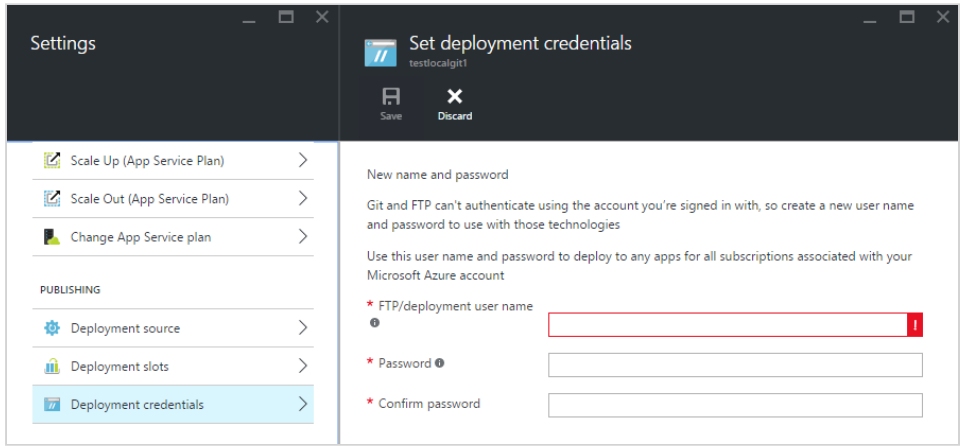
where *<appname>* is the name of the site you want to create. This will result in a url like *appname.azurewebsites.net*.

Step 3: Commit your changes to git, and push to your Azure site

```
git add .
git commit -m "<your commit message>"
git push azure master
```

You will be prompted to enter your deployment credentials. If you don't have them, you can configure them on the Azure Portal by following these simple steps:

1. Visit the Azure Portal
2. Click on the site you've just created, and open the all settings blade
3. In the Publishing section, click on Deployment credentials, enter a username and password, and save



4. Go back to your command prompt, and enter the deployment credentials
5. Your bot will be deployed to your Azure site

Step 4: Test the connection to your bot

Test your bot with the Bot Framework Emulator

I want to setup continuous integration from github

Let's use use the sample bot as a starter template for your own Node.js based bot.

note: in the examples below, replace "echobotsample" with your bot ID for any settings or URLs.

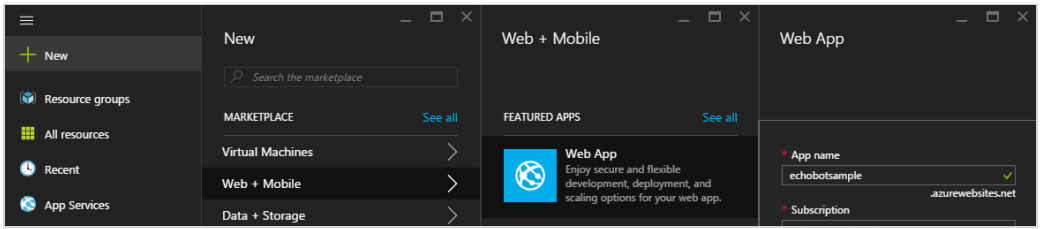
Prerequisites

- Get a Github account

Step 1: Get a github repo

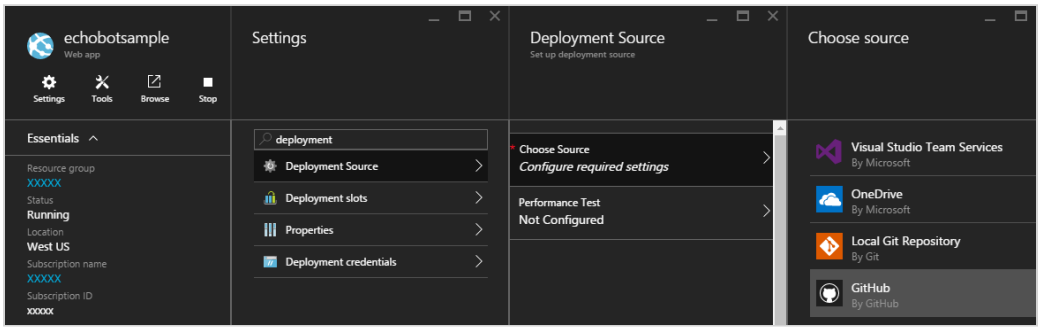
For this tutorial we'll use the sample echobot repo. Fork the echobot repo.

Step 2: Create an Azure web app

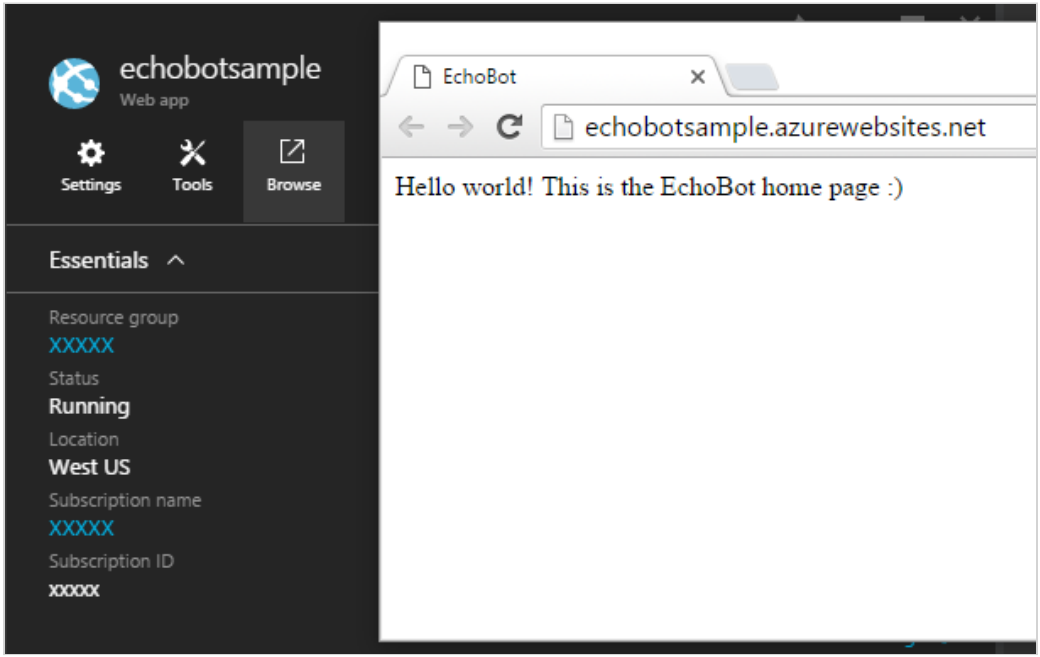


Step 3: Set up continuous deployment to Azure from your Github repo

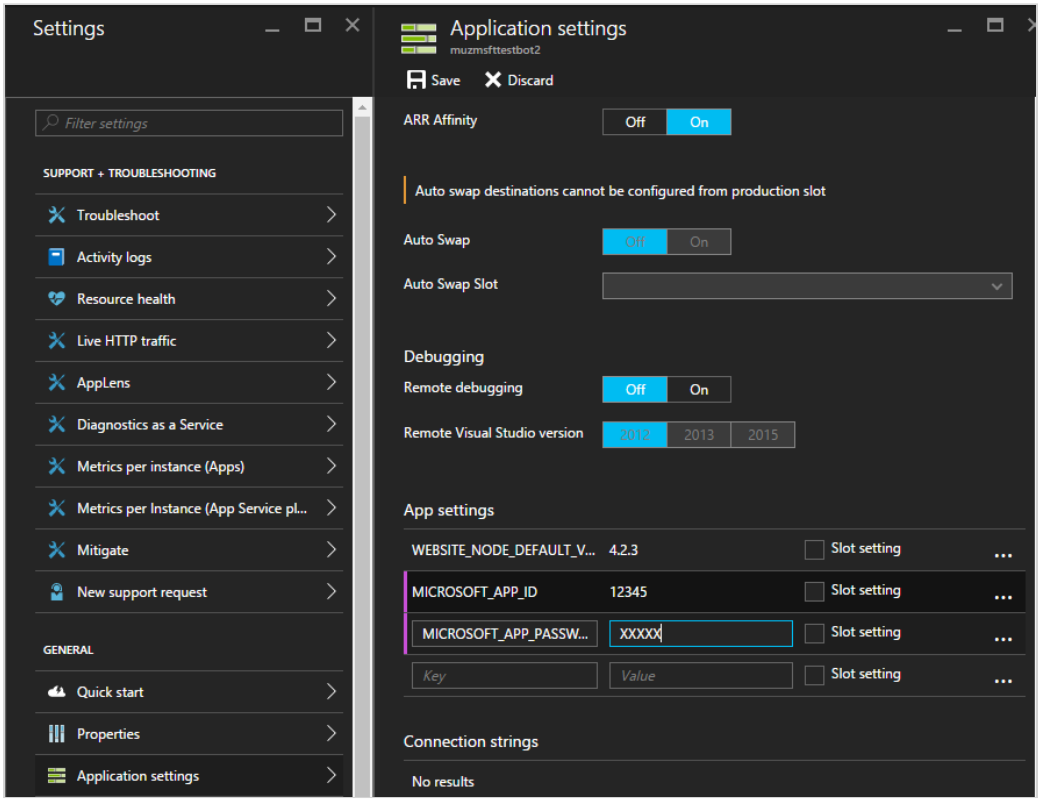
You will be asked to authorize Azure access to your GitHub repo, and then choose your branch from which to deploy.



Verify the deployment has completed by visiting the web app. http://echobotsample.azurewebsites.net/. It may take a minute of two for the initial fetch and build from your repo.



Step 4: Enter your temporary Bot Framework App ID and App Secret into Application settings



- MICROSOFT_APP_ID
- MICROSOFT_APP_PASSWORD

Note: You'll change these values after you register your bot with the Bot Framework Developer Portal.

Step 5: Test the connection to your bot

Test your bot with the Bot Framework Emulator

I want to deploy from Visual studio

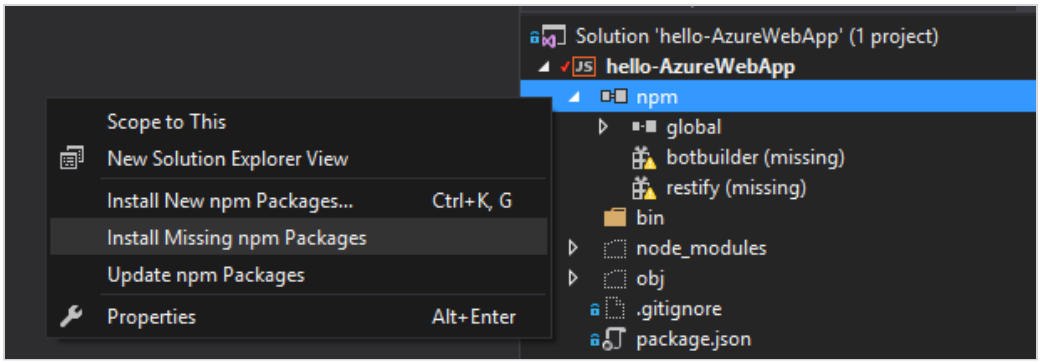
Step 1: Get the Bot Builder SDK samples

Clone the Bot Builder SDK Github repo, by opening a command prompt, choosing a location of your choice (e.g. c:\code), and typing the following:

```
git clone https://github.com/Microsoft/BotBuilder/
```

Step 2: Open the hello-AzureWebApp sample, install the missing npm packages, and configure the temporary appId and appSecret

Open the hello-AzureWebApp.sln solution in Visual Studio, right click on the npm folder, and click on “Install missing npm packages”.



When finished, open the Web.config, and edit it as follows:

```
<appSettings>
  <add key="BOTFRAMEWORK_APPID" value="appid" />
  <add key="BOTFRAMEWORK_APPSECRET" value="appsecret" />
</appSettings>
```

Note: You'll change these values after you register your bot with the Bot Framework Developer Portal.

Step 3: Publish to Azure

1. Right click on the hello-AzureWebApp project in solution explorer, and click on publish
2. Provide your Azure credentials, and then either create a new Web App or select an existing one (if you have created one through the portal)
3. Follow the publishing wizard, and click on Publish

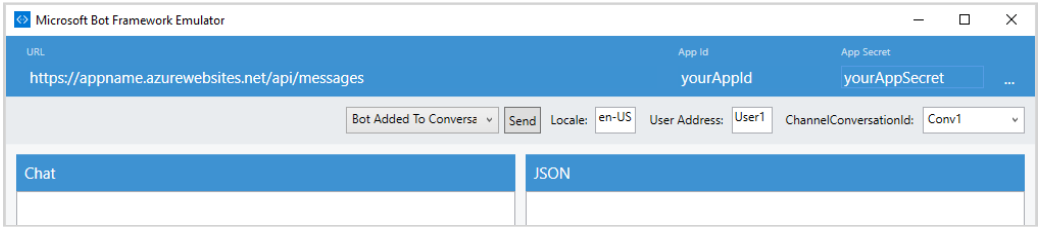
Step 4: Test the connection to your bot

Test your bot with the Bot Framework Emulator

Test your Azure bot with the Bot Framework Emulator

If you have not done it already, install the Bot Framework Emulator. Start the Bot Framework Emulator, and paste the url of your newly deployed bot into the *Bot Url* field. Make sure that:

- The protocol is https
- App Id and App Secret match the values you've set in code



Learn more about the Bot Framework Emulator.

Next steps

- Register your bot with the Bot Framework Developer Portal

All

< Documentation Home

Bot Builder for Node.js

Getting Started

What's new or changed in v3.3

Guides

Core Concepts

Understanding Natural Language

Debug Locally with VSCode

Deploying to Azure

Examples

Chat Bots

UniversalBot

Dialogs

Session

Prompts

IntentDialog

Localization

Calling Bots

UniversalCallBot

UniversalBot

Overview

The UniversalBot class forms the brains of your bot. It's responsible for managing all of the conversations your bot has with a user. You first initialize your bot with a connector that connects your bot to either the Bot Framework or the console. Next you can configure your bot with dialogs that implement the actual conversation logic for your bot.

> **NOTE:** For users of Bot Builder v1.x the BotConnectorBot, TextBot, and SkypeBot class have been deprecated. They will

> continue to function in most cases but developers are encouraged to migrate to the new UniversalBot class at their earliest convenience. The old SlackBot class has been removed from the SDK and unfortunately, at this time the only option is for developers to migrate their native Slack bots to the Bot Framework.

Connectors

The UniversalBot class supports an extensible connector system the lets you configure the bot to receive messages & events and sources. Out of the box, Bot Builder includes a ChatConnector class for connecting to the Bot Framework and a ConsoleConnector class for interacting with a bot from a console window.

ChatConnector

The ChatConnector class configures the UniversalBot to communicate with either the emulator or any of the channels supported by the Bot Framework. Below is an example of a “hello world” bot that’s configured to use the ChatConnector:

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

```
//=========================================================
// Bot Setup
//=========================================================

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

// Create chat bot
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());

//=========================================================
// Bots Dialogs
//=========================================================

bot.dialog('/', function (session) {
    session.send("Hello World");
});
```

The appId & appPassword settings are generally required and will be generated when registering your bot in the developer portal. The one exception to that rule is when running locally against the emulator. When you're first developing your bot you can leave the "Microsoft App Id" & "Microsoft App Password" blank in the emulator and no security will be

enforced between the bot and emulator. When deployed, however, these values are required for proper operation.

The example is coded to retrieve its appId & appPassword settings from environment variables. This sets up the bot to support storing these values in a config file when deployed to a hosting service like Microsoft Azure. When testing your bots security settings locally you'll need to either manually set the environment variables in the console window you're using to run your bot or if you're using VSCode you can add them to the "env" section of your launch.json file.

ConsoleConnector

The ConsoleConnector class configures the UniversalBot to interact with the user via the console window. This connector is primarily useful for quick testing of a bot or for testing on a Mac where you can't easily run the emulator. Below is an example of a "hello world" bot that's configured to use the ConsoleConnector:

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

var connector = new builder.ConsoleConnector().listen();
var bot = new builder.UniversalBot(connector);
bot.dialog('/', function (session) {
    session.send('Hello World');
});
```

If you're debugging your bot using VSCode you'll want to start your bot using a command similar to node `--debug-brk app.js` and then you'll want to start the debugger using attach mode.

Proactive Messaging

The SDK supports two modes of message delivery.

- **Reactive Messages** are messages your bot sends in response to a message received from the user. This is the most common message delivery mode and can be achieved using the session.send() method.
- **Proactive Messages** are messages your bot sends in response to some sort of external event like a timer firing or a notification being triggered. Good examples of proactive messages are a notification that an item has shipped or a daily poll asking team members for their status.

Saving Users Address

The `UniversalBot` class exposes bot.send() and bot.beginDialog() methods for communicating with a user proactively. Before you can use either method you will need to save the address of the user you wish to communicate with. You can do that by serializing the session.message.address property to a string which you then store for future use:

```
bot.dialog('/createSubscription', function (session, args) {
    // Serialize users address to a string.
```

```
        var address = JSON.stringify(session.message.address);

        // Save subscription with address to storage.
        session.sendTyping();
        createSubscription(args.userId, address, function (err) {
            // Notify the user of success or failure and end the dialog.
            var reply = err ? 'unable to create subscription.' : 'subscription
            session.endDialog(reply);
        });
});
```

You shouldn't assume that an `address` object for a user will always be valid. Addresses returned by the `ChatConnector` class in particular contain a `serviceUrl` property which can theoretically change and prevent your bot from contacting the user. Because of that you should consider periodically updating the address object stored for a user.

Sending Messages

To proactively send a message to a user you'll need to first add the web hook or other logic that will trigger your proactive notification. In the example below we've added a web hook to our bot that lets us trigger the delivery of a notification message to one of our bots users:

```
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();
    });
});
```

Within this web hook we're de-serializing the users address which we previously saved. Then we compose a message for the user and deliver it with bot.send(). We can optionally provide a callback to receive the success or failure of the send.

Starting Conversations

In addition to sending messages proactively you can use bot.beginDialog() to start new conversations with a user. 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.

Starting a proactive conversation is very similar to sending a proactive message. In the example below we have a web hook that triggers running a standup with multiple team members. In the web hook we simply loop over all of the team members and call `bot.beginDialog()` with the address of each team member. The bot will then ask the team member their status and roll their answer up into a daily status report:

```
server.post('/api/standup', function (req, res) {
    // Get list of team members to run a standup with.
    var members = req.body.members;
    var reportId = req.body.reportId;
    for (var i = 0; i < members.length; i++) {
        // Start standup for the specified team member
        var user = members[i];
        var address = JSON.parse(user.address);
        bot.beginDialog(address, '/standup', { userId: user.id, reportId:
    }
    res.status(200);
    res.end();
});

bot.dialog('/standup', [
    function (session, args) {
        // Remember the ID of the user and status report
        session.dialogData.userId = args.userId;
        session.dialogData.reportId = args.reportId;

        // Ask user their status
        builder.Prompts.text(session, "What is your status for today?");
    },
    function (session, results) {
        var status = results.response;
```

```
        var userId = session.dialogData.userId;
        var reportId = session.dialogData.reportId;

        // Save their repsonse to the daily status report.
        session.sendTyping();
        saveTeamMemberStatus(userId, reportId, status, function (err)
            if (!err) {
                session.endDialog('Got it... Thanks!');
            } else {
                session.error(err);
            }
        });
    }
]);
```

Proactive Messaging and Localization

For bots that support multiple languages you'll need to **always** use `bot.beginDialog()` to communicate with a user. Thats because the users preferred locale is persisted as part of the Session object which is currently only available within a dialog. Our original notification example can easily be updated to use `bot.beginDialog()` instead of `bot.send()`:

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

    // Send notification as a proactive message
    bot.beginDialog(address, '/notify', { msgId: notification, params: pa
```

```
        res.status(200);
        res.end();
});

bot.dialog('/notify', function (session, args) {
        // Deliver notification to the user.
        session.endDialog(args.msgId, args.params);
});
```

The delivered notification will now use the SDK's built-in localization support to deliver the user a notification in their preferred language. The one disadvantage of using `bot.beginDialog()` over `bot.send()` is that any existing conversation between the bot & user will be ended before sending the user the notification. So for bots that support only a single language, the use of `bot.send()` is still preferred.

All

< Documentation Home

Bot Builder for Node.js

Getting Started

What's new or changed in v3.3

Guides

Core Concepts

Understanding Natural Language

Debug Locally with VSCode

Deploying to Azure

Examples

Chat Bots

UniversalBot

Dialogs

Session

Prompts

IntentDialog

Localization

Calling Bots

UniversalCallBot

Dialogs

Overview

Bot Builder uses dialogs to manage a bots conversations with a user. 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.

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

var connector = new builder.ConsoleConnector().listen();
var bot = new builder.UniversalBot(connector);
bot.dialog('/', [
    function (session, args, next) {
        if (!session.userData.name) {
            session.beginDialog('/profile');
        } else {
            next();
        }
    },
    function (session, results) {
        session.send('Hello %s!', session.userData.name);
    }
]);

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

The example above shows a bot with 2 dialogs. The first message from a user will be routed to the Dialog Handler for the root '/' dialog. This function gets passed a session object which can be used to inspect the users message, send a reply to the user, save state on behalf of the user, or redirect to another dialog.

When a user starts a conversation with our bot we'll first look to see if we know the users name by checking a property off the session.userData object. This data will be persisted across all of the users interactions with the bot and can be used to store things like profile information. If we don't know the users name we're going to redirect them to the '/profile' dialog to ask them their name using a call to session.beginDialog().

The '/profile' dialog is implemented as a waterfall and when beginDialog() is called the first step of the waterfall will be immediately executed. This step simply calls Prompts.text() to ask the user their name. This built-in prompt is just another dialog that gets redirected to. The framework maintains a stack of dialogs for each conversation so if we were to inspect our bots dialog stack at this point it would look something like ['/', '/profile', 'BotBuilder:Prompts']. The conversations dialog stack helps the framework know where to route the users reply to.

When the user replies with their name, the prompt will call session.endDialogWithResult() with the users response. This response will be passed as an argument to the second step of the '/profile' dialogs waterfall. In this step we'll save the users name to session.userData.name property and return control back to the root '/' dialog through a call to endDialog(). At that point the next step of the root '/' dialogs waterfall will be executed ad a custom greeting will be sent to the user.

It's worth noting that the built-in prompts will let the user cancel an action by saying something like 'nevermind' or 'cancel'. It's up to the dialog that called the prompt to determine what cancel means so to detect that a prompt was canceled you can either check the ResumeReason code returned in result.resumed or simply check that result.response isn't null. There are actually a number of reasons that can cause the prompt to return without a response so checking for a null response tends to be the best approach. In our example bot, should the user say 'nevermind' when prompted for their name, the bot would simply ask them for their name again.

Dialog Handlers

A bots dialogs can be expressed using a variety of forms.

Waterfall

Waterfalls will likely be the most common form of dialog you use so understanding how they work is a fundamental skill in bot development. Waterfalls let you collect input from a user using a sequence of steps. A bot is always in a state of providing a user with information or asking a question and then waiting for input. In the Node version of Bot Builder its waterfalls that drive this back-n-forth flow.

Paired with the built-in Prompts you can easily prompt the user with a series of questions:

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

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. 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. You can also call [session.beginDialog())] to start one of your own dialogs to move the conversation to the next step:

```
bot.dialog('/', [
    function (session) {
        session.beginDialog('/askName');
    },
    function (session, results) {
        session.send('Hello %s!', results.response);
    }
```

```
]);
bot.dialog('/askName', [
    function (session) {
        builder.Prompts.text(session, 'Hi! What is your name?');
    },
    function (session, results) {
        session.endDialogWithResult(results);
    }
]);
```

This achieves the same basic behavior as before but calls a child dialog to prompt for the users name. That's somewhat pointless in this example but could be a useful way of partitioning the conversation if you had multiple profile fields you wanted to populate.

This example can actually be simplified some. All waterfalls contain a phantom last step which automatically returns the result from the last step so we could actually simplify this to:

```
bot.dialog('/', [
    function (session) {
        session.beginDialog('/askName');
    },
    function (session, results) {
        session.send('Hello %s!', results.response);
    }
]);
bot.dialog('/askName', [
    function (session) {
        builder.Prompts.text(session, 'Hi! What is your name?');
```

```
    }
]);
```

The first step of a waterfall can receive arguments passed to the dialog and every step receives a `next()` function that can be used to advance the waterfall forward manually. In the example below we've paired these two features together to create an '/ensureProfile' dialog that will verify that a users profile is filled in and prompt the user for any missing fields. This pattern would let us add fields to the profile later that would be automatically filled in as users message the bot:

```
bot.dialog('/', [
    function (session) {
        session.beginDialog('/ensureProfile', session.userData.profile);
    },
    function (session, results) {
        session.userData.profile = results.response;
        session.send('Hello %(name)s! I love %(company)s!', session.user
    }
]);
bot.dialog('/ensureProfile', [
    function (session, args, next) {
        session.dialogData.profile = args || {};
        if (!session.dialogData.profile.name) {
            builder.Prompts.text(session, "What's your name?");
        } else {
            next();
        }
    },
    function (session, results, next) {
        if (results.response) {
```

```
            session.dialogData.profile.name = results.response;
        }
        if (!session.dialogData.profile.company) {
            builder.Prompts.text(session, "What company do you work f
        } else {
            next();
        }
    },
    function (session, results) {
        if (results.response) {
            session.dialogData.profile.company = results.response;
        }
        session.endDialogWithResult({ response: session.dialogData.prof
    }
]);
```

In the '/ensureProfile' dialog we're using session.dialogData to temporarily hold the users profile. We do this because when our bot is distributed across multiple compute nodes, every step of the waterfall could be processed by a different compute node. The dialogData field ensures that the dialogs state is properly maintained between each turn of the conversation. You can store anything you want into this field but should limit yourself to JavaScript primitives that can be properly serialized.

It's worth noting that the `next()` function can be passed an [IDialogResult](idialogresult) so it can mimic any results returned from a built-in prompt or other dialog which sometimes simplifies your bots control logic.

Closure

You can also pass a single function for your dialog handler which simply results in creating a 1 step waterfall:

```
bot.dialog('/', function (session) {
    session.send("Hello World");
});
```

Dialog Object

For more specialized dialogs you can add an instance of a class that derives from the Dialog base class. This gives maximum flexibility for how your bot behaves as the built-in prompts and even waterfalls are implemented internally as dialogs.

```
bot.dialog('/', new builder.IntentDialog()
    .matches(/^hello/i, function (session) {
        session.send("Hi there!");
    })
    .onDefault(function (session) {
        session.send("I didn't understand. Say hello to me!");
    }));
```

SimpleDialog

Implementing a new Dialog from scratch can be tricky as there are a lot of things to consider. To try and cover the bulk of the scenarios not covered

by waterfalls, we include a SimpleDialog class. The closure passed to this class works very similar to a waterfall step with the exception that the results from calling a built-in prompt or other dialog will be passed back to one closure. Unlike a waterfall there's no phantom step that the conversation is advanced to. This is powerful but also dangerous as your bot can easily get stuck in a loop so care should be used:

```
bot.dialog('/', new builder.SimpleDialog(function (session, results) {
    if (results && results.response) {
        session.send(results.response.toString('base64'));
    }
    builder.Prompts.text(session, "What would you like to base64 encode
}));
```

The above example is a base64 bot that all it does is convert a users input to base64. It sits in a tight loop prompting the user for string which it then encodes.

All

< Documentation Home

Bot Builder for Node.js

Getting Started

What's new or changed in v3.3

Guides

Core Concepts

Understanding Natural Language

Debug Locally with VSCode

Deploying to Azure

Examples

Chat Bots

UniversalBot

Dialogs

Session

Prompts

IntentDialog

Localization

Calling Bots

UniversalCallBot

Session

Overview

The session object is passed to your dialog handlers anytime your bot receives a message from the user. The session object is the primary mechanism you'll use to send messages to the user and to manipulate the bots dialog stack.

Sending Messages

The session.send() method can be used to easily send messages, attachments, and rich cards to the user. Your bot is free to call `send()` as many times as it likes in response to a message from the user. When sending multiple replies, the individual replies will be automatically grouped into a batch and delivered to the user as a set in an effort to preserve the original order of the messages.

> **Auto Batching**
>
> When a bot sends multiple replies to a user using `session.send()`, those replies will automatically be grouped into a batch using a feature called "auto batching." Auto batching works by waiting a default of 250ms after every call to `send()` for an additional call to `send()`. To avoid a 250ms pause after the last call to `send()` you can manually trigger delivery of the batch by calling session.sendBatch(). In practice it's rare that you actually need to call sendBatch() as the built-in prompts and session.endConversation() automatically call sendBatch() for you.
>
> The goal of batching is to try and avoid multiple messages from the bot being displayed out of order. Unfortunately, not

> all chat clients can guarantee this. In particular, the clients tend to want to download images before displaying a message to the user so if you send a message containing an image followed immediately by a message without images you'll sometimes see the messages flipped in the users feed. To minimize the chance of this you can try to insure that your images are coming from CDNs and try to avoid the use of overly large images. In extreme cases you may even need to insert a 1-2 second delay between the message with the image and the one without. You can make this delay feel a bit more natural to the user by calling session.sendTyping() before starting your delay.
>
> The SDKs auto batching delay is configurable so If you'd like to disable the SDK's auto-batching logic all together you can set the default delay to a large number and then manually call sendBatch() with a callback that will be invoked after the batch has been delivered.

Text Messages

To send a simple text message to the user you can simply call `session.send("hello there")`. The message can also contain template parameters which can be expanded using `session.send("hello there`

`%s", name)`. The SDK currently uses a library called node-sprintf to implement the template functionality so for a full list of what's possible consult the sprintf documentation.

> **NOTE:** One word of caution about using named parameters with sprintf. It's a common mistake to forget the trailing format specifier (i.e. the 's') when using named parameters `session.send("Hello there %(name)s", user)`. If that happens you'll get an invalid template exception at runtime. Use that as an indicator that you should verify your templates are correct.

Attachments

Many chat services support sending image, video, and file attachments to the user. You can use `session.send()` for this as well but you'll need to use it in conjunction with the SDK's Message builder class. You can use either the attachments() or addAttachment() methods to create a message containing an image:

```
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/Be
                }]);
            session.endDialog(msg);
        }
]);
```

Cards

Several chat services are starting to support the sending of rich cards to the user containing text, images, and even buttons. Not all chat services support cards or have the same level of richness so you'll need to consult the individual services documentation to determine what's currently supported.

The BotBuilder SDK contains a set of helper classes that can be used to render cards to the user in a cross platform way. The HeroCard and ThumbnailCard classes can be used to render a card with some text, an image, and optional buttons. On most channels you'll notice no difference between these two cards but on skype you can use them to control the presentation of the image to the user.

```
bot.dialog('/cards', [
    function (session) {
        var msg = new builder.Message(session)
            .textFormat(builder.TextFormat.xml)
            .attachments([
                new builder.HeroCard(session)
```

```
                .title("Hero Card")
                .subtitle("Space Needle")
                .text("The <b>Space Needle</b> is an observation
                .images([
                    builder.CardImage.create(session, "https://uplo
                ])
                .tap(builder.CardAction.openUrl(session, "https://en
            ]);
        session.endDialog(msg);
    }
]);
```

Cards can typically have tap actions which let you specify what should happen when the user taps on the card (open a link, send a message, etc.) You can use the CardAction class to customize this behavior. There are several different types of actions you can specify but most channels only support a handful of actions. You'll generally want to stick with using the openUrl(), postBack(), and dialogAction() actions for maximum compatibility.

> **NOTE:** The differences between postBack() and imBack() actions is subtle. The intention is that imBack() will show the message beign sent to the bot in the users feed where postBack() will hide the sent message from the user. Not all channels (like Skype) currently support postBack() so those channels will simply fall back to using imBack(). This generally won't change how your bot behaves but it does

> mean that if you're including data like an order id in your postBack() it may be visible on certain channels when you didn't expect it to be.

The SDK also supports more specialized card types like ReceiptCard and SigninCard. It's not practical for the SDK to support every card or attachment format supported by the underlying chat service so we have a Message.sourceEvent() method that you can use to send custom messages & attachments in the channel native schema:

```
bot.dialog('/receipt', [
    function (session) {
        session.send("You can send a receipts for facebook using Bot Bui
        var msg = new builder.Message(session)
            .attachments([
                new builder.ReceiptCard(session)
                    .title("Recipient's Name")
                    .items([
                        builder.ReceiptItem.create(session, "$22.00", "
                        builder.ReceiptItem.create(session, "$22.00", "
                    ])
                    .facts([
                        builder.Fact.create(session, "1234567898", "Or
                        builder.Fact.create(session, "VISA 4076", "Payn
                    ])
                    .tax("$4.40")
                    .total("$48.40")
            ]);
        session.send(msg);
```

```
session.send("Or using facebooks native attachment schema...");
msg = new builder.Message(session)
    .sourceEvent({
        facebook: {
            attachment: {
                type: "template",
                payload: {
                    template_type: "receipt",
                    recipient_name: "Stephane Crozatier",
                    order_number: "12345678902",
                    currency: "USD",
                    payment_method: "Visa 2345",
                    order_url: "http://petersapparel.parseapp.
                    timestamp: "1428444852",
                    elements: [
                        {
                            title: "Classic White T-Shirt",
                            subtitle: "100% Soft and Luxuriou
                            quantity: 2,
                            price: 50,
                            currency: "USD",
                            image_url: "http://petersapparel.
                        },
                        {
                            title: "Classic Gray T-Shirt",
                            subtitle: "100% Soft and Luxuriou
                            quantity: 1,
                            price: 25,
                            currency: "USD",
                            image_url: "http://petersapparel.
                        }
                    ],
                    address: {
                        street_1: "1 Hacker Way",
                        street_2: "",
```

```
                                    city: "Menlo Park",
                                    postal_code: "94025",
                                    state: "CA",
                                    country: "US"
                                },
                                summary: {
                                    subtotal: 75.00,
                                    shipping_cost: 4.95,
                                    total_tax: 6.19,
                                    total_cost: 56.14
                                },
                                adjustments: [
                                    { name: "New Customer Discount", a
                                    { name: "$10 Off Coupon", amount:
                                ]
                            }
                        }
                    }
                });
            session.endDialog(msg);
        }
]);
```

Typing Indicator

Not all chat services support sending typing events, but for those that do you can use session.sendTyping() to tell the user that the bot is actively composing a reply. This is particular useful if you're about to begin an asynchronous operation that could take a few seconds to complete. The amount of time the indicator stays on varies by service (Slack is 3 seconds and Facebook is 20 seconds) so as a general rule, if you need the indicator

to stay on for more than a few seconds you should add logic to call sendTyping() periodically.

```
bot.dialog('/countItems', function (session, args) {
    session.sendTyping();
    lookupItemsAsync(args, function (err, items) {
        if (!err) {
            session.send("%d items found", items.length);
        } else {
            session.error(err);
        }
    });
});
```

Dialog Stack

With the Bot Builder SDK you'll use dialogs to organize your bots conversations with the user. The bot tracks where it is in the conversation with a user using a stack that's persisted to the bots storage system. When the bot receives the first message from a user it will push the bots default dialog onto the stack and pass that dialog the users message. The dialog can either process the incoming message and send a reply directly to the user or it can start other dialogs which will guide the user through a series of questions that collect input from the user needed to complete some task.

The session includes several methods for managing the bots dialog stack and therefore manipulate where the bot is conversationally with the user. Once you get the hang of working with the dialog stack you can use a combination of dialogs and the sessions stack manipulation methods to achieve just about any conversational flow you can dream of.

Starting and Ending dialogs

You can use session.beginDialog() to call a dialog (pushing it onto the stack) and then either session.endDialog() or session.endDialogWithResults() to return control back to the caller (popping off the stack.) When paired with waterfalls you have a simple mechanism for driving conversations forward. The example below uses two waterfalls to prompt the user for their name and then provide them with a custom greeting:

```
bot.dialog('/', [
    function (session) {
        session.beginDialog('/askName');
    },
    function (session, results) {
        session.send('Hello %s!', results.response);
    }
]);
bot.dialog('/askName', [
    function (session) {
        builder.Prompts.text(session, 'Hi! What is your name?');
    },
```

```
    function (session, results) {
        session.endDialogWithResult(results);
    }
]);
```

If you were to run through this sample using the emulator you would see a console output like below:

```
restify listening to http://[::]:3978
ChatConnector: message received.
session.beginDialog(/)
/ - waterfall() step 1 of 2
/ - session.beginDialog(/askName)
./askName - waterfall() step 1 of 2
./askName - session.beginDialog(BotBuilder:Prompts)
..Prompts.text - session.send()
..Prompts.text - session.sendBatch() sending 1 messages

ChatConnector: message received.
..Prompts.text - session.endDialogWithResult()
./askName - waterfall() step 2 of 2
./askName - session.endDialogWithResult()
/ - waterfall() step 2 of 2
/ - session.send()
/ - session.sendBatch() sending 1 messages
```

We can see that the user sent two messages to the bot. The first message of “hi” resulted in the default “/” dialog being pushed onto the stack, entering step 1 of the first waterfall. That step called `beginDialog()` and pushed “/askName” onto the stack, entering step 1 of the second waterfall. That step then called Prompts.text() to ask the user their name. Prompts are themselves dialogs so we see that going onto the stack (notice that you can

use the dots "." prefixing each line of console output to tell your current stack depth.)

When the user replies with their name we can see the text() prompt return their input to the second waterfall using `endDialogWithResults()`. The waterfall passes this value to the step 2 which itself calls `endDialogWithResult()` to pass it back to the first waterfall. The first waterfall passes that result to step 2 which is where we generate the actual personalized greeting that's sent to the user.

In the example we used `session.endDialogWithResults()` to return control back to the caller and pass them a value (the users input.) You can pass your own complex values back to the caller using `session.endDialogWithResults({ response: { name: 'joe smith', age: 37 } })`. We can also return control a send the user a message using `session.endDialog("ok… operation canceled")` or just simply return control to the caller using `session.endDialog()`.

When calling a dialog with `session.beginDialog()` you can optionally pass in a set of arguments which lets you truly call dialogs in much the same way you would a function. We can update our previous example to prompt the user to provide their profile information and then remember it for future conversations:

```
bot.dialog('/', [
    function (session) {
        session.beginDialog('/ensureProfile', session.userData.profile);
    },
    function (session, results) {
        session.userData.profile = results.profile;
        session.send('Hello %s!', session.userData.profile.name);
    }
]);
bot.dialog('/ensureProfile', [
    function (session, args, next) {
        session.dialogData.profile = args || {};
        if (!args.profile.name) {
            builder.Prompts.text(session, "Hi! What is your name?");
        } else {
            next();
        }
    },
    function (session, results, next) {
        if (results.response) {
            session.dialogData.profile.name = results.response;
        }
        if (!args.profile.email) {
            builder.Prompts.text(session, "What's your email address?");
        } else {
            next();
        }
    },
    function (session, results) {
        if (results.response) {
            session.dialogData.profile.email = results.response;
        }
        session.endDialogWithResults({ repsonse: session.dialogData.pro
    }
]);
```

We're using session.userData to remember the users `profile` so we'll pass that to our "/ensureProfile" dialog in the call to `beginDialog()`. Dialogs can use session.dialogData to temporarily hold values they're working on. We'll use that to hold the `profile` object we were passed. On the first call this will be undefined so we'll initialize a new profile that we'll fill in. We can use the `next()` function passed to every waterfall step to essentially skip any fields that are already filled in.

Replacing Dialogs

The session.replaceDialog() method lets you end the current dialog and replace it with a new one without returning to the caller. This method can be used to create a number of interesting flows. One of the most useful being the creation of loops.

The SDK includes a useful set of built-in prompts but there will be times when you'll want to create your own custom prompts that either add some custom validation logic. Using a combination of `Prompts.text()` and `session.replaceDialog()` you can easily build new prompts. The example below shows how to build a fairly flexible phone number prompt:

```
bot.dialog('/', [
    function (session) {
```

```
            session.beginDialog('/phonePrompt');
        },
        function (session, results) {
            session.send('Got it... Setting number to %s', results.response);
        }
]);
bot.dialog('/phonePrompt', [
        function (session, args) {
            if (args && args.reprompt) {
                builder.Prompts.text(session, "Enter the number using a forn
            } else {
                builder.Prompts.text(session, "What's your phone number?")
            }
        },
        function (session, results) {
            var matched = results.response.match(/\d+/g);
            var number = matched ? matched.join('') : '';
            if (number.length == 10 || number.length == 11) {
                session.endDialogWithResult({ response: number });
            } else {
                session.replaceDialog('/phonePrompt', { reprompt: true });
            }
        }
]);
```

The SDK's stack based model for managing the conversation with a user is useful but not always the best approach for every bot. Some bots, like a text adventure bot, might be better served using a more state machine like model where the user is moved from one state or location to the next. This pattern can easily be achieved using `replaceDialog()` to transition between the states:

```
bot.dialog('/', [
    function (session) {
        session.send("You're in a large clearing. There's a path to the nor
        builder.Prompts.choice(session, "command?", ["north", "look"]);
    },
    function (session, results) {
        switch (results.repsonse.entity) {
            case "north":
                session.replaceDialog("/room1");
                break;
            default:
                session.replaceDialog("/");
                break;
        }
    }
]);
bot.dialog('/room1', [
    function (session) {
        session.send("There's a small house here surrounded by a white f
        builder.Prompts.choice(session, "command?", ["open gate", "sou
    },
    function (session, results) {
        switch (results.repsonse.entity) {
            case "open gate":
                session.replaceDialog("/room2");
                break;
            case "south":
                session.replaceDialog("/");
                break;
            case "west":
                session.replaceDialog("/room3");
                break;
            default:
                session.replaceDialog("/room1");
                break;
```

```
            }
        }
]);
```

This example creates a seperate dialog for every location and moves from location-to-location using `replaceDialog()`. We can make things more data driven using something like:

```
var world = {
    "room0": {
        description: "You're in a large clearing. There's a path to the nort
        commands: { north: "room1", look: "room0" }
    },
    "room1": {
        description: "There's a small house here surrounded by a white f
        commands: { "open gate": "room2", south: "room0", west: "roo
    }
}

bot.dialog('/', [
    function (session, args) {
        session.beginDialog("/location", { location: "room0" });
    },
    function (session, results) {
        session.send("Congratulations! You made it out!");
    }
]);
bot.dialog('/location', [
    function (session, args) {
        var location = world[args.location];
        session.dialogData.commands = location.commands;
        builder.Prompts.choice(session, location.description, location.co
```

```
    },
    function (session, results) {
        var destination = session.dialogData.commands[results.respons
        session.replaceDialog("/location", { location: destination });
    }
]);
```

Canceling Dialogs

Sometimes you may want to do more extensive stack manipulation. For that you can use the session.cancelDialog() to end a dialog at any arbitrary point in the dialog stack and optionally start a new dialog in its place. You can call `session.cancelDialog('/placeOrder')` with the ID of a dialog to cancel. The stack will be searched backwards and the first occurrence of that dialog will be canceled causing that dialog plus all of its children to be removed from the stack. Control will be returned to the original caller and they can check for a results.resumed code equal to ResumeReason.notCompleted to detect the cancelation.

You can also pass the zero-based index of the dialog to cancel. Calling `session.cancelDialog(0, '/storeHours')` with an index of 0 and the ID of a new dialog to start lets you easily terminate any active task and start a new one in its place.

Ending Conversations

The session.endConversation() method provides a convenient method for quickly terminating a conversation with a user. This could be in response to the user saying "goodbye" or because you've simply completed the users task. While you can technically end the conversation using `session.cancelDialog(0)` there are a few advantages to using `endConversation()` instead.

The `endConversation()` method not only clears the dialog stack, it also clears the entire session.privateConversationData variable that gets persisted to storage. That means you can use `privateConversationData` to cache state relative to the current task and so long as you call `endConversation()` when the task is completed all of this state will be automatically cleaned up.

You can pass a message to `session.endConversation("Ok… Goodbye.")` and `session.sendBatch()` will be automatically called causing the message to be immediately sent to the user. It's also worth noting that anytime your bot throws an exception, `endConversation()` gets called with a configurable error message in an effort to return the bot to a consistent state.

Using Session in Callbacks

Inevitably you're going to want to make some asynchronous network call to retrieve data and then send those results to the user using the session object. This is completely fine but there are a few best practices you'll want to follow.

ok to use session

If you're making an async call in the context of a message you received from the user you're generally ok calling `session.send()`:

```
bot.dialog('listItems', function (session) {
    session.sendTyping();
    lookupItemsAsync(function (results) {
        // OK to call session.send() here.
        session.send(results.message);
    });
});
```

not ok to use session

Here's a common mistake we see developers make. They start an asynchronous call and then immediately call something like `session.endDialog()` that changes their bots conversation state:

```
bot.dialog('listItems', function (session) {
    session.sendTyping();
```

```
    lookupItemsAsync(function (results) {
        // Calling session.send() here is dangerous because you've done a
        // triggered a change in your bots conversation state.
        session.send(results.message);
    });
    session.endDialog();
});
```

In general this is a pattern you should avoid. The correct way to achieve the above behavior is to move the `endDialog()` call into your callback:

```
bot.dialog('listItems', function (session) {
    session.sendTyping();
    lookupItemsAsync(function (results) {
        // Calling session.send() here is dangerous because you've done a
        // triggered a change in your bots conversation state.
        session.endDialog(results.message);
    });
});
```

dangerous to use session

The other case where you probably shouldn't use session (or at least should be careful) is when you're doing some long running task and you wish to communicate with the user at the beginning and end of the task:

```
bot.dialog('orderPizza', function (session, args) {
    session.send("Starting your pizza order...");
```

```
    queue.startOrder({ session: session, order: args.order });
});

queue.orderReady(function (session, order) {
    session.send("Your pizza is on its way!");
});
```

This can be dangerous because the bots server could crash or the user could send other messages while the bot is doing the task and that could leave the bot in a bad conversation state. The better approach is to persist the users session.message.address object and then send them a proactive message using bot.send() once their order is ready:

```
bot.dialog('orderPizza', function (session, args) {
    session.send("Starting your pizza order...");
    queue.startOrder({ address: session.message.address, order: args.or
});

queue.orderReady(function (address, order) {
    var msg = new builder.Message()
        .address(address)
        .text("Your pizza is on its way!");
    bot.send(msg);
});
```

Prompts

Collecting Input

Bot Builder comes with a number of built-in prompts that can be used to collect input from a user.

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.

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

These built-in prompts are implemented as a Dialog so they'll return the users response through a call to session.endDialogWithresult(). Any DialogHandler can receive the result of a dialog but waterfalls tend to be the simplest way to handle a prompt result.

Prompts return to the caller an IPromptResult. The users response will be contained in the results.response field and may be null. There are a number of reasons for the response to be null. The built-in prompts let the user cancel an action by saying something like 'cancel' or 'nevermind' which will result in a null response. Or the user may fail to enter a properly formatted response which can also result in a null response. The exact reason can be determined by examining the ResumeReason returned in result.resumed.

Prompts.text()

The Prompts.text() method asks the user for a string of text. The users response will be returned as an IPromptTextResult.

```
builder.Prompts.text(session, "What is your name?");
```

Prompts.confirm()

The Prompts.confirm() method will ask the user to confirm an action with yes/no response. The users response will be returned as an IPromptConfirmResult.

```
builder.Prompts.confirm(session, "Are you sure you wish to cancel your or
```

Prompts.number()

The Prompts.number() method will ask the user to reply with a number. The users response will be returned as an IPromptNumberResult.

```
builder.Prompts.number(session, "How many would you like to order?");
```

Prompts.time()

The Prompts.time() method will ask the user to reply with a time. The users response will be returned as an IPromptTimeResult. The framework uses a library called Chrono to parse the users response and supports both relative “in 5 minutes” and non-relative “june 6th at 2pm” type responses.

The results.response returned is an entity that can be resolved into a JavaScript Date object using EntityRecognizer.resolveTime().

```
bot.dialog('/createAlarm', [
    function (session) {
        session.dialogData.alarm = {};
        builder.Prompts.text(session, "What would you like to name this
    },
    function (session, results, next) {
        if (results.response) {
            session.dialogData.name = results.response;
            builder.Prompts.time(session, "What time would you like to
        } else {
            next();
        }
    },
    function (session, results) {
        if (results.response) {
            session.dialogData.time = builder.EntityRecognizer.resolveT
        }

        // Return alarm to caller
        if (session.dialogData.name && session.dialogData.time) {
            session.endDialogWithResult({
                response: { name: session.dialogData.name, time: sess
            });
        } else {
            session.endDialogWithResult({
                resumed: builder.ResumeReason.notCompleted
            });
        }
    }
]);
```

Prompts.choice()

The Prompts.choice() method asks the user to pick an option from a list. The users response will be returned as an IPromptChoiceResult. The list of choices can be presented to the user in a variety of styles via the IPromptOptions.listStyle property. The user can express their choice by either entering the number of the option or its name. Both full and partial matches of the options name are supported.

The list of choices can be passed to Prompts.choice() in a variety of ways. As a pipe '|' delimited string.

```
builder.Prompts.choice(session, "Which color?", "red|green|blue");
```

As an array of strings.

```
builder.Prompts.choice(session, "Which color?", ["red","green","blue"]);
```

Or as an Object map. When an Object is passed in Objects keys will be used to determine the choices.

```
var salesData = {
    "west": {
        units: 200,
        total: "$6,000"
    },
    "central": {
        units: 100,
```

```
            total: "$3,000"
        },
        "east": {
            units: 300,
            total: "$9,000"
        }
};

bot.dialog('/', [
    function (session) {
        builder.Prompts.choice(session, "Which region would you like sal
    },
    function (session, results) {
        if (results.response) {
            var region = salesData[results.response.entity];
            session.send("We sold %(units)d units for a total of %(total)s
        } else {
            session.send("ok");
        }
    }
]);
```

Prompts.attachment()

The Prompts.attachment() method will ask the user to upload a file attachment like an image or video. The users response will be returned as an IPromptAttachmentResult.

```
builder.Prompts.attachment(session, "Upload a picture for me to transforr
```

Dialog Actions

Dialog actions offer shortcuts to implementing common actions. The DialogAction class provides a set of static methods that return a closure which can be passed to anything that accepts a dialog handler. This includes but is not limited to UniversalBot.dialog(), Library.dialog(), IntentDialog.matches(), and IntentDialog.onDefault().

Action Type	Description
DialogAction.send	Sends a static message to the user.
DialogAction.beginDialog	Passes control of the conversation to a new dialog.
DialogAction.endDialog	Ends the current dialog.
DialogAction.validatedPrompt	Creates a custom prompt by wrapping one of the built-in prompts with a validation routine.

IntentDialog

Overview

The IntentDialog class lets you listen for the user to say a specific keyword or phrase. We call this intent detection because you are attempting to determine what the user is intending to do. IntentDialogs are useful for building more open ended bots that support natural language style understanding. For an in depth walk through of using IntentDialogs to add natural language support to a bot see the Understanding Natural Language guide.

> **NOTE:** For users of Bot Builder v1.x the CommandDialog and LuisDialog classes have been deprecated. These classes will continue to function but developers are encouraged to upgrade to the more flexible IntentDialog class at their earliest convenience.

Matching Regular Expressions

The IntentDialog.matches() method lets you trigger a handler based on the users utterance matching a regular expressions. The handler itself can take a variety of forms.

A waterfall when you need to collect input from the user:

```
var intents = new builder.IntentDialog();
bot.dialog('/', intents);

intents.matches(/^echo/i, [
    function (session) {
        builder.Prompts.text(session, "What would you like me to say?");
    },
    function (session, results) {
        session.send("Ok... %s", results.response);
    }
]);
```

A simple closure that behaves as 1 step waterfall:

```
var intents = new builder.IntentDialog();
bot.dialog('/', intents);

intents.matches(/^version/i, function (session) {
    session.send('Bot version 1.2');
});
```

A DialogAction that can provide a shortcut for implementing simpler closures:

```
var intents = new builder.IntentDialog();
bot.dialog('/', intents);

intents.matches(/^version/i, builder.DialogAction.send('Bot version 1.2'));
```

Or the ID of a dialog to redirect to:

```
bot.dialog('/', new builder.IntentDialog()
    .matches(/^add/i, '/addTask')
    .matches(/^change/i, '/changeTask')
    .matches(/^delete/i, '/deleteTask')
    .onDefault(builder.DialogAction.send("I'm sorry. I didn't understand.")
);
```

Intent Recognizers

The IntentDialog class can be configured to use cloud based intent recognition services like LUIS through an extensible set of recognizer plugins. Out of the box, Bot Builder comes with a LuisRecognizer that can be used to call a machine learning model you've trained via their web site. You can create a LuisRecognizer that's pointed at your model and then pass that recognizer into your IntentDialog at creation time using the options structure.

```
var recognizer = new builder.LuisRecognizer('<your models url>');
var intents = new builder.IntentDialog({ recognizers: [recognizer] });
bot.dialog('/', intents);

intents.matches('Help', '/help');
```

Intent recognizers return matches as named intents. To match against an intent from a recognizer you pass the name of the intent you want to handle to IntentDialog.matches() as a *string* instead of a *RegExp*. This lets you mix in the matching of regular expressions alongside your cloud based recognition model. To improve performance, regular expressions are always evaluated before cloud based recognizer(s) and an exact match regular expression will avoid calling the cloud based recognizer(s) all together.

You can together multiple LUIS models by passing in an array of recognizers. You can control the order in which the recognizers are evaluated using the recognizeOrder option. By default the recognizers will

be evaluated in parallel and the recognizer returning the intent with the highest score will be matched. You can change the recognize order to series and the recognizers will be evaluated in series. Any recognizer that returns an intent with a score of 1.0 will prevent the recognizers after it from being evaluated.

> **NOTE:** you should avoid adding a matches() handler for LUIS's "None" intent. Add a onDefault() handler instead. The reason for this is that a LUIS model will often return a very high score for the None intent if it doesn't understand the users utterance. In the scenario where you've configured the IntentDialog with multiple recognizers that could cause the None intent to win out over a non-None intent from a different model that had a slightly lower score. Because of this the LuisRecognizer class suppresses the None intent all together. If you explicitly register a handler for "None" it will never be matched. The onDefault() handler, however can achieve the same effect because it essentially gets triggered when all of the models reported a top intent of "None".

Entity Recognition

LUIS can not only identify a users intention given an utterance, it can extract entities from their utterance as well. Any entities recognized in the users utterance will be passed to the intent handler via its args parameter. Bot Builder includes an EntityRecognizer class to simplify working with these entities.

Finding Entities

You can use EntityRecognizer.findEntity() and EntityRecognizer.findAllEntities() to search for entities of a specific type by name.

```
var recognizer = new builder.LuisRecognizer('<your models url>');
var intents = new builder.IntentDialog({ recognizers: [recognizer] });
bot.dialog('/', intents);

intents.matches('AddTask', [
    function (session, args, next) {
        var task = builder.EntityRecognizer.findEntity(args.entities, 'Tas
        if (!task) {
            builder.Prompts.text(session, "What would you like to call th
        } else {
            next({ response: task.entity });
        }
    },
    function (session, results) {
        if (results.response) {
            // ... save task
            session.send("Ok... Added the '%s' task.", results.response);
        } else {
```

```
            session.send("Ok");
        }
    }
]);
```

Resolving Dates & Times

LUIS has a powerful builtin.datetime entity recognizer that can recognize a wide range of relative & absolute dates expressed using natural language. The issue is that when LUIS returns the dates & times it recognized, it does so by returning their component parts. So if the user says “june 5th at 9am” it will return separate entities for the date & time components of the utterance. These entities need to be combined using EntityRecognizer.resolveTime() to get the actual resolved date. This method will try to convert an array of entities a valid JavaScript Date object. If it can’t resolve the entities to a valid Date it will return null.

```
var recognizer = new builder.LuisRecognizer('<your models url>');
var intents = new builder.IntentDialog({ recognizers: [recognizer] });
bot.dialog('/', intents);

intents.matches('SetAlarm', [
    function (session, args, next) {
        var time = builder.EntityRecognizer.resolveTime(args.entities);
        if (!time) {
            builder.Prompts.time(session, 'What time would you like to s
        } else {
            // Saving date as a timestamp between turns as session.dialo
            session.dialogData.timestamp = time.getTime();
```

```
            next();
        }
    },
    function (session, results) {
        var time;
        if (results.response) {
            time = builder.EntityRecognizer.resolveTime([results.respon
        } else if (session.dialogData.timestamp) {
            time = new Date(session.dialogData.timestamp);
        }

        // Set the alarm
        if (time) {

            // .... save alarm

            // Send confirmation to user
            var isAM = time.getHours() < 12;
            session.send('Setting alarm for %d/%d/%d %d:%02d%s',
                time.getMonth() + 1, time.getDate(), time.getFullYear(
                isAM ? time.getHours() : time.getHours() - 12, time.g
        } else {
            session.send('Ok... no problem.');
        }
    }
]);
```

Matching List Items

Bot Builder includes a powerful choice() prompt which lets you present a list of choices to a user for them to pick from. LUIS makes it easy to map a users choice to a named entity but it doesn't do any validation that the user entered a valid choice. You can use EntityRecognizer.findBestMatch() and

EntityRecognizer.findAllMatches() to verify that the user entered a valid choice. These methods are the same methods used by the choice() prompt and offer a lot of flexibility when matching a users utterance to a value in a list.

List items can be matched using a case insensitive exact match so given the list ["Red","Green","Blue"] the user can say "red" to match the "Red" item. Using a partial match where the user says "blu" to match the "Blue" item. Or a reverse partial match where the user says "the green one" to match the "Green" item. Internally the match functions calculate a coverage score when evaluating partial matches. For the "blu" utterance that matched the "Blue" item the coverage score would have been 0.75 and for the "the green one" utterance that matched "green" the coverage score would have been 0.88. The minimum score needed to trigger a match is 0.6 but this can be adjusted for each match.

An IFindMatchResult is returned for each match and contains the entity, index, and score for the list item that was watched.

```
var recognizer = new builder.LuisRecognizer('<your models url>');
var intents = new builder.IntentDialog({ recognizers: [recognizer] });
bot.dialog('/', intents);

intents.matches('DeleteTask', [
    function (session, args, next) {
        // Process optional entities received from LUIS
```

```
        var match;
        var entity = builder.EntityRecognizer.findEntity(args.entities, 'Ta
        if (entity) {
            match = builder.EntityRecognizer.findBestMatch(tasks, enti
        }

        // Prompt for task name
        if (!match) {
            builder.Prompts.choice(session, "Which task would you like t
        } else {
            next({ response: match });
        }
    },
    function (session, results) {
        if (results.response) {
            delete tasks[results.response.entity];
            session.send("Deleted the '%s' task.", results.response.entity)
        } else {
            session.send('Ok... no problem.');
        }
    }
]);
```

onBegin & onDefault Handlers

The IntentDialog lets you register an onBegin handler that will be notified anytime the dialog is first loaded for a conversation and an onDefault handler that will be notified anytime the users utterance failed to match one of the registered patterns.

The onBegin handler is invoked when session.beginDialog() has been called for the dialog and gives the dialog an opportunity to process optional arguments passed in the call to beginDialog(). The handler is passed a `next()` function which should be invoked to continue executing the dialogs default logic.

```
intents.onBegin(function (session, args, next) {
    session.dialogData.name = args.name;
    session.send("Hi %s...", args.name);
    next();
});
```

The onDefault handler is invoked anytime the users utterance doesn't match one of the registered patterns. The handler can be a waterfall, closure, DialogAction, or the ID of a dialog to redirect to.

```
intents.onDefault(builder.DialogAction.send("I'm sorry. I didn't understand
```

The onDefault handler can also be used to manually process intents as it gets passed all of the raw recognizer results via its args parameter.

Localization

Overview

Bot Builder includes a rich localization system for building bots that can communicate with the user in multiple languages. All of your bots prompts can be localized using JSON files stored in your bots directory structure and if you're using a system like LUIS to perform natural language processing you can configure your LuisRecognizer with a separate model for each language your bot supports and the SDK will automatically select the model matching the users preferred locale.

Determining Locale

The first step to localizing your bot for the user is adding the ability to identify the users preferred language. The SDK provides a session.preferredLocale() method to both save and retrieve this preference on a per user basis. Below is short example dialog to prompt the user for their preferred language and then persist their choice:

```
bot.dialog('/localePicker', [
    function (session) {
        // Prompt the user to select their preferred locale
        builder.Prompts.choice(session, "What's your preferred language
    },
    function (session, results) {
        // Update preferred locale
        var locale;
        switch (results.response.entity) {
            case 'English':
                locale = 'en';
            case 'Español':
                locale = 'es';
            case 'Italiano':
                locale = 'it';
                break;
        }
        session.preferredLocale(locale, function (err) {
            if (!err) {
                // Locale files loaded
                session.endDialog("Your preferred language is now %s."
            } else {
                // Problem loading the selected locale
                session.error(err);
            }
        });
```

```
    }
]);
```

Another option is to install a piece of middleware that uses a service like Microsofts Text Analytics API to automatically detect the users language based upon the text of the message they sent:

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

bot.use({
    receive: function (event, next) {
        if (event.text && !event.textLocale) {
            var options = {
                method: 'POST',
                url: 'https://westus.api.cognitive.microsoft.com/text/ana
                body: { documents: [{ id: 'message', text: event.text }
                json: true,
                headers: {
                    'Ocp-Apim-Subscription-Key': '<YOUR API KEY>'
                }
            };
            request(options, function (error, response, body) {
                if (!error && body) {
                    if (body.documents && body.documents.length >
                        var languages = body.documents[0].detected
                        if (languages && languages.length > 0) {
                            event.textLocale = languages[0].iso6391N
                        }
                    }
                }
                next();
            });
```

```
        } else {
            next();
        }
    }
});
```

Calling `session.preferredLocale()` will automatically return the detected language if a user selected locale hasn't been assigned. The exact search order for `preferredLocale()` is:

- Locale saved by calling `session.preferredLocale()`. This value is stored in `session.userData['BotBuilder.Data.PreferredLocale']`.
- Detected locale assigned to `session.message.textLocale`.
- Bots configured default locale.
- English ('en').

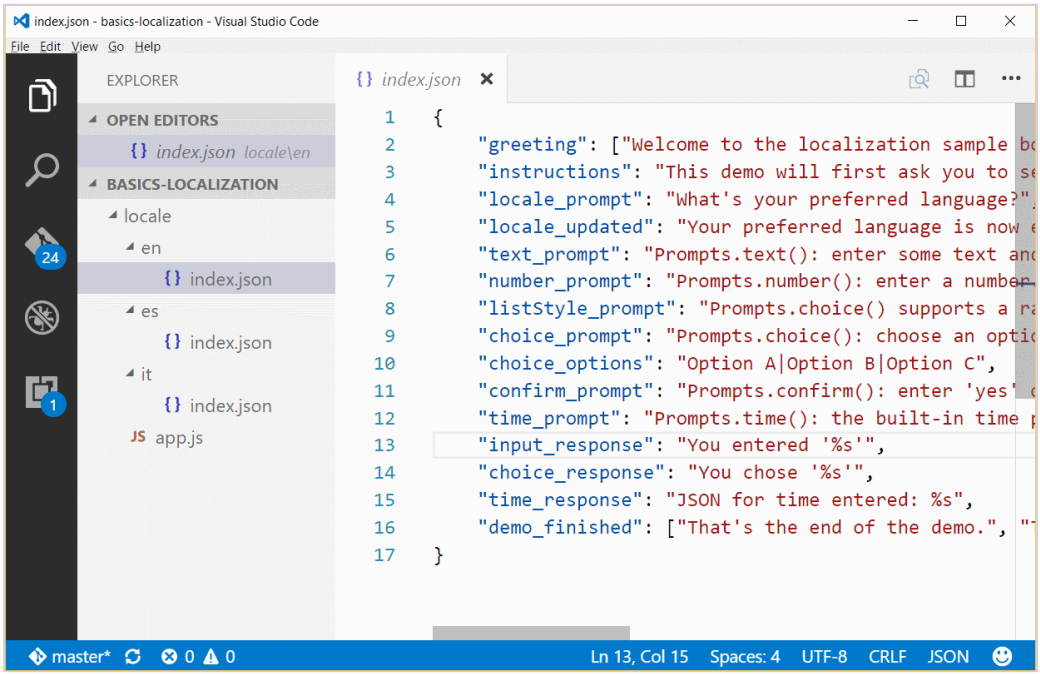
You can configure the bots default locale during construction:

```
var bot = new builder.UniversalBot(connector, {
    localizerSettings: {
        defaultLocale: "es"
    }
});
```

Localizing Prompts

The default localization system for Bot Builder is file based and lets a bot support multiple languages using JSON files stored on disk. By default, the

localization system will search for the bots prompts in the `./locale/<IETF TAG>/index.json` file where `<IETF TAG>` is a valid IETF language tag representing the preferred locale to use the prompts for. Below is a screenshot of the directory structure for a bot that supports three languages, English, Italian, and Spanish:



The structure of the file is straight forward. It's a simple JSON map of message ID's to localized text strings. If the value is an `array` instead of a `string` a prompt will be chosen at random anytime that value is retrieved using session.localizer.gettext(). Returning the localized version of a

message generally happens automatically by simply passing the message ID in a call to session.send() instead of language specific text:

```
bot.dialog("/", [
    function (session) {
        session.send("greeting");
        session.send("instructions");
        session.beginDialog('/localePicker');
    },
    function (session) {
        builder.Prompts.text(session, "text_prompt");
    },
```

Internally, the SDK will call `session.preferredLocale()` to get the users preferred locale and will then use that in a call to `session.localizer.gettext()` to map the message ID to its localized text string. There are times where you may need to manually call the localizer. For instance, the enum values passed to Prompts.choice() are never automatically localized so you may need to manually retrieve a localized list prior to calling the prompt:

```
    var options = session.localizer.gettext(session.preferredLocale(), "ch
    builder.Prompts.choice(session, "choice_prompt", options);
```

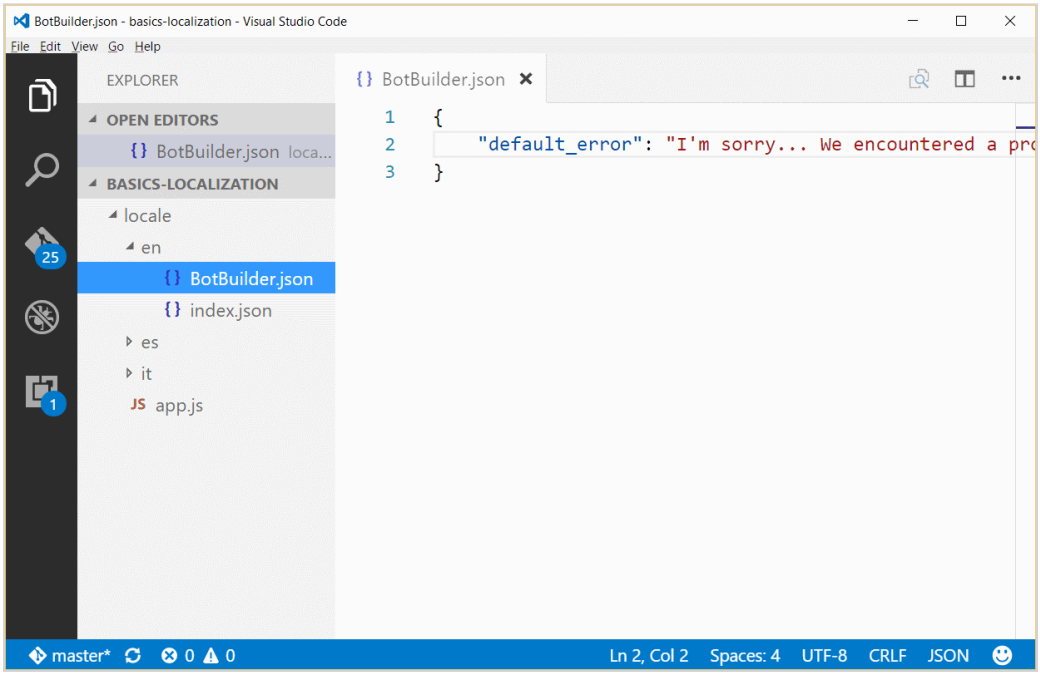
The default localizer will search for a message ID across multiple files and if it can't find an ID (or if no localization files were provided) it will simply

return the text of ID, making the use of localization files transparent and optional. Files are searched in teh following order:

- First the `index.json` file under the locale returned by `session.preferredLocale()` will be searched.
- Next, if the locale included and optional subtag like `en-US` then the root tag of `en` will be searched.
- Finally, the bots configured default locale will be searched.

Namespaced Prompts

The default localizer supports the namespacing of prompts to avoid collisions between message ID's. Name spaced prompts can also be overridden by the bot to essentially let a bot customize or re-skin the prompts from another namespace. Today, you can leverage this capability to customize the SDK's built-in messages, letting you either add support for additional languages or to simply re-word the SDK's current messages. For instance, you can change the SDK's default error message by simply adding a file called `BotBuilder.json` to your bots locale directory and then adding an entry for the `default_error` message ID:



UniversalCallBot

Overview

Skype supports a rich feature called Calling Bots. When enabled, users can place a voice call to your bot and interact with it using Interactive Voice Response (IVR). The Bot Builder for Node.js SDK includes a special Calling SDK which developers can use to add calling features to their chat bot.

Architecturally the Calling SDK is very similar to the Chat SDK. They have similar classes, share common constructs like waterfalls, and you can even use the Chat SDK to send a message to the user you're on a call with. The

two SDK's are designed to run side-by-side but even though they're similar, there are significant differences and you should generally avoid mixing classes from the two libraries.

Installation

To get started either install the Bot Builder module via NPM:

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

Or clone our GitHub repository using Git. This may be preferable over NPM as it will provide you with numerous example code fragments and there's a full `demo-skype-calling` bot you can run:

```
git clone https://github.com/Microsoft/BotBuilder.git
cd BotBuilder/Node
npm install
```

Hello World

The "Hello World" for a calling bot looks very similar to "Hello World" for a chat bot:

```
var restify = require('restify');
var calling = require('botbuilder-calling');
```

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

// Create calling bot
var connector = new calling.CallConnector({
    callbackUrl: 'https://<your host>/api/calls',
    appId: '<your bots app id>',
    appPassword: '<your bots app password>'
});
var bot = new calling.UniversalCallBot(connector);
server.post('/api/calls', connector.listen());

// Add root dialog
bot.dialog('/', function (session) {
    session.send('Watson... come here!');
});
```

The emulator doesn't currently support testing calling bots so you'll need to go through most of the steps need to publish your bot to be able to test it. You can at least run your bot locally during development using a tool like ngrok but you'll need to use a Skype client to interact with the bot.

setup ngrok

Follow the instructions here to setup ngrok on your machine and prep your environment for debugging.

register your bot

Follow the instructions outlined here to register your bot and enable the skype channel. You will need provide a messaging endpoint when you register your bot in the developer portal and we typically recommend that you pair your calling bot with a chat bot so the chat bot's endpoint is what you would normally put in that field. If you're only registering a calling bot you can simply paste your calling endpoint into that field.

To enable the actual calling feature you'll need to go into the skype channel for your bot and turn on the calling feature. You'll then be provided with a field to copy your calling endpoint into. Make sure you use the https ngrok link for the host portion of your calling endpoint.

configure your bot

During the registration of your bot you'll be assigned an app ID & password which you should paste into the connector settings for your hello world bot. You'll also need to take your full calling link and paste that in for the callbackUrl setting.

add bot to contacts

On your bots registration page in the developer portal you'll see a `add to skype` button next to your bots skype channel. Click this link to get you bot

added to your contact list in skype. Once you do that you (and anyone you give the join link to) will be able to communicate with the bot.

test your bot

You can test your bot using a skype client. You should notice the call icon light up when you click on your bots contact entry (you may have to search for the bot to see it.) It can take a few minutes for the call icon to light up if you've added calling to an existing bot.

If you press the call button it should dial your bot and you should hear it speak "Watson... come here!" and then hang up.

Calling Basics

The UniversalCallBot and CallConnector classes let you author a calling bot in much the same way you would a chat bot. You add dialogs to your bot that are essentially identical to chat dialogs. You can add waterfalls to your bot and the steps will get a session object just like in chat but this session object is a CallSession class which contains added answer(), hangup(), and reject() methods for managing the current call. In general, you don't need to worry about these call control methods though as the CallSession has logic to automatically manage the call for you. The session will automatically answer the call if you take an action like sending a message

or calling a built-in prompt. It will also automatically hangup/reject the call if you call endConversation() or it detects that you've stopped asking the caller questions (you didn't call a built-in prompt.)

Another difference between calling and chat bots is that while chat bots typically send messages, cards, and keyboards to a user a calling bot deals in Actions and Outcomes. Skype calling bots are required to create workflows that are comprised of one or more actions. This is another thing that in practice you don't have to worry too much about as the Bot Builder calling SDK will manage most of this for you. The CallSession.send() method lets you pass either actions or strings which it will turn into PlayPromptActions. The session contains auto batching logic to combine multiple actions into a single workflow that's submitted to the calling service so you can safely call send() multiple times. And you should rely on the SDK's built-in prompts to collect input from the user as they process all of the outcomes.

Prompts

Collecting Input

Bot Builder comes with a number of built-in prompts that can be used to collect input from a user.

Prompt Type	Description
Prompts.choice	Asks the user to choose from a list of choices.
Prompts.digits	Asks the user to enter a sequence of digits.
Prompts.confirm	Asks the user to confirm an action.
Prompts.record	Asks the record a message.
Prompts.action	Sends a raw action to the calling service and lets the bot manually process its outcome.

These built-in prompts are implemented as a Dialog so they'll return the users response through a call to session.endDialogWithresult(). Any DialogHandler can receive the result of a dialog but waterfalls tend to be the simplest way to handle a prompt result.

Prompts return to the caller an IPromptResult. The users response will be contained in the results.response field and may be null should the user fail to input a proper response.

Prompts.choice()

The Prompts.choice() method asks the user to pick an option from a list. The users response will be returned as an IPromptChoiceResult. The list of choices is passed to the prompt as an array of IRecognitionChoice objects.

You can configure the prompt to use speech recognition to recognize the callers choice:

```
calling.Prompts.choice(session, "Which department? support, billing, or cl
    { name: 'support', speechVariation: ['support', 'customer service'] },
    { name: 'billing', speechVariation: ['billing'] },
    { name: 'claims', speechVariation: ['claims'] }
]);
```

Or DTMF input using Skypes diling pad:

```
calling.Prompts.choice(session, "Which department? Press 1 for support, 2
    { name: 'support', dtmfVariation: '1' },
    { name: 'billing', dtmfVariation: '2' },
    { name: 'claims', dtmfVariation: '3' }
]);
```

Or both speech recognition and DTMF:

```
bot.dialog('/departmentMenu', [
    function (session) {
        calling.Prompts.choice(session, "Which department? Press 1 for s
            { name: 'support', dtmfVariation: '1', speechVariation: ['sup
            { name: 'billing', dtmfVariation: '2', speechVariation: ['billir
            { name: 'claims', dtmfVariation: '3', speechVariation: ['clain
            { name: '(back)', dtmfVariation: '*', speechVariation: ['back
        ]);
    },
    function (session, results) {
        if (results.response !== '(back)') {
            session.beginDialog('/' + results.response.entity + 'Menu');
        } else {
            session.endDialog();
        }
    },
    function (session) {
        // Loop menu
        session.replaceDialog('/departmentMenu');
    }
]);
```

The users choice is returned as an IFindMatchResult similar to chat bots and the choices name will be assigned to the response.entity property.

Prompts.digits()

The Prompts.digits() method asks the user to enter a sequence of digits followed by an optional stop tone. The users response will be returned as an IPromptDigitsResult.

```
calling.Prompts.digits(session, "Please enter your account number followe
```

Prompts.confirm()

The Prompts.confirm() method asks the user to confirm some action. This prompt builds on the choices prompt by calling it with a standard set of yes & no choices. The user can reply by saying a range of responses or they can press 1 for yes or 2 for no. The users response will be returned as an IPromptConfirmResult.

```
calling.Prompts.confirm(session, "Would you like to end the call?");
```

Prompts.record()

The Prompts.record() method asks the user to record a message. This prompt builds on the choices prompt by calling it with a standard set of yes

& no choices. The recorded message will be returned as an IPromptRecordResult and the recorded audio will be available off that object as a *{Buffer}*.

```
calling.Prompts.record(session, "Please leave a message after the beep.");
```

Prompts.action()

The Prompts.action() method lets you send the calling service a raw action object. The outcome will be returned as an IPromptActionResult for manual processing by your bot.

In general, you shouldn't ever need to call this prompt but one scenario where you might is if you wanted to send a playPrompt action that plays a file to the user and you'd like to keep the call active so you can take another action once that completes. The normal session.send() method you'd use to play a file will automatically end the call if that playPrompt action isn't followed by a recognize or record action so this gives you a way of dynamically chaining play prompts together. You might do this if you want to play the user hold music or silence while you periodically check for some long running operation to complete.

Release Notes

Bot Builder for Node.js is targeted at Node.js developers creating new bots from scratch. By building your bot using the Bot Builder framework you can easily adapt it to run on nearly any communication platform. This gives your bot the flexibility to be wherever your users are.

- Bot Builder for Node.js Reference
- Bot Builder on GitHub

Install

Get the latest version of Bot Builder using npm.

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

Release Notes

The framework is still in preview mode so developers should expect breaking changes in future versions of the framework. A list of current

issues can be found on our GitHub Repository.

v3.7.0

Changes

- Added prompt locale files for PT (Portuguese) language.
- Added new Intent Forwarding feature that lets middleware forward intents to the message router.
- Simplified the build process.
- Fixed a bug in Session.validateDialogStack() that was causing it to always succeed.
- Added a ducktyping check to the RegExpRecognizer to fix an issue on Node-RED.
- Removed the deprecated node-uuid module that we weren't even using anymore.
- Added logic to Prompts to validate that the session object is passed in and throw a more meaningful error message when it's not.
- Updated package.json version.

v3.6.0

Changes

- Fixed a bug causing global recognizers to be run twice..
- Added a new UniversalBot.loadSession() method to let you load a session object for an address.
- Added a new ChatConnector.onInvoke() callback for use with the protocols new invoke activity type..
- Updated LKG build and package.json version.

v3.5.4

Bug Fixes

- Fix bug where service url is modified in Teams, breaking even basic bots in Teams

v3.5.3

New Features

- All new customizable routing system.
- UniversalBot now derives from Library so you can easily aggregate child libraries and bots under a single parent bot.
- Added support for registering recognizers globally.
- Added new `Dialog.triggerAction()` for specifying rules that auto launch dialogs based on a users utterance.
- All actions are now customizable.
- Added `confirmPrompt` to `Dialog.cancelAction()`.
- New `Session` methods for saving and replacing dialog stacks.
- Expanded IRecognizeContext object to be just a read-only version of the session.
- Simplified creation of basic send/receive bot: Added ability to pass default dialog to `UniversalBot` constructor and deprecated passing of additional settings.
- Added new `UnoversalBot.onDisambiguateRoute` hook.
- Added new `MediaCard` classes for building media card attachments.
- Added support for French to the built-in prompts.
- Added support for cloning a library or bot.

Bug Fixes

- Updated `LuisRecognizer` to support v2 URL's.
- Fixed issue with localization namespace not being found on macs.
- Fixed an issue with the session always assuming '*' as the root namespace.
- Chat connector fix for case sensitive Authorization header.
- Several localization related fixes uncovered by new unit tests.
- Fix `EntityRecognizer` is very relaxed when parsing affirmative/negative.

- Fix bug in url joining where the last part of the service uri gets lost due to `url.resolve` behavior.
- Fixes related to the move to TypeScript 2.1.

Other Changes

- Upgrade Node.js Bot Builder to v3.1 auth.
- Added passing of `user-agent` header in `ChatConnector` calls.
- Added several new feature specific examples.
- Updated all examples to use new 3.5 features.
- Added several new unit tests.
- Updated `Channel.ts` to reflect latest capabilities.

v3.4.2

- Fixed an exception being raised for bots without a default locale.
- Fixed an inadvertent rename of `Library.name` to `Library.namespace` in the libraries typescript definition file.
- Updated LKG build and package.json version.

v3.4.0

- Fixed a bug where path to localization files was being lowercased.
- Added support for localizing prompts on a per/library basis. Each library can now have it's own /locale/ folder and prompts which can be overridden by the bot.
- Removed a content.message.text guard from IntentDialog. Now you can recognize based on attachments as well as text.
- Updated LKG build and package.json version.

v3.3.3

- Removed requirement for IDialogResult.resumed.
- Removed ability to pass in a custom localizer. The new localizer system should be used instead.

- Changed a warning that was getting emitted from localizer to a debug statement.
- Fixed a bug where the localizers path wasn't defaulting to "./locale/"
- Updated LKG build and package.json version.

v3.3.2

- Fixed an issue with ordinal parsing in EntityRecognizer.parseNumber().
- Fixed an issue with ListStyle.inline being used for Prompts.confirm() on text based channels.
- Fixed a bug with localized Prompts.confirm() options not being recognized.
- Updated LKG build and package.json version.

v3.3.1

- Added new locals for Spanish, Italian, and Chinese.
- Fixed an issue with the preferred local not being passed to LuisRecognizer class.
- Fixed an issue with localizationNamespace param not being passed for prompts.
- Fixed an issue with DefaultLocalizer class assuming it would always have a done callback.
- Cleaned up passing of localizerSettings from bot to DefaultLocalizer.
- Updated typescript definitions and docs.
- Updated LKG build and package.json version.

v3.3.0

- Added new prompt localization system.
- Fixed an issue with callbacks passed to UniversalBot.send() not being called.
- Added missing Keyboard class export.
- Fixed a missing callback in Session.sendBatch().
- Updated Session.sendTyping() to send the current batch immediately.

- Fixed waterfall step count that's logged to console.
- Took a PR to prevent a server crash if request body has nothing.
- Added a new IntentDialog.recognizer() method .
- Added a new Session.preferredLocale() method.
- Fixed a bug where a late bound connector wasn't getting used as storage.
- Updated reference docs.
- Updated LKG build and package.json version.

v3.2.3

- Moved setting of sessionState.lastAccess to happen after middleware runs. This lets you write middleware that expires old sessions.
- Fixed a couple of issues with proactive conversations not working. Also should fix issues with proactive conversations for groups not working.
- Updated LKG build and package.json version.

v3.2.2

- Fixed undesired forward resume status in waterfall step.
- Updated unit tests. Removed old deprecated ones.
- Added missing export for RecognizeMode and fixed a type-o with RecognizeOrder export.
- Updated score returned from LuisRecognizer for 'none' intent. It's now a score of 0.1 so it will trigger but won't stomp on other models.
- Updated botbuilder.d.ts file bundled with npm.
- Updated LKG build and package.json version.

v3.2.1

Breaking Changes

These changes will impact a small number of bots.

- Updated IRecognizerActionResult.matched and IIntentRecognizerResult.matched to return all the matched results as an array versus just the text that matched. These changes shouldn't effect many people as its likely you were already re-evaluating the matched expression if you needed the capture data.
- Added a new RecognizeMode for IntentDialogs which solves in issue where launching an IntentDialog as a sub dialog would cause it to immediately process the last utterance. This should only be a breaking change if you are using multiple IntentDialogs to scope down from a general model to a more specialized model. In that case you'll want to create your specialized IntentDialogs with a `recognizeMode: builder.RecognizeMode.onBegin`.

Other Changes

- Added support for ordinal words to EntityRecognizer.parseNumber().
- Added support for Facebook Quick Replies.
- Added new Session.sendTyping() function.
- Added optional callback param to sendBatch().
- Added IntentDialog.matchesAny() method that takes an array of intents to match.
- Fixed an issue for middleware that was causing session.beginDialog() calls to always run in the current dialogs library content which was often a system prompt. Now middleware assumes the default library context.
- Added support for gzip your bot data.
- Fixed an issue where storing too much bot data would get your bot into a stuck state.
- Updated LKG build and package.json version.

v3.1.0

- Removed try catches that were causing errors to be ate. Added logic to dump stack trace because node isn't always dumping on uncaught exceptions.

- Added NODE_DEBUG logging switch to enable logging of channels other than the emulator.
- Implemented actions.
- Updated reference docs.
- Added keyboard concept and updated Prompts.choice() to use keyboards.
- Added basic support for Facebook quick_replies using keyboards.
- Fixed auth issues around ChatConnector.
- Added new CardAction.dialogAction() type.
- Removed 'cancel' checks from Prompts.
- Updated prompts to not by default exit out after too many retries.
- Added Session.sendTyping() method.
- Updated LKG build and package.json version.

v3.0.1

- Fixed an issue with channelData being sent for messages without channel data.
- Fixed an issue where we weren't reporting a lot of errors.
- Added logic to properly verify the bots identity when called from the emulator.
- Added console logging to report failures around security with recommendations.
- Added a new tracing system that logs the bots session & dialog activity to the console for the emulator.
- Updated LKG build and package.json version.

v3.0.0

- See What's New for a relatively complete list of changes.

v1.0.0

Breaking Changes

These changes will impact some bots.

- Simple closure based handlers are now single step waterfalls.
- If a dialog steps past the end of a waterfall the dialog is automatically ended.

Other Changes

- Exposed SimpleDialog class from both module & docs.
- DialogAction.validatedPrompt() now returns a Dialog which makes it more strongly typed.
- Fixed an issue with the LuisDialog on() & onDefault() handlers eating exceptions.
- Fixed issue with telegram not showing buttons on re-prompt.
- Updated LKG build and package.json version.

v0.11.1

- Fixed a bug causing multiple messages to get rejected by the live servers.
- Updated LKG build and package.json version.

v0.11.0

- Added Prompts.attachment() method.
- Updated Message.randomPrompt() to take a string or an array.
- Updated Session to clone() raw IMessage entries before sending (fixes a serialization bug)
- Fixed issue where configured BotConnectorBot endpoint wasn't getting used in production.
- Tweaked the way built-in dialogs get registered.
- Added support for showing Prompts.confirm() using buttons.
- Improved the way re-prompting works.
- Created type specific default re-prompts.

- Minor tweak to the way the emulators callback URL is calculated.
- Updated LKG build and package.json version.

v0.10.2

- Fixed a bug in CommandDialog preventing onDefault() handlers from resuming properly.
- Updated LKG build and package.json version.

v0.10.1

- Fixed a bug preventing BotConnectorBot configured greeting messages from being delivered.
- Fixed a couple of issues with Prompts.choice() when not using ListStyle.auto.
- Updated LKG build and package.json version.

v0.10.0

- Added logic to automatically detect messages from the emulator. This removes the need to manually set an environment variable to configure talking to the emulator.
- Added support for new Action attachment type (buttons.)
- Exposed static LuisDialog.recognize() method. Can be used to manually call a LUIS model.
- Added support to Prompts.choice() to render choices as buttons using ListStyle.button.
- Added new ListStyle.auto option to Prompts.choice() which automatically selects the appropriate rendering option based on the channel and number of choices. This is the new default style.
- Added support to all Prompts for passing in an array of prompt & re-prompt strings. A prompt will be selected at random.
- Added support to all Prompts for passing in an IMessage. This lets you specify prompts containing images and other future attachment types.
- Updated LKG build and package.json version.

v0.9.2

- Fixed an undefined bug in Message.setText()
- Updated LKG build and package.json version.

v0.9.1

- Changed Math.round to Math.floor to fetch random array element
- Updated LKG build and package.json version.

v0.9.0

Breaking Changes

None of these changes are likely to effect anyone but they could so here are the ones that may break things:

- Updated arguments passed to BotConnectorBot.listen().
- Renamed ISessionArgs to ISessionOptions and also renamed Session.args to Session.options.
- Made Session.createMessage() private. It doesn't need to be public now that we have new Message builder class.
- Changed EntityRecognizer.parseNumber() to return Number.NaN instead of undefined when a number isn't recognized.

Other Changes

- Significant improvements to the Reference Docs.
- Fixed a couple of bugs related to missing intents coming back from LUIS.
- Fixed a deep copy bug in MemoryStorage class. I now use JSON.stringify() & JSON.parse() to ensure a deep copy.

- Made dialogId passed to Session.reset() optional.
- Updated Message.setText() to support passing an array of prompts that will be chosen at random.
- Added methods to Message class to simplify building complex multi-part and randomized prompts.
- BotConnectorBot changes needed to support continueDialog() method that's in development.
- Fixed a typo in the import of Consts.ts for Mac builds.
- Updated LKG build and package.json version.

v0.8.0

- Added minSendDelay option that slows down the rate at which a bot sends replies to the user. The default is 1 sec but can be set using an option passed to the bot. See TextBot.js unit test for an example of that.
- Added support to SlackBot for sending an isTyping indicator. This goes along with the message slow down.
- Added a new Message builder class to simplify returning messages with attachments. See the send-attachment.js test in TestBot for an example of using it.
- Added a new DialogAction.validatedPrompt() method to simplify creating custom prompts with validation logic. See basics-validatedPrompt example for a sample of how to use it.
- SlackBot didn't support returning image attachments so I added that and fixed a couple of other issues with the SlackBot.
- Updated the LKG build, unit tests, and package.json version.

v0.7.2

- Fixed bugs preventing BotConnectorBot originated messages from working. Also resolved issues with sending multiple messages from a bot.
- Fixed bugs preventing SlackBot originated messages from working.
- Updated LKG build and package.json version.

v0.7.1

- Fixed a critical bug in Session.endDialog() that was causing Session.dialogData to get corrupted.
- Updated LKG build and package.json version.

v0.7.0

- Making Node CommandDialog robust against undefined matched group.
- Added the ability to send a message as part of ending a dialog..
- Updated LKG build and package.json version.

v0.6.5

- Fixed bad regular expressions used by Prompts.confirm() and adding missing unit tests for Prompts.confirm().
- Updated LKG build and package.json version.

v0.6.4

- LUIS changed their scheme for the prebuilt datetime entity and are no longer returning a resolution_type which caused issues for EntityRecognizer.resolveTime(). I know use either resolution_type or entity.type.
- Updated LKG build and package.json version.

v0.6.3

- LUIS changed their schema for the pre-built Cortana app which caused the basics.naturalLanguage example to stop working. This build fixes that issue.
- Updated LKG build and package.json version.

v0.6.2

- Fixed an issue where Session.endDialog() was eating error messages when a dialog throws an exception. Now exceptions result in the 'error' event being emitted as expected.
- Updated BotConnectorBot.verifyBotFramework() to only verify authorization headers over HTTPS.
- Removed some dead code from LuisDialog.ts.
- Updated LKG build and package.json version.

v0.6.1

- Fixed an issue with SlackBot & SkypeBot escapeText() and unescapeText() methods not doing a global replace.
- Changed the URL that the BotConnectorBot sends outgoing bot originated messages to. We had an old server link.
- Updated LKG build and package.json version.