Microsoft

Microsoft Azure 自習書シリーズ

Cognitive Services と Bot Service で作る 業務アプリケーション

この自習書では、Microsoft が提供する Cognitive Services と Bot Service を利用して、複数の機能を持つ Bot アプリケーションを作成する流れを、ハンズオン形式で学習します。

発行日：2018 年 4 月 1 日

更新履歴

版数	発行日	更新履歴
第 1 版	2018 年 4 月 1 日	初版発行

このドキュメントは瀬尾ソフト代表 瀬尾佳孝氏（y.seo@seosoft.jp）の多大なるご協力により完成しました。ここにこれまでのご協力への感謝を記します。

目次

1. はじめに

この自習書では、Microsoft が提供する Cognitive Services と Azure Bot Service とを利用して Bot アプリケーションを作成する流れを、ハンズオン形式で学習します。

この自習書では、以下のサービスとソフトウェアを使用します。

- サービス
  - Azure Bot Service
  - Cognitive Services
    - Language Understanding (LUIS)
    - QnA Maker API
    - Bing Web Search API
  - Azure Table Storage
- ソフトウェア
  - Visual Studio 2017 Update 6 以降 (Community Edition 利用可能)
  - Bot Framework Emulator

作業 PC の OS としては Windows 10 を想定しています。

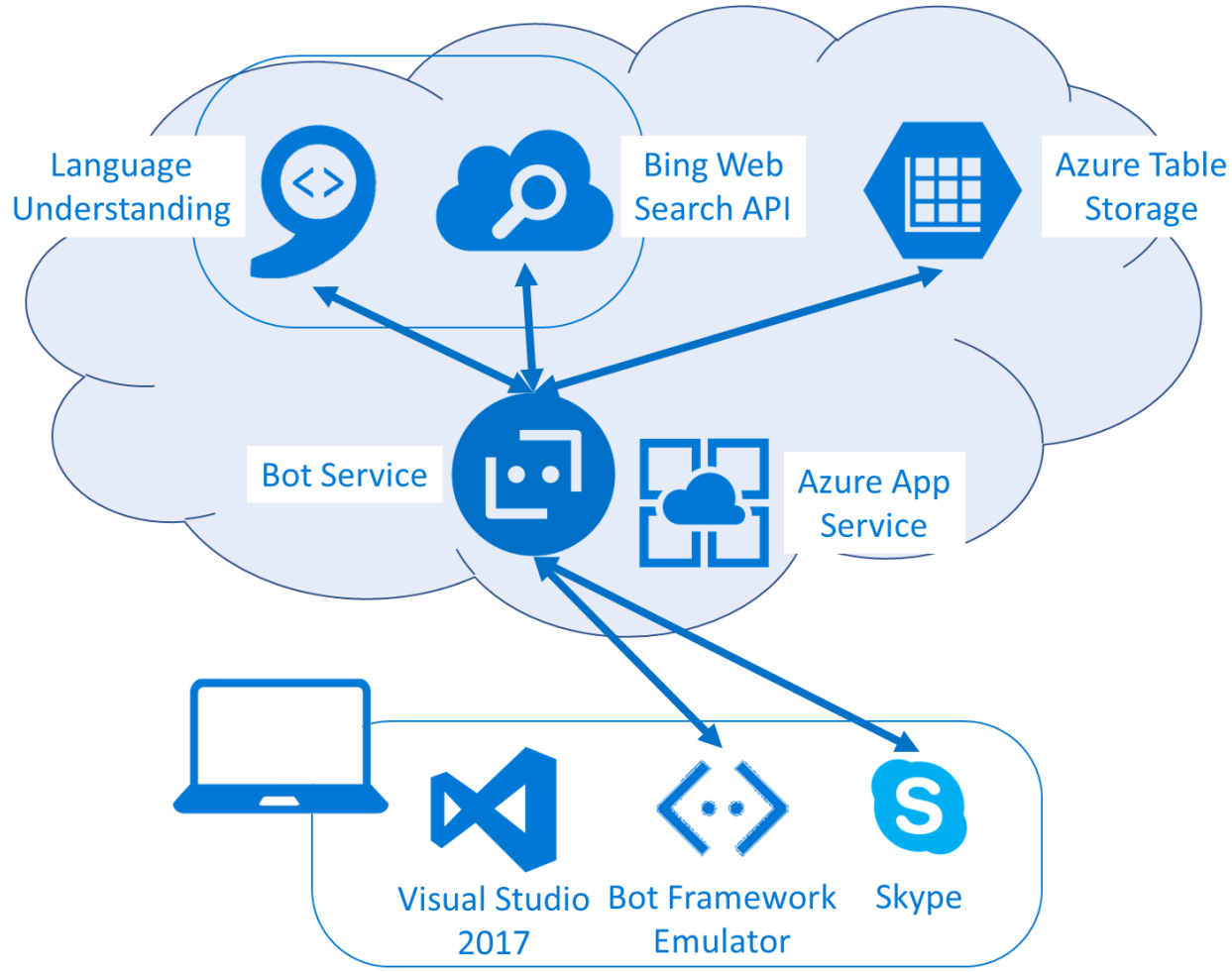
STEP 1. 実習環境の準備

このステップでは、実習に必要な環境の準備について説明します。

- ✓ 開発する Bot アプリケーション
- ✓ Microsoft Azure サブスクリプションの準備
- ✓ Visual Studio 2017 のインストール
- ✓ Bot Framework Emulator のインストール

2. 自習書で開発する Bot アプリケーション

この自習書で開発する Bot アプリケーションは以下の構成です。



- Bot アプリケーションの機能

この自習書で作成する Bot アプリケーションは、チャットクライアントを通してユーザーの入力に応答したり、指定された情報を記憶したりする機能を持ちます。

- 利用者の名前を含めて挨拶をする
- Q&A に答える
- Q&A で適切なものが見つからない場合に Web 検索する
- 利用者のタスクを登録する
- 利用者のタスクの有無や内容を答える

- Bot アプリケーション構築と開発フレームワーク

この自習書では、Bot アプリケーションの構築に Azure Bot Service の Web App Bot を利用します。
また Bot アプリケーション開発のフレームワークとしては BotBuilder SDK for .NET を使用します。

- 利用する API

最初に挙げた応答を実現するために、

- Language Understanding (LUIS)
- QnA Maker API
- Bing Web Search API

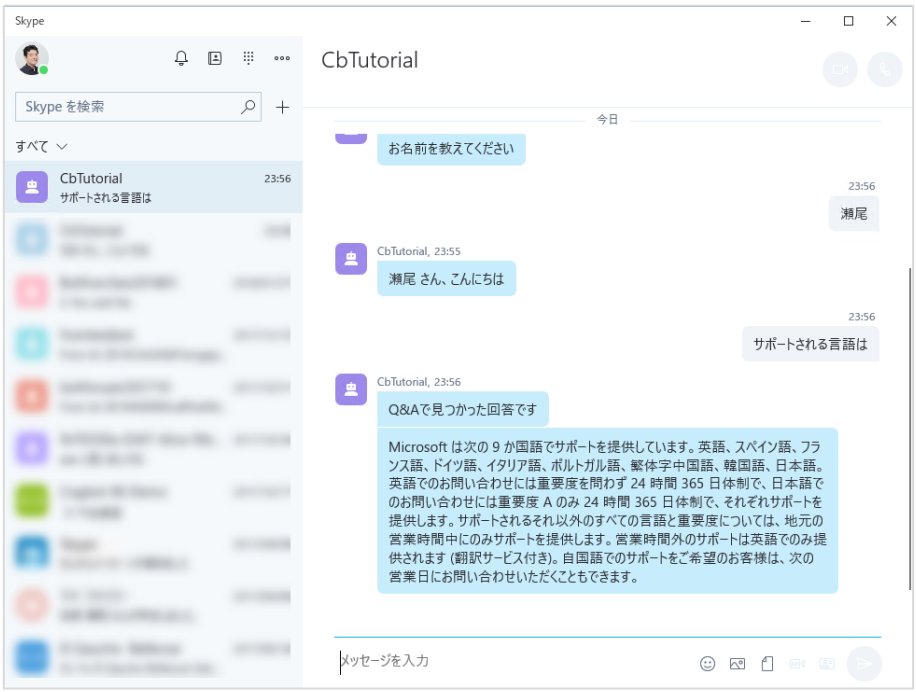
の API を利用します。

- データストア

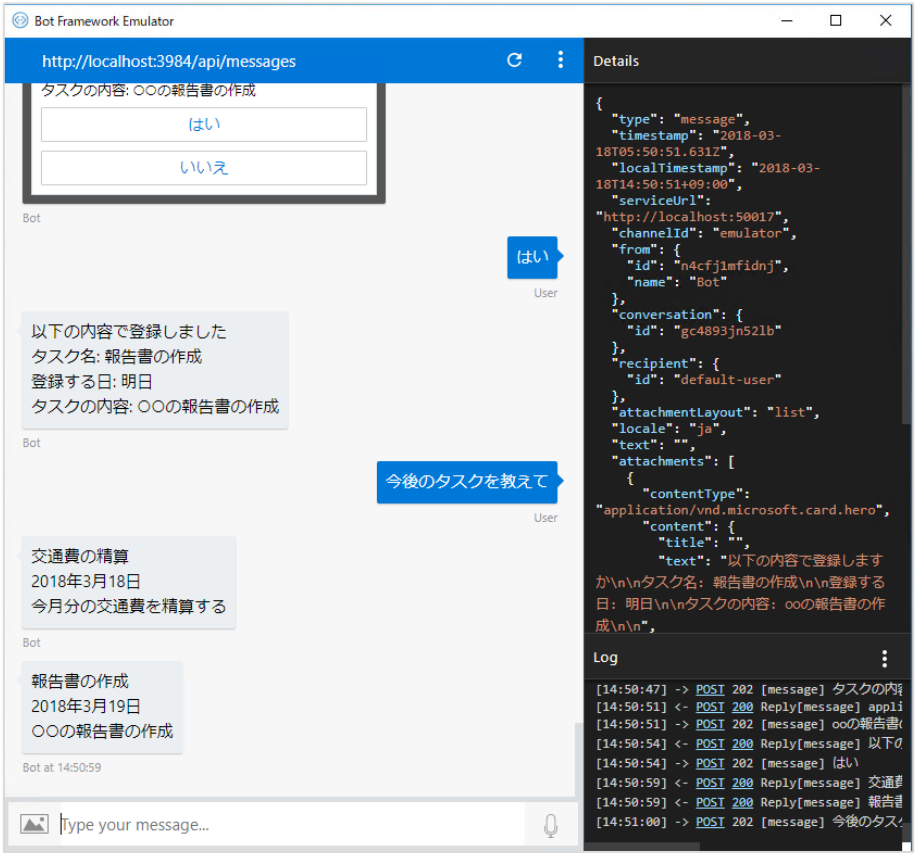
ユーザー情報を保存するためのデータストアとして Azure テーブルストレージを使用します。

- チャットクライアント

Bot Framework は多数のチャットクライアントに対応していますが、この自習書では Bot Framework Emulator (デバッグ用途)、Web Chat、Skype、Microsoft Teams を利用する手順を紹介します。
Microsoft Teams については Office 365 サブスクリプションをお持ちの場合のみ実習可能です。




Cogbot Tutorial - Microsoft
CbTutorial | Microsoft T
コマンドの検索または入力
最近使ったアイテム
CbTutorial
Microsoft は次の 9 か国語でサポ...
おすすめ
チーム サイト
CbTutorial
会話
瀬尾
CbTutorial 23:42
瀬尾 さん、こんにちは
23:42
サポートされる言語は
CbTutorial 23:42
Q&Aで見つかった回答です
Microsoft は次の 9 か国語でサポートを提供しています。英語、スペイン語、フランス語、ドイツ語、イタリア語、ポルトガル語、繁体字中国語、韓国語、日本語。
英語でのお問い合わせには重要度を問わず 24 時間 365 日体制で、日本語でのお問い合わせには重要度 A のみ 24 時間 365 日体制で、それぞれサポートを提供します。サポートされるそれ以外のすべての言語と重要度については、地元の営業時間中にのみサポートを提供します。営業時
ここに質問を入力してください


Bot Framework Emulator
http://localhost:3984/api/messages
タスクの内容: ○○の報告書の作成
はい
いいえ
Bot
はい
User
以下の内容で登録しました
タスク名: 報告書の作成
登録する日: 明日
タスクの内容: ○○の報告書の作成
Bot
今後のタスクを教えて
User
交通費の精算
2018年3月18日
今月分の交通費を精算する
Bot
報告書の作成
2018年3月19日
○○の報告書の作成
Bot at 14:50:59
Type your message...
Details
{
"type": "message",
"timestamp": "2018-03-18T05:50:51.631Z",
"localTimestamp": "2018-03-18T14:50:51+09:00",
"serviceUrl": "http://localhost:50017",
"channelId": "emulator",
"from": {
"id": "n4cfj1mfidnj",
"name": "Bot"
},
"conversation": {
"id": "gc4893jn521b"
},
"recipient": {
"id": "default-user"
},
"attachmentLayout": "list",
"locale": "ja",
"text": "",
"attachments": [
{
"contentType": "application/vnd.microsoft.card.hero",
"content": {
"title": "",
"text": "以下の内容で登録しますか\n\nタスク名: 報告書の作成\n\n登録する日: 明日\n\nタスクの内容: ○○の報告書の作成\n\n",
Log
[14:50:47] -> POST 202 [message] タスクの内
[14:50:51] <- POST 200 Reply[message] appli
[14:50:51] -> POST 202 [message] ○○の報告書
[14:50:54] <- POST 200 Reply[message] 以下の
[14:50:54] -> POST 202 [message] はい
[14:50:59] <- POST 200 Reply[message] 交通費
[14:50:59] <- POST 200 Reply[message] 報告書
[14:51:00] -> POST 202 [message] 今後のタス

3. Microsoft Azure サブスクリプションの準備

この自習書を進めるには Microsoft Azure サブスクリプションの作成が必要です。
すでに Microsoft Azure サブスクリプションをお持ちの場合は、このステップの実施は不要です。

1. Microsoft アカウントの作成（※お持ちでない場合のみ）

Microsoft Azure サブスクリプションを契約するには、Microsoft アカウントが必要です。
お持ちでない場合は、

http://www.microsoft.com/ja-jp/msaccount/signup/default.aspx

を Web ブラウザーで開き、新しく Microsoft アカウントを作成します。

2. Microsoft Azure サブスクリプションの作成

Microsoft Azure サブスクリプションを作成するには、

https://azure.microsoft.com/ja-jp/free/

を Web ブラウザーで開き、手順に従って Microsoft Azure サブスクリプションを作成します。

ワンポイント
Microsoft Azure サブスクリプション作成時には、 ● 確認コードを音声または SMS で受け取るための携帯電話 ● 身元確認のためのクレジットカード が必要です。

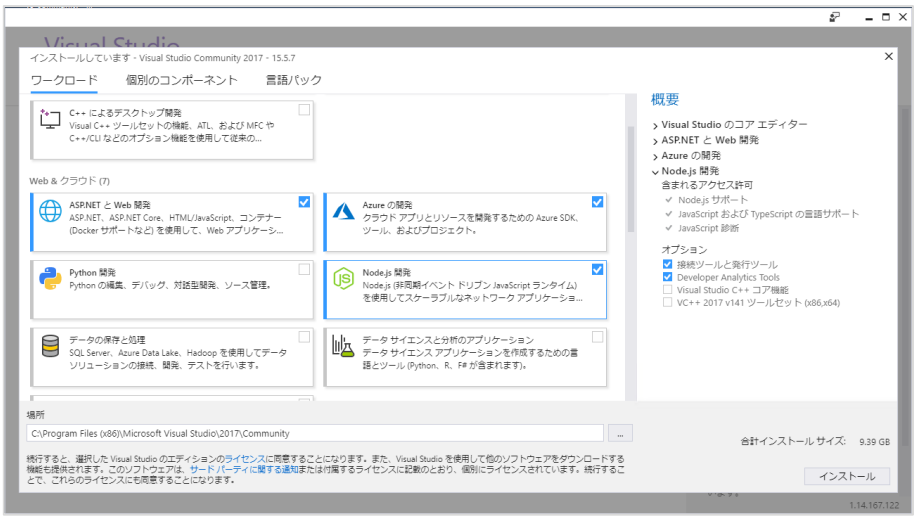
4. Visual Studio 2017 のインストール

この自習書では開発ツールとして Visual Studio 2017 を使用します。Visual Studio 2017 は 2018 年 3 月現在、Update 6 が公開されています。
Community （個人利用では無償）、有償の Professional、Enterprise のいずれのエディションも利用可能です。

Visual Studio 2017 をお持ちでない方は、https://www.visualstudio.com/ja/vs/ でインストーラーをダウンロードしてください。その際には、お持ちのライセンスに応じて利用可能なエディションのインストーラーを選択してください。

インストーラーを起動したら、[ワークロード] で以下を選択してからインストールを実行します。
以下が含まれていれば、他のワークロードも選択されていても問題ありません。

- ASP.NET と Web 開発
- Azure の開発

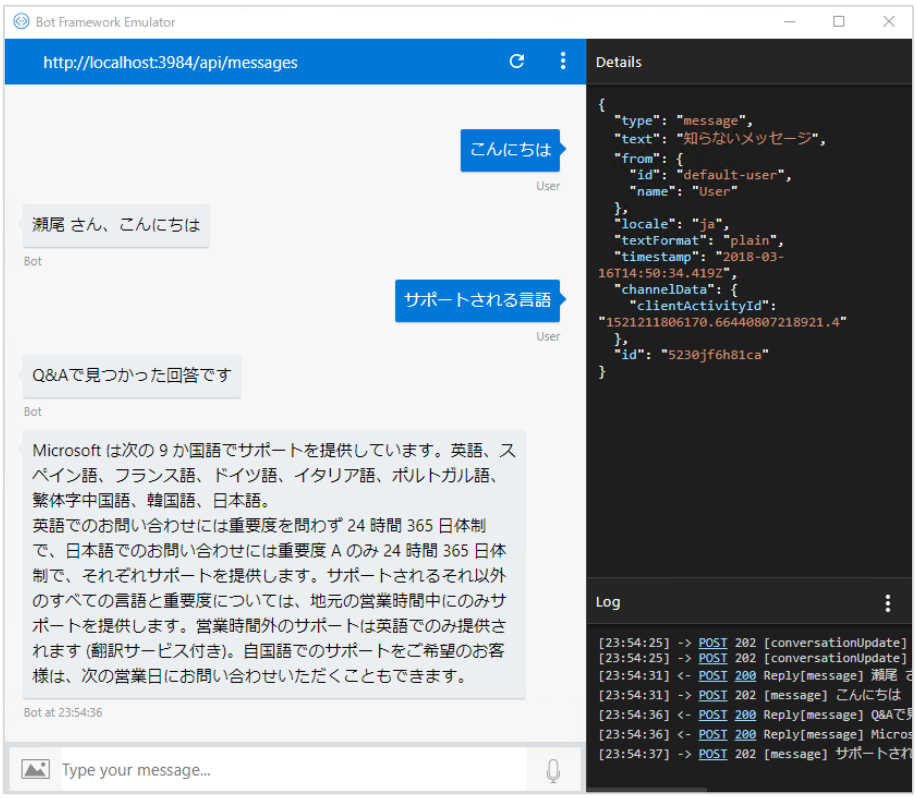


ワンポイント

この自習書の範囲では、[Node.js 開発] ワークロードはインストール不要です。
Function Bot を作成してローカルデバッグする場合には [Node.js 開発] が必要になります。今後のステップアップのためにインストールしておいでもいいでしょう。

5. Bot Framework Emulator のインストール

この自習書では、Visual Studio 2017 以外に Bot Framework Emulator を使用します。



Bot Framework Emulator はチャットクライアントの一つです。
Bot Framework Emulator の特徴は、通常のチャット画面だけではなく、送受信された REST データやログが表示可能であることです。
また 他のチャットクライアントは Bot アプリケーションに接続するには Bot Service のサイトで チャンネル登録を行う必要がありますが、Bot Framework Emulator ではチャンネル登録の必要はありません。
このため、Bot アプリケーションのデバッグ時には Bot Framework Emulator を使用するのが便利です。

Bot Framework Emulator のインストーラーは、

https://github.com/Microsoft/BotFramework-Emulator/releases

からダウンロードできます。
Windows 用インストーラーをダウンロードして、作業 PC にインストールしてください。

STEP 2. Cognitive Services と Bot Service

このステップでは、マイクロソフトの AI 技術である、Cognitive Services と Bot Service について説明します。

- ✓ 認知機能
- ✓ Cognitive Services
- ✓ Bot アプリケーション
- ✓ Azure Bot Service

6. 認知機能

IT の分野での「認知機能」とは何を指すのか、ここで概要を理解しておきましょう。

IT での認知機能とは、人間が持っている能力、

- 見る
- 聞く
- 話す

などに加えて、

- 理解する
- 考える

の能力をコンピューターで実現することです。

明るさや音の大きさを数値で表すようなセンサー技術は、すでに一般化しています。
それに対して、「見ているものは何か」、「何と言ったか」を IT で知覚することは難しいものでした。

この数年で、ハードウェア性能（特に GPU の発展）が向上し、ソフトウェア技術（ツールやライブラリの充実）が向上したことで、コンピューターで認知機能を実現するためのハードルが下がってきました。

最近では、学習済みの認知機能がサービスとして提供されています。
Microsoft の Cognitive Services もそういったサービスの一つです。

7. Cognitive Services

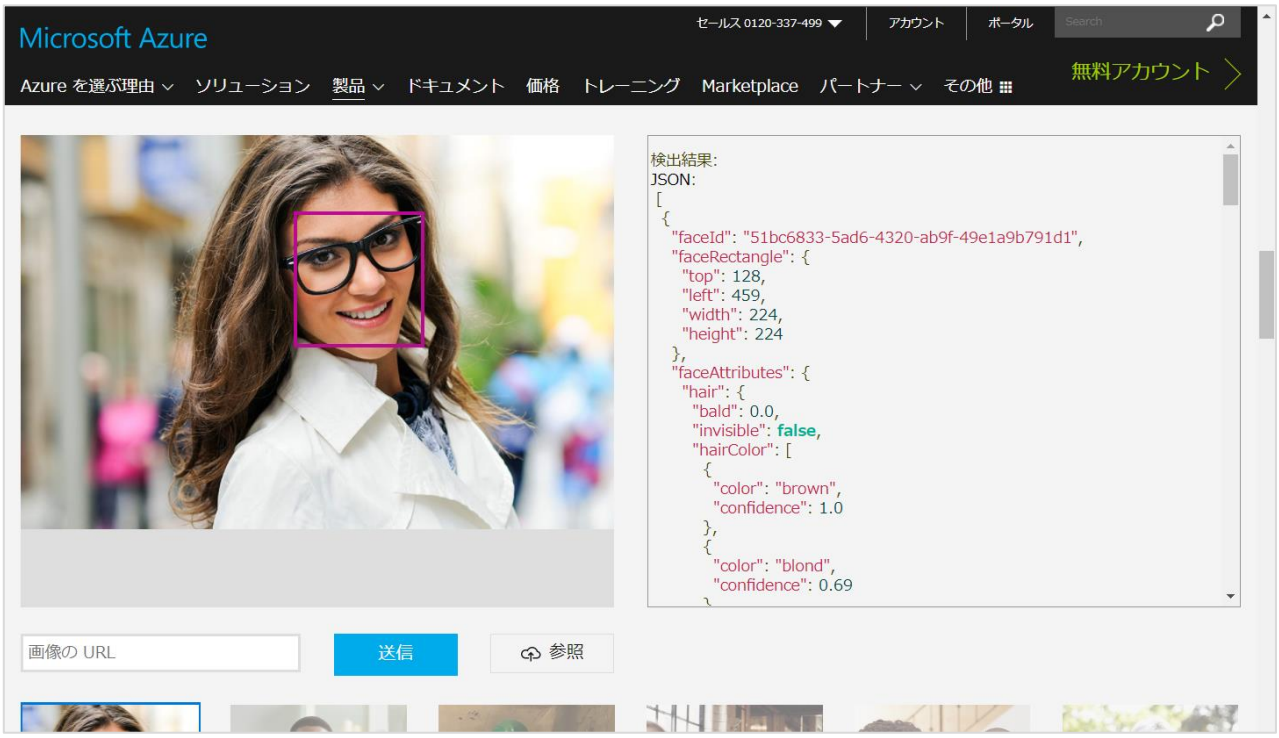
Microsoft Cognitive Services (https://azure.microsoft.com/ja-jp/services/cognitive-services/) とは、コンピューターによる認知機能を、マイクロソフトがサービスとしているものです。
単にハードウェアリソースを提供するのではなく、サービスとして認知機能を提供しています。

Cognitive Services は、5 つのカテゴリーと初期の技術検証サービスを集めたカテゴリー 1 つの合計 6 つのカテゴリーで構成されます。

- 視覚・・・写真や動画の識別、分析、サービス化など
- 音声・・・音声の識別や、音声とテキストとの相互変換など
- 言語・・・機械翻訳や、人間の意図の理解、話題の評価など
- 知識・・・自然言語での問い合わせに対して最適な回答を返すなど
- 検索・・・Bing で提供されている機能を API 化
- Labs・・・実験的なサービスの提供

各サービスの詳細ページで、提供されている機能を実際に体験することができます。

例えば Face API (https://azure.microsoft.com/ja-jp/services/cognitive-services/face/) のページでは、写真に含まれる顔について、位置や向き、感情、眼鏡の有無、髪の毛の色などの認知ができることを体験できます。



ワンポイント

Cognitive Services は、日々進化や拡張を繰り返しています。
機能の追加やサービスの正式リリース（GA）が高い頻度で行われているため、この自習書をお読みになるタイミングによっては内容が異なっている可能性があります。

8. Bot アプリケーション

Bot アプリケーションとは、入力に対して適切な応答を返すアプリケーションです。
これだと広い意味の言葉ですが、多くはチャットクライアントを通してサービスを利用する Web アプリケーションのことを Bot アプリケーションと呼びます。

Bot アプリケーションには、例えば、このようなシナリオがあります。

https://docs.microsoft.com/en-us/azure/bot-service/bot-service-scenario-overview

これらの例からも分かるように、Bot アプリケーションは必ずしも “AI” が使われているとは限りませんし、必ずしも使われている必要もありません。(“AI” 技術を利用している Bot アプリケーションも多く存在し、活発に研究開発が進められていることも事実です)
あくまでも「チャットクライアントを通してサービスを利用する Web アプリケーション」が Bot アプリケーションです。

この自習書では、この「チャットクライアントを通してサービスを提供する」 Bot アプリケーションの開発手順を紹介していきます。

マイクロソフトが提供する Bot アプリケーションは、BotBuilder SDK (Bot アプリケーション開発用のフレームワーク) を利用して開発します。
BotBuider SDK を利用し、Microsoft Azure 上で開発・運用される Bot アプリケーションを構築してくれるサービスが Bot Service です。

9. Bot Service

Azure Bot Service (https://azure.microsoft.com/ja-jp/services/bot-service/) は、Microsoft Azure で提供されているサービスです。
Bot アプリケーションを開発・運用するための基盤やサービスをまとめて構築してくれます。

2017 年 12 月に Azure Bot Service が GA (正式サービスとしてリリース) するまでは、Bot アプリケーションを開発するには

1. BotBuilder SDK を使って Bot アプリケーションを実装
2. Bot アプリケーションを専用ポータルで登録
3. Azure Web Apps でクラウドに配置

という手順を踏む必要がありました。

Azure Bot Service としてサービス化されたことで、これらがワンストップでできるようになりました。

次のステップから、実際に Azure Bot Service を使ったアプリケーション開発の実際に進んでいきます。

STEP 3. 開発を始める

このステップでは、Bot アプリケーション開発を始めるための準備と、作業 PC 上の Visual Studio 2017 で開発を進めるための手順を説明します。

- ✓ Web App Bot ソリューションの作成
- ✓ Visual Studio 2017 で開発
- ✓ Bot Framework Emulator でデバッグ実行
- ✓ NuGet パッケージのアップデート

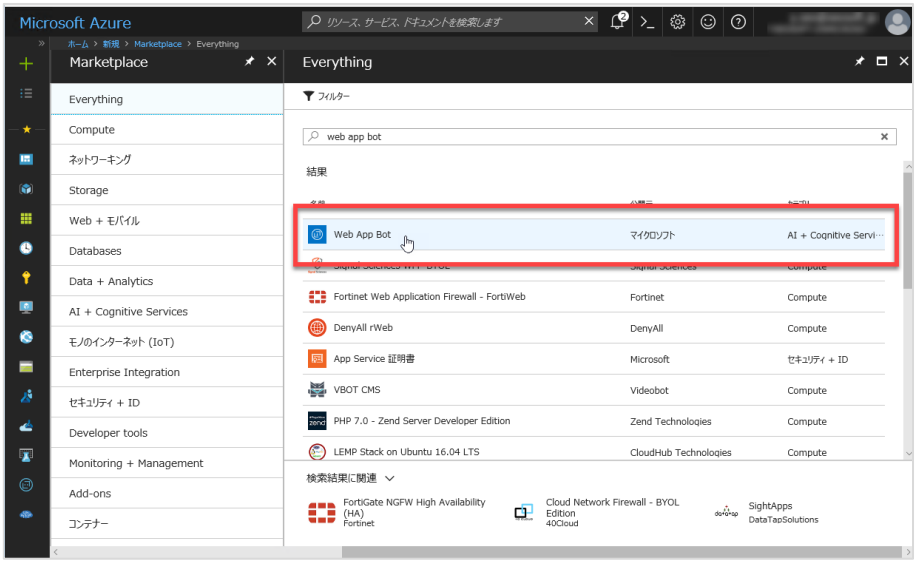
10. Web App Bot ソリューションの作成

ここから、いよいよ Bot アプリケーションを作っていきます。
この自習書で扱うのは、C# 版の Web App Bot です。

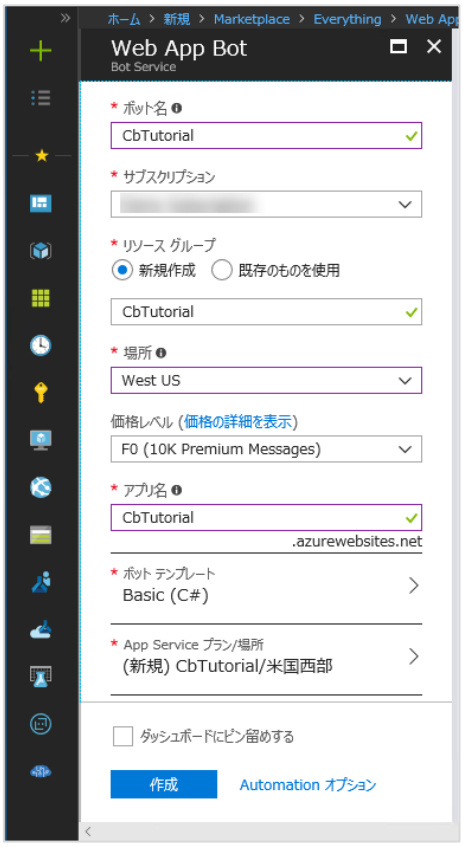
このステップでは、Microsoft Azure 管理ポータルで、Web App Bot を新規作成します。
また効率よくコーディングやデバッグを進めるために、ソースコードをローカルにダウンロードします。

1. Microsoft Azure 管理ポータル (https://portal.azure.com/) を開いて、サインインします。

2. [+] (リソースの作成) で、[Web App Bot] を選択します。

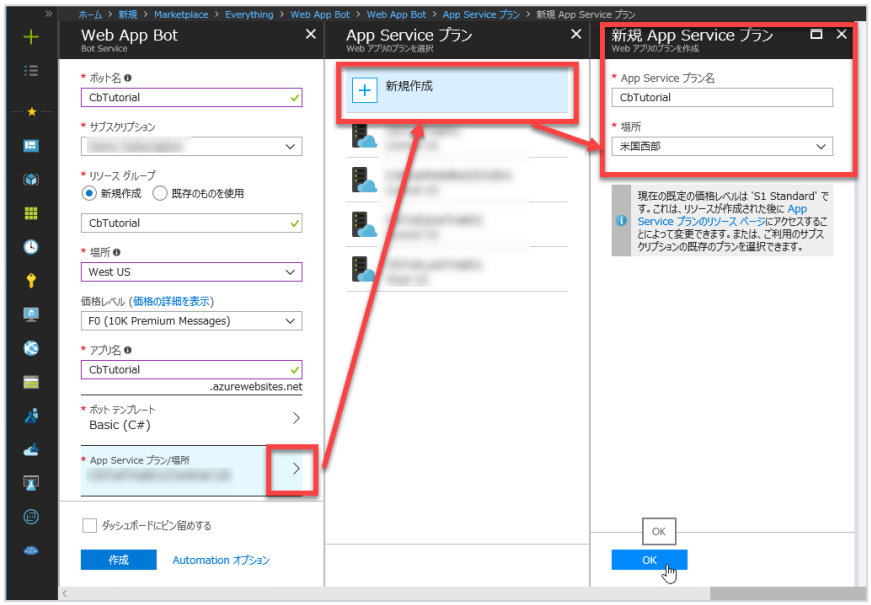


3. [Web App Bot] ブレードで [作成] をクリックします。[Web App Bot] ブレードが開いたら、以下の値で設定します。



設定項目	設定値
ボット名	任意の値
サブスクリプション	お持ちのサブスクリプション名
リソースグループ	[新規作成] ※既定値のままでかまいません
場所	選択可能な任意のリージョン
価格レベル	この自習書内では F0 (無料)。F0 を選択できない場合は S1
アプリ名	任意の値。ただしクラウド全体で一意の値のみ設定可能
ボットテンプレート	[Basic (C#)]
App Service プラン／場所	[>] をクリックして後述の通り設定します
Azure Storage	[新規作成] ※ストレージ名は既定値のままかまいません
Application Insights	[オン]
Application Insights の場所	選択可能な任意のリージョン
Microsoft App ID and password	[Auto create App ID and pass]

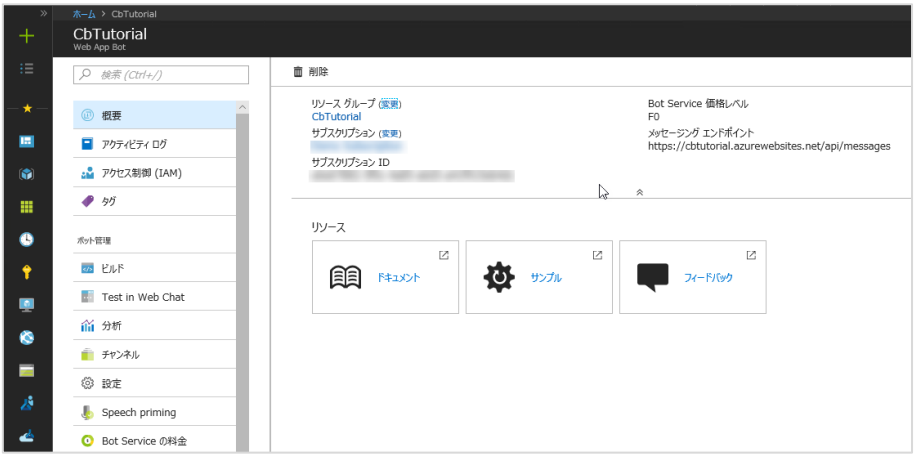
4. [新規 App Service プラン] ブレードで、App Service プランおよび場所を以下のように設定します。設定したら [OK] をクリックします。



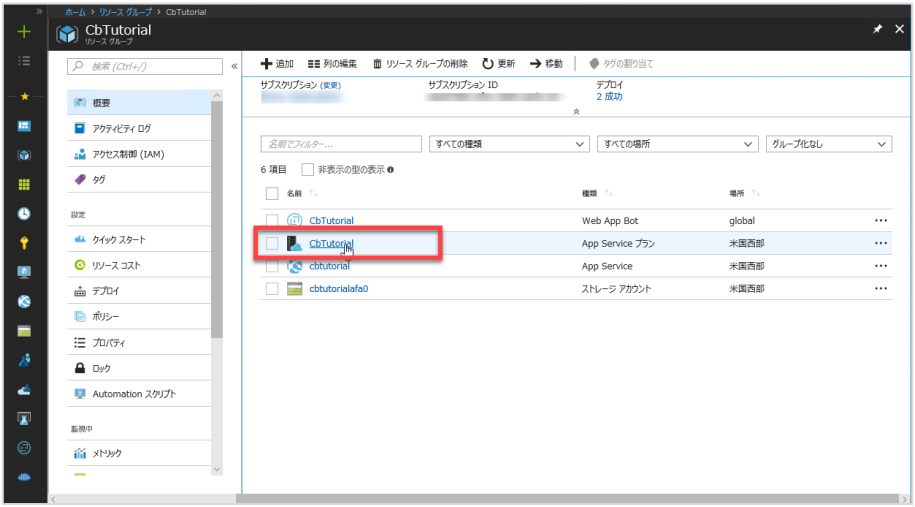
設定項目	設定値
App Service プラン名	任意の値 ※既定値のままでかまいません
場所	選択可能な任意のリージョン

5. [Web App Bot] ブレードで [作成] をクリックします。

6. 少し待つと新しい Bot Service が作成されます。



7. （以降はオプション）開発をする上では必須ではありませんが、この自習書の範囲では、App Service プランを “F1” (無償プラン) に変更することができます。
まず、Bot Service を作成したリソースグループを開きます。

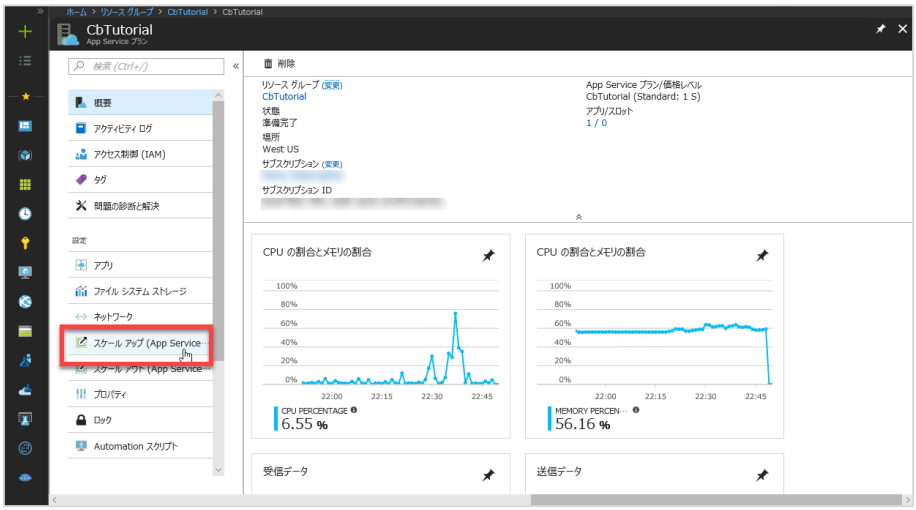


ワンポイント

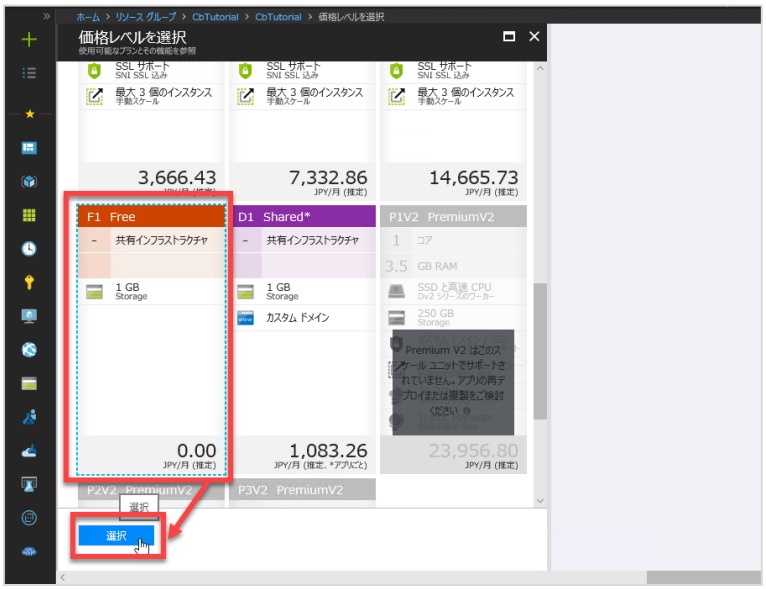
お持ちの Azure サブスクリプションの利用状況により “F1” を選択できないこともあります。その場合は、デフォルトで作成された “S0” のまま利用してください。

8. [スケールアップ] をクリックします。

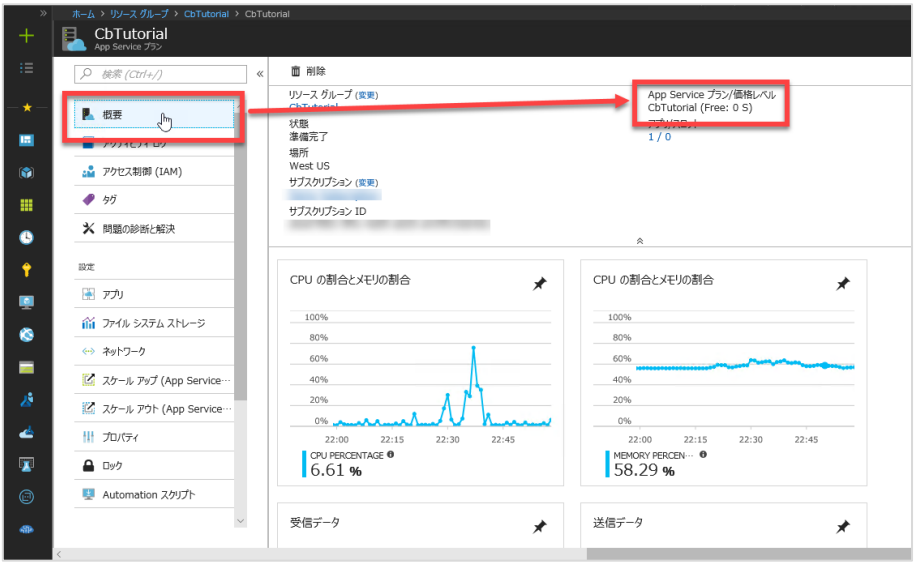
これから無償プランに変更しますが、メニュー項目としては“スケールアップ”となっています。



9. [F1 Free] を選択して、[選択] をクリックします。



10. 選択後の処理はすぐに終わります。[概要] をクリックして、価格レベルが“Free”になっていれば変更完了です。



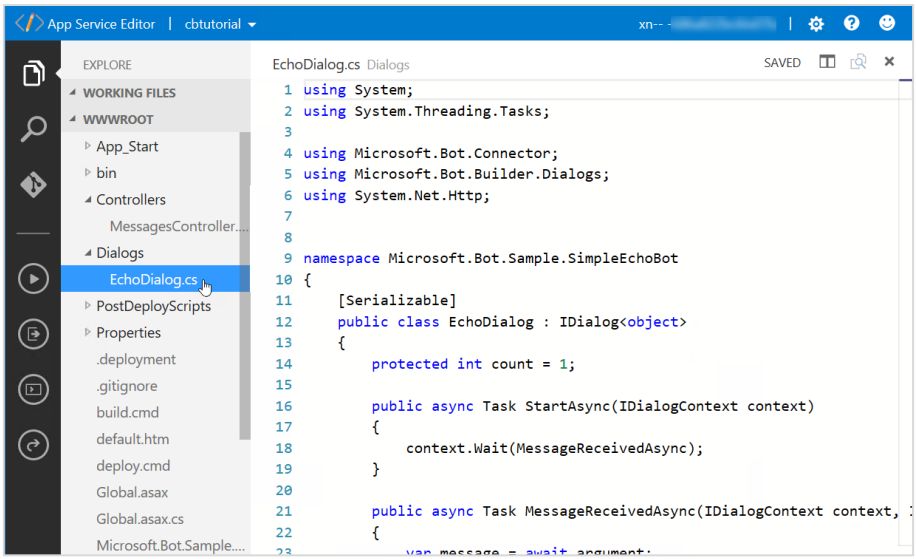
11. Visual Studio 2017 で開発

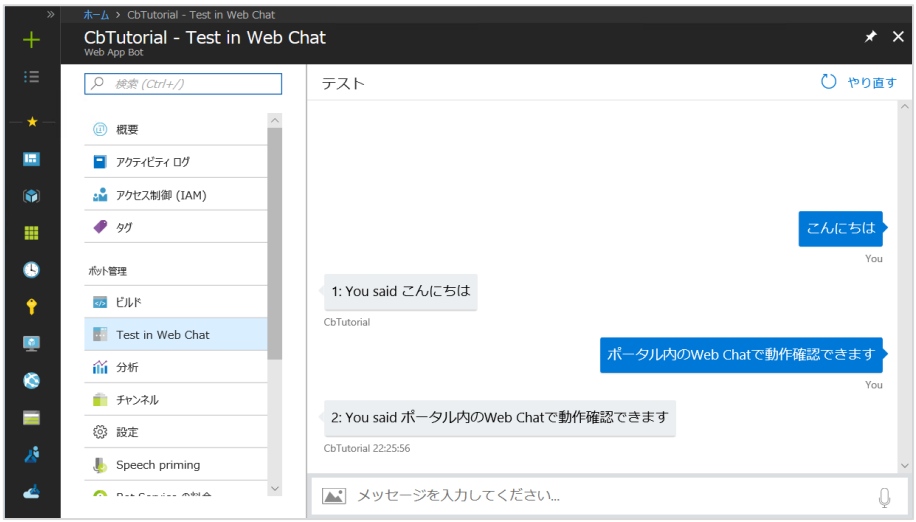
このステップでは、Bot アプリケーションのソースコードをダウンロードして、作業 PC 上の Visual Studio 2017 で開発を始めるための手順を説明します。

- オンラインコードエディターでの開発

Bot Service はオンラインコードエディターでソースコードを編集したり、管理ポータル内の Web Chat で動作確認をしたりすることができます。

ちょっとした修正や簡単な動作確認であれば、再デプロイまで含めて、すべてオンラインで済ませることもできます。





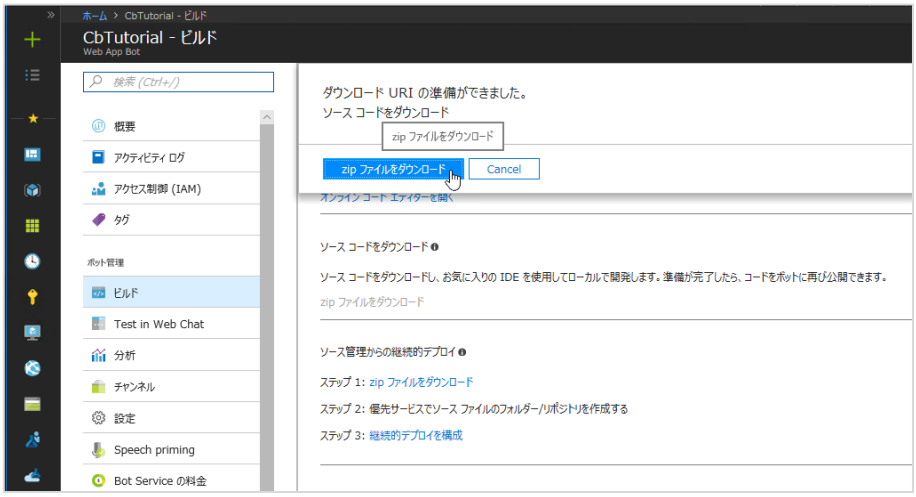
- Visual Studio 2017 での開発

オンラインコードエディターだけで Bot アプリケーションの開発を進めることも可能ですが、ローカルの Visual Studio を使うことでコーディングやデバッグの効率化が期待できます。

この自習書ではこれ以降、管理ポータルからソースコードをダウンロードして、作業 PC 上の Visual Studio 2017 で開発する手順を説明します。

1. [Web App Bot] ブレードのメニューで [ボット管理] – [ビルド] を選択し、[zip ファイルをダウンロード] をクリックします。

2. まもなく [zip ファイルをダウンロード] ボタンが表示されるのでダウンロードします。

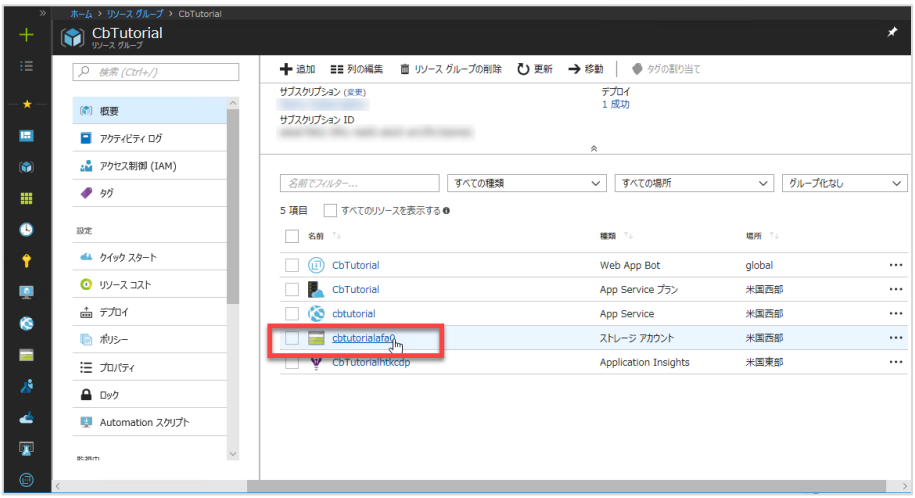


3. ダウンロードした zip ファイルを任意のフォルダーに展開します。

4. Visual Studio2017 でソリューションファイルを開きます。

5. [ビルド] – [ソリューションのリビルド] でソリューションをリビルドします。ビルドエラーが発生しないことを確認します。

6. Microsoft Azure 管理ポータルで、今回作成した Bot Service のリソースグループブレードに移し、ストレージアカウントのリソースを開きます。



7. ストレージアカウントブレードの [アクセスキー] をクリックします。

8. key1 の接続文字列の [コピー] をクリックして、接続文字列をクリップボードにコピーします。

9. Visual Studio 2017 に戻り、Web.config ファイルを開きます。

10. <appSettings> セクションの中の、<add key="MicrosoftAppPassword" value="" /> の行の下に、以下の行を挿入します。
<接続文字列> の部分には、前の手順でクリップボードにコピーした Azure ストレージアカウントの接続文字列を貼り付けます。

    <add key="AzureWebJobsStorage" value=" <接続文字列>" />

11. 保存して、もう一度ソリューションをビルドします。ビルドエラーが発生する場合は、Web.config の編集に間違いがないか確認してください。

以上で Bot アプリケーションを作業 PC 上の Visual Studio 2017 で開発する準備が整いました。
次のステップでは、Visual Studio 2017 でデバッグ実行する手順を紹介します。

ワンポイント

ダウンロードした Bot アプリケーションのソースコードは、ソリューション名、プロジェクト名や、名前空間などが“Sample.SimpleEchoBot”という名前になっており、決してスマートではありません。
ちょっとした検証目的などは別として、ソースコードをダウンロードしたら、最初にこれらを適切な名前に変更することをお勧めします。

この自習書では、Bot アプリケーション開発そのものに集中したいため、名前にはこだわらず、“Sample.SimpleBot”のままで進めます。

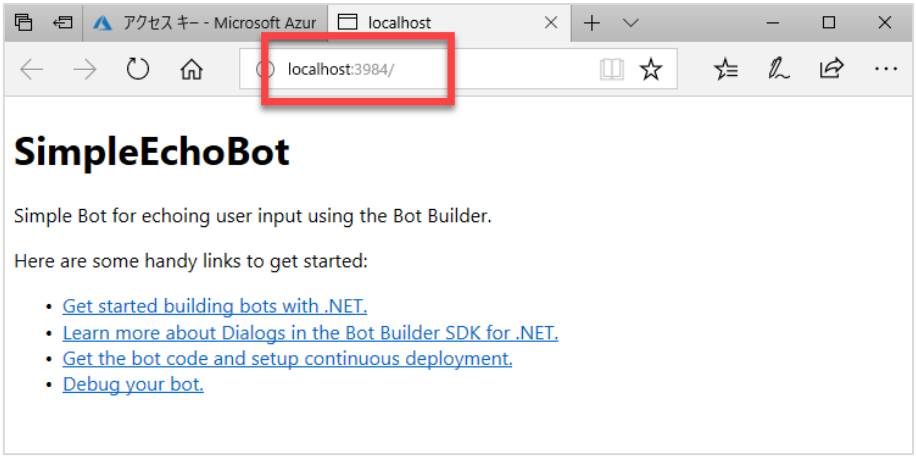
12. Bot Framework Emulator でデバッグ実行

ソースコードをダウンロードして、Web.config にストレージアカウントの接続文字列を設定したことで、Bot アプリケーションを Visual Studio 2017 でデバッグ実行する準備が整いました。

このステップでは、Visual Studio 2017 と Bot Framework Emulator とを使って Bot Appliation をデバッグ実行してみます。

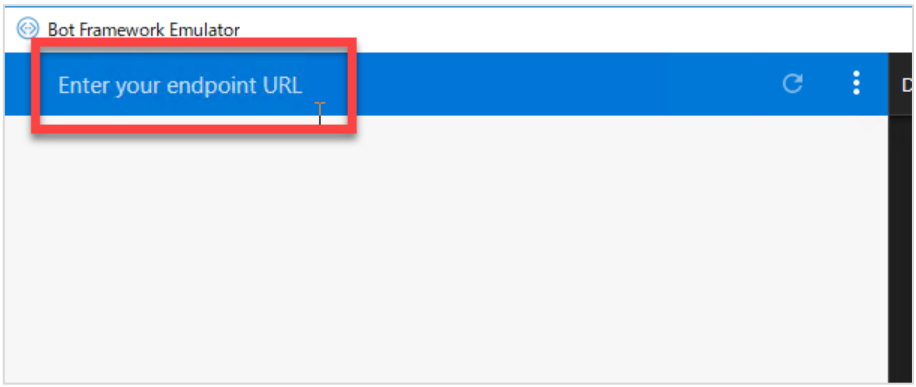
1. Visual Studio 2017 で [デバッグ] – [デバッグの開始] を選択します。

2. Web ブラウザー (規定では Microsoft Edge) が自動的に開いて、以下のようなページが表示されます。このページのアドレス “localhost:<ポート番号>/” は、この後の手順で使います。

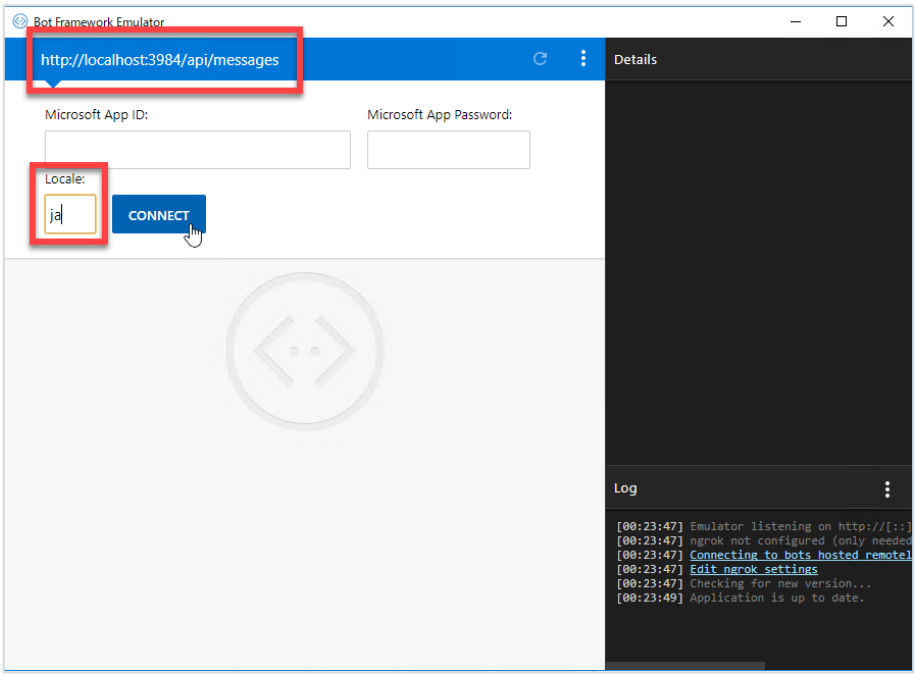


3. Bot Framework Emulator を起動します。

4. Bot Framework Emulator の [Enter your endpoint URL] 部分をクリックします。

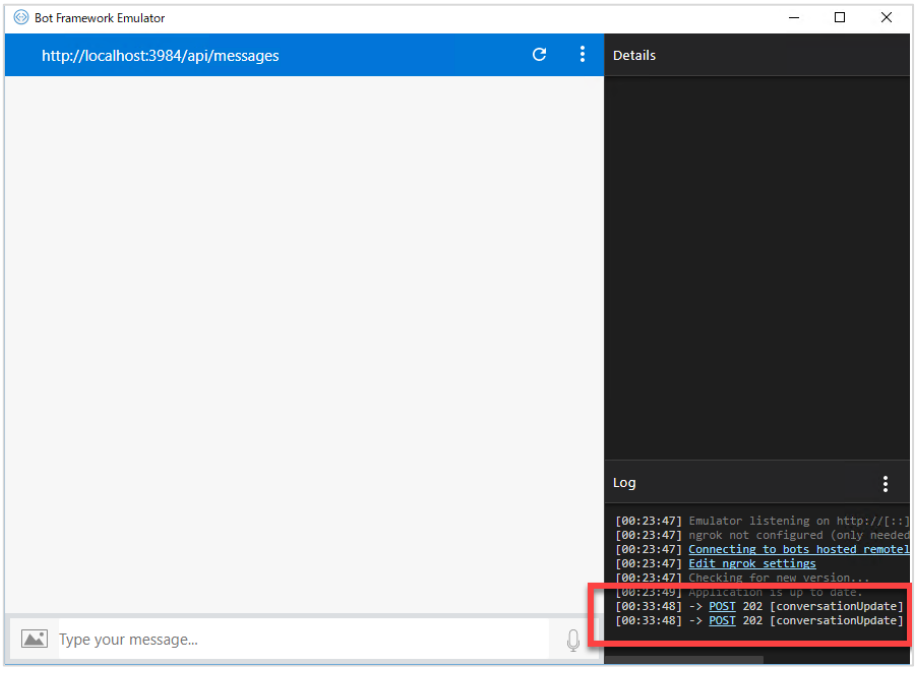


5. Web ブラウザーに表示されているアドレスを使って、“http://localhost:< ポート番号 >/api/messages” と入力します。[Locale] には “ja” と入力します。続いて、[Connect] をクリックします。



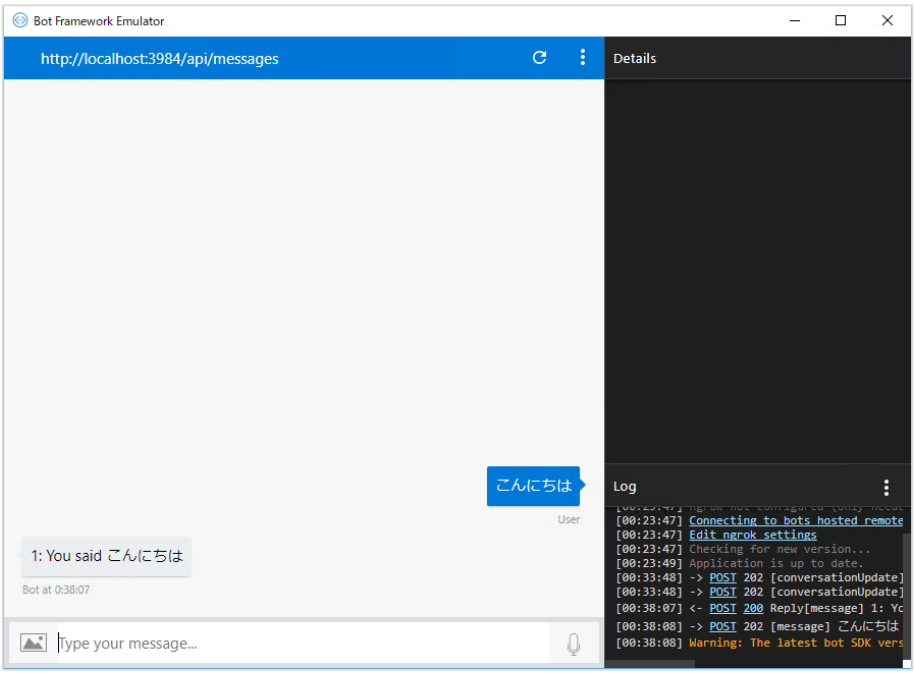
6. Bot Framework Emulator の右下のペインにログが表示されます。

Bot アプリケーションが正常に動作していれば、右下のログペインに “ [ConversationUpdate]” という行が 2 行表示されます。これは、ユーザーと Bot との両方がチャットに参加した、つまり双方向にメッセージを送受信できるようになったことを表します。



7. [Type youe message] の部分にメッセージを入力します。

8. Bot アプリケーションが、「1: You said <ユーザーが入力したメッセージ>」と応答します。



以上で、Visual Studio 2017 と Bot Framework Emulator とを使って、Bot アプリケーションのデバッグ実行できることが確認できました。

13. NuGet パッケージのアップデート

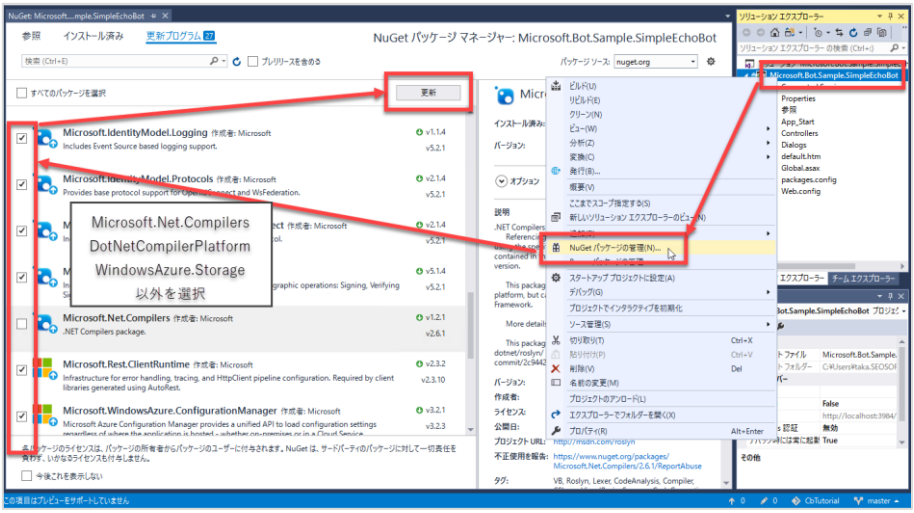
BotBuilder SDK for .NET (詳しくは 後述) は NuGet パッケージで提供されており、随時更新されます。他の NuGet パッケージも、それぞれ異なるタイミングで更新されます。

多くの場合、各パッケージを更新することで、パフォーマンスが向上したり不具合が修正されたりすることが期待できます。
ただし Bot アプリケーションに限らず、一部の NuGet を更新することでアプリケーションが動作しなくなるケースもあります。

2018 年 3 月時点では、以下の 3 個のパッケージは最新版に“更新しない”ようにしてください。3 個のパッケージを更新すると、ビルドエラーや実行時エラーが発生します。

1. ソリューションエクスプローラーの Project 名で右クリックして [NuGet パッケージの管理] を選択し、NuGet パッケージマネージャーで以下の“3 ファイル以外”を最新版に更新します。

NuGet パッケージ名	指定するバージョン
Microsoft.CodeDom.Providers.DotNetCompilerPlatform	1.0.1
Microsoft.Net.Compilers	1.2.1
WindowsAzure.Storage	8.7.0 (※初期状態では 7.2.1 ですが、8.7.0 まではアップデート可能)



2. 更新が終わったらソリューションをリビルドして、改めてデバッグ実行してみます。ビルドエラー、実行時エラーが発生しないことを確認します。

3. NuGet パッケージマネージャーで、"WindowsAzure.Storage" を 8.7.0 までアップデートします。

4. ソリューションをリビルドして、改めてデバッグ実行してみます。ビルドエラー、実行時エラーが発生しないことを確認します。

STEP 4. ユーザーと対話する

このステップでは、Bot アプリケーションのフレームワークである BotBuilder SDK for .NET を使用して、ユーザーと対話する Bot アプリケーションを開発します。

- ✓ BotBuilder SDK for .NET
- ✓ Controller クラスと Dialog クラス
- ✓ 新しい Dialog クラスの実装
- ✓ UserData で状態管理

14. BotBuilder SDK for .NET

BotBuilder SDK は、Bot アプリケーション開発に利用するフレームワークです。

特に難しいフレームワークではありませんが、今後の開発の際には、この自習書や公式のドキュメント(https://docs.microsoft.com/en-us/azure/bot-service/) を参照して、内容を理解してから利用してください。

BotBuilder SDK は、現在 2 種類公開されています。

- BotBuilder SDK for .NET ・・・NuGet パッケージをインストールできます
- BotBuilder SDK for Node.js ・・・npm コマンドでインストールできます

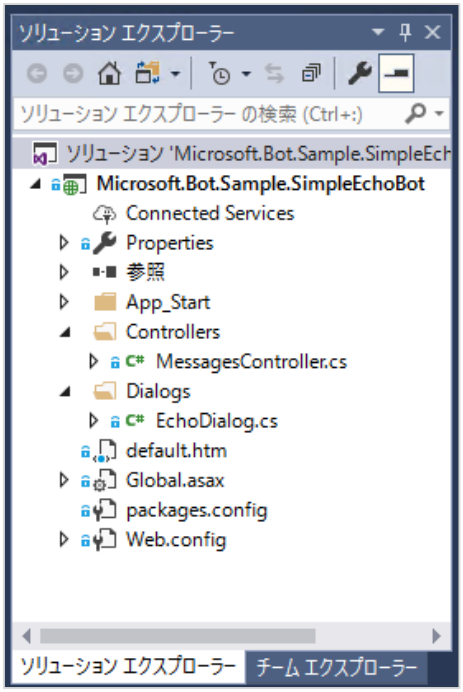
どちらの場合も Bot Service から Bot アプリケーションを作成すると、ソリューションに必要なパッケージが含まれます。このため通常は BotBuilder SDK のインストールを意識することはないかもしれません。
ただし、SDK は随時アップデートされます。BotBuilder SDK for .NET が更新された際には、NuGet パッケージマネージャーで更新します。

15. Controller クラスと Dialog クラス

実際にソースコードの内容を確認してみます。

Bot Service で作成した Bot アプリケーションは、ASP.NET MVC フレームワークを利用しています。.NET Framework または .NET Core 上の Web アプリケーションを開発したことがある方は、比較的容易に Bot アプリケーションの構造を理解することができるでしょう。

ただし、Bot アプリケーション自体は UI を持たない “Web サービス” なので、Views フォルダーの代わりに Dialogs フォルダーがあります。



ここでは、Bot アプリケーションにおいて大事な Controller クラスと Dialog クラスを見ることで、この後の機能拡張の方針を理解します。

1. Controller クラス

ASP.NET MVC フレームワークは、HTTP(S) で受信した要求を Controller クラスにルーティングします。受信した要求は、Post メソッドで処理されます。

```
[ResponseType(typeof(void))]
public virtual async Task<HttpResponseMessage> Post([FromBody] Activity activity)
{
    // check if activity is of type message
    if (activity != null && activity.GetActivityType() == ActivityTypes.Message)
    {
        await Conversation.SendAsync(activity, () => new EchoDialog());
    }
    else
    {
        HandleSystemMessage(activity);
    }
    return new HttpResponseMessage(System.Net.HttpStatusCode.Accepted);
}
```

チャットクライアントでユーザーがメッセージを入力すると、Controller には Activity という形で渡されます。
この時、Activity.ActivityGetActivityType の値 は ActivityTypes.Message になっています。文字通り、Activity の種類は“メッセージ”であるということです。

ActivityTypes は他には、

- ContactUpdate ・・・ユーザーまたは Bot が会話に参加したなど、会話の状態が変わったことを表します。
- Typing ・・・ (ユーザーに対して) Bot がメッセージを作成中であることを通知します。チャットクライアントで相手が入力中に“...”のアニメーションが表示されたり、”入力中です”などと表示されたりすることがあります。”Typing”はそれを指示するものです。
  ただし、どのように表示されるかはチャットクライアント次第です。Bot アプリケーションから表示をコントロールすることはできません。

などがあります。

ユーザーからのメッセージは常に ActivityTypes.Message であるので、これに応答する処理を記述することで、Bot アプリケーションとしての基本的な処理を実現できます。

Controller は ユーザーからのメッセージを自分自身で処理するのではなく、Dialog クラスに送信して、Dialog クラスでメッセージを組み立てたりユーザーにメッセージを送信したりします。

2. Dialog クラス

Dialog クラスは、ユーザーからのメッセージの内容に応じて、メッセージを生成してユーザーに応答する処理を記述します。

Dialog クラス定義は 2 点に注意してください。

- Serializable 属性を適用する
- IDialog インターフェイスを実装する

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

Serializable 属性が適用されているため、ユーザーごとに任意の情報をメンバー変数に保持しておき、継続する対話の中で利用することができます。（上記の例では count 変数）

Dialog で最初に呼ばれるメソッドは StartAsync メソッドです。
StartAsync メソッドでは、対話の状態を初期化したり、初回限定のメッセージをユーザーに送信したりすることができます。
その他の大事な処理として、Context.Wait メソッドでユーザーからのメッセージを処理するメソッドを設定するというものがあります。

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

上記のコードは、これ以降はユーザーからのメッセージを MessageReceivedAsync メソッドで処理することを意味します。

MessageReceivedAsync のもっとも簡単な実装例は以下の通りです。

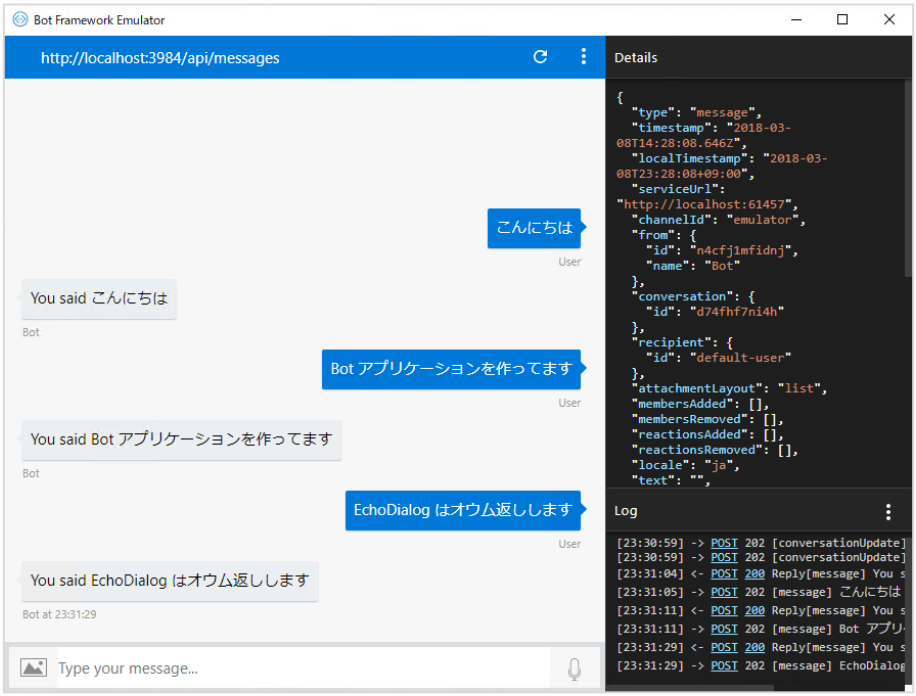
```
public async Task MessageReceivedAsync(IDialogContext context, IAwaitable<IMessageActivity> argument)
{
    var message = await argument;
    await context.PostAsync($"You said {message.Text}");
    context.Wait(MessageReceivedAsync);
}
```

Context.PostAsync メソッドで、ユーザーに対して実際にメッセージを送信します。この例では、”You said <ユーザーが入力したメッセージ>” を送信します。

最後に

```
context.Wait(MessageReceivedAsync);
```

を呼び出して、このユーザーからの次のメッセージも MessageReceivedAsync で処理するようにしています。



管理ポータルからダウンロードしたソースコードでは、MessageReceivedAsync メソッドはもう少し複雑な処理をしています。

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

    if (message.Text == "reset")
    {
        PromptDialog.Confirm(
            context,
            AfterResetAsync,
            "Are you sure you want to reset the count?",
            "Didn't get that!",
            promptStyle: PromptStyle.Auto);
    }
    else
    {
        await context.PostAsync($"{this.count++}: You said {message.Text}");
        context.Wait(MessageReceivedAsync);
    }
}
```

ユーザーの入力が“reset”の場合、 PromptDialog.Confirm メソッドを呼び出しています。
このメソッドはユーザーの Yes/No の選択を求めるもので、Windows アプリケーションの MessageBox に相当します。

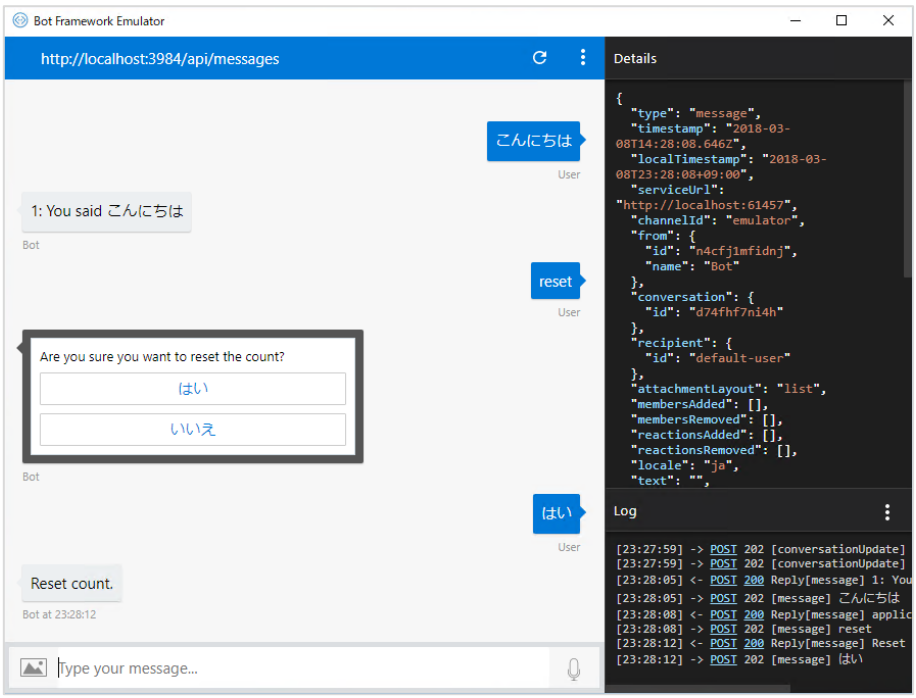
PromptDialog.Comfirm メソッドの第 2 引数はコールバック関数です。ユーザーが Yes/No を選択して処理が終了すると、ここで指定されたメソッド、上記のコードで言うと AfterResetAsync が呼び出されます。

AfterResetAsync メソッドでも、やはり最後に

context.Wait(MessageReceivedAsync);

を呼び出して、次のメッセージも MessageReceivedAsync で処理するように設定しています。

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

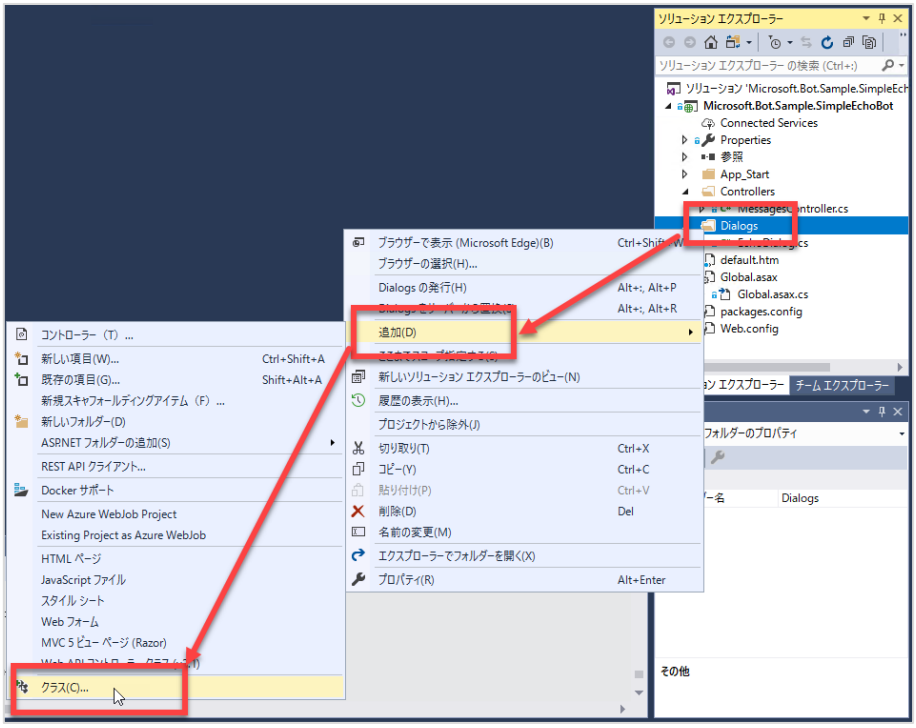


16. 新しい Dialog クラスの追加

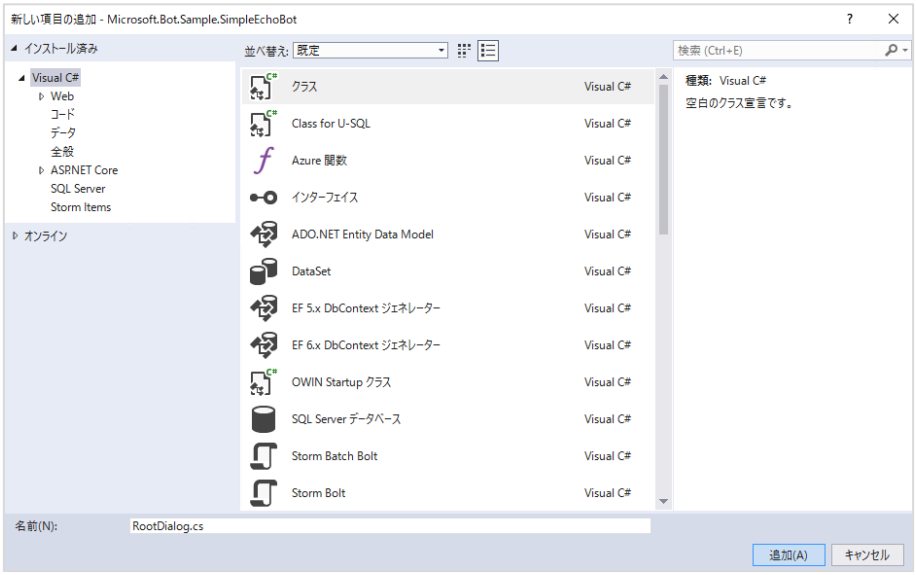
次に、Bot アプリケーションに独自の機能を追加実装していきます。
EchoDialog クラスを変更してもいいのですが、今回は新しい Dialog クラスを作ってみます。

なお、これ以降、EchoDialog クラスは使いません。ファイルが残っているのが気になる方は、EchoDiaog.cs を削除してもかまいません。

1. ソリューションエクスプローラーの“Dialogs”フォルダーで右クリックして、[追加] – [クラス] を選択します。



2. [新しい項目の追加] ダイアログで、[名前] に “RootDialog.cs” と入力して [追加] をクリックします。



3. RootDialog.cs 全体を以下のように変更します。

```
using System;
using System.Threading.Tasks;
using Microsoft.Bot.Builder.Dialogs;
using Microsoft.Bot.Connector;

namespace Microsoft.Bot.Sample.SimpleEchoBot.Dialogs
{
    [Serializable]
    public class RootDialog : IDialog<object>
    {
        public async Task StartAsync(IDialogContext context)
        {
            context.Wait(MessageReceivedAsync);
        }

        private async Task MessageReceivedAsync(IDialogContext context,
            IAwaitable<IMessageActivity> result)
        {
            var message = await result;
            await context.PostAsync($"{message.Text} と言いましたね");
            context.Wait(MessageReceivedAsync);
        }
    }
}
```

4. MessagesController.cs を開いて、ダイアログクラスへの送信を変更します。

具体的には、

new EchoDialog()

の部分を

new RootDialog()

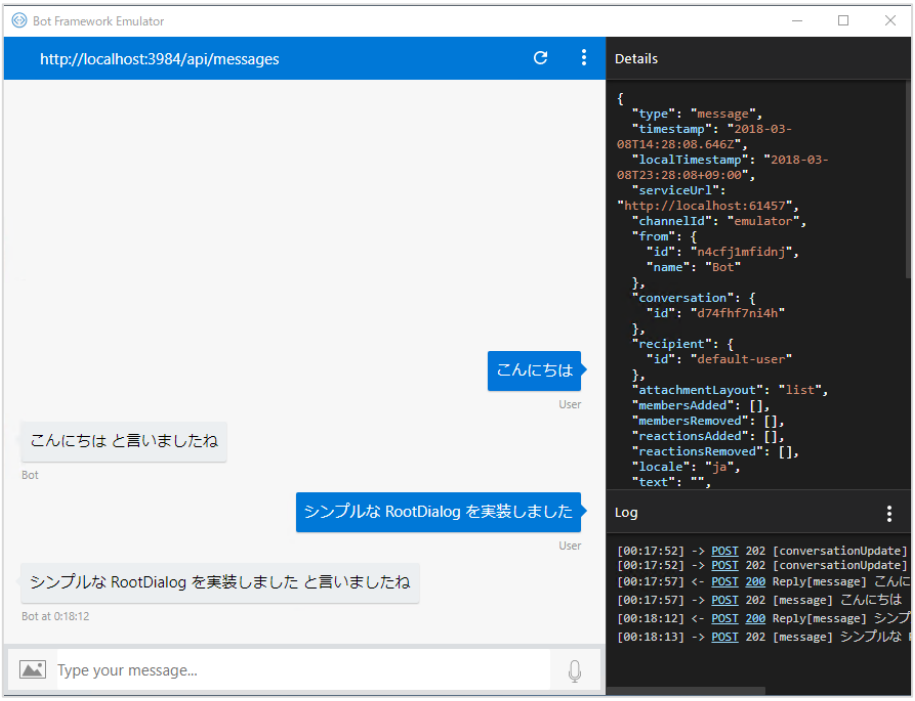
に変更します。

結果として、Post メソッドは以下のように記述します。

```
[ResponseType(typeof(void))]
public virtual async Task<HttpResponseMessage> Post([FromBody] Activity activity)
{
    // check if activity is of type message
    if (activity != null && activity.GetActivityType() == ActivityTypes.Message)
    {
        await Conversation.SendAsync(activity, () => new Dialogs.RootDialog());
    }
    else
    {
        HandleSystemMessage(activity);
    }
    return new HttpResponseMessage(System.Net.HttpStatusCode.Accepted);
}
```

5. ソリューションをビルドしたらデバッグ実行します。

Bot Framework Emulator で接続して、いくつかメッセージを入力してみます。Bot アプリケーションがオウム返ししてくれれば、シンプルな RootDialog の完成です。



17. UserData で状態管理

Bot アプリケーションでは、これまでの会話でユーザーから収集した情報や会話の状態を利用したいことがあります。

Dialog クラスは Serializable 属性が適用されているので、任意の型のデータを保持することができます。ただし、例えば「ユーザー名」のように長い期間に渡って情報を保持したい場合は、IDialogContext.UserData を利用します。
UserData は操作が非常に簡単です。開発者がストレージへのアクセスを意識することなく Azure テーブルストレージにユーザー情報を保存したり取得したりすることができます。

ワンポイント
UserData はデフォルトでは Azure テーブルストレージに永続化されるようになっていますが、Azure Cosmos DB やインメモリーのデータストアに変更することができます。 UserData のデータストアを変更するには、Global.asax.cs の Application_Start メソッドを変更してください。変更後のコードがコメントアウトされているので、Azure テーブルストレージ用の記述と切り替えるだけです。

以下では、Dialog のメンバー変数と Azure テーブルストレージに永続化される UserData を使って、ユーザー情報や会話の状態を利用する方法を説明します。

1. RootDialog クラス定義の先頭に、Private のメンバー変数 _needGreet を宣言します。

初めてこの Bot アプリケーションと会話するユーザーに対して、名前（ニックネーム、ハンドル）を尋ねて、その後で挨拶をするという処理にします。一連の処理のために _needGreed 変数を使用します。なお StartAsync メソッドには変更する箇所はありません。

```
[Serializable]
public class RootDialog : IDialog<object>
{
    private bool _needGreet;

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

2. MessageReceivedAsync メソッドを以下のように変更します。

```
private async Task MessageReceivedAsync(IDialogContext context,
    IAwaitable<IMessageActivity> result)
{
    var message = await result;

    string userName;
    if (!context.UserData.TryGetValue<string>("UserName", out userName))
        _needGreet = true;

    if (_needGreet)
    {
        await GreetAsync(context, message);
    }
    else if (message.Text == "こんにちは")
    {
        await context.PostAsync($"{userName} さん、こんにちは");
    }
    else
    {
        await context.PostAsync($"{message.Text} と言いましたね");
    }

    context.Wait(MessageReceivedAsync);
}
```

Context.UserData.TryGetValue メソッドは、

- UserData に “UserName” という要素がすでに保存されているかを確認
- 保存されていれば username 変数に値を取り出して、メソッドは true を返す
- 保存されていなければ false を返す

という処理を行います。
「Azure テーブルストレージにアクセスしてデータを取り出している」ということを意識せずに、直感的なコードでユーザー情報を保存、取得できることが分かります。
Bot アプリケーションがこのユーザーの名前をまだ知らない（＝このユーザーとは初めての会話である）場合、MessageReceivedAsync メソッドは GreetAsync メソッドを呼び出します。

ワンポイント

上記のコードで

```
string userName;
if (!context.UserData.TryGetValue<string>("UserName", out userName))
```

の箇所は、NuGet パッケージの Microsoft.CodeDom.Providers.DotNetCompilerPlatform および Microsoft.Net.Compilers を更新することで、もう少しすっきりしたコードに変更することができます。

```
if (!context.UserData.TryGetValue<string>("UserName", out var userName))
```

現在の C# の言語仕様を知っている方には少し物足りないコードですが、この自習書ではこれら 2 個の NuGet パッケージについては更新しない進め方にしたため、上記のようなコードになります。

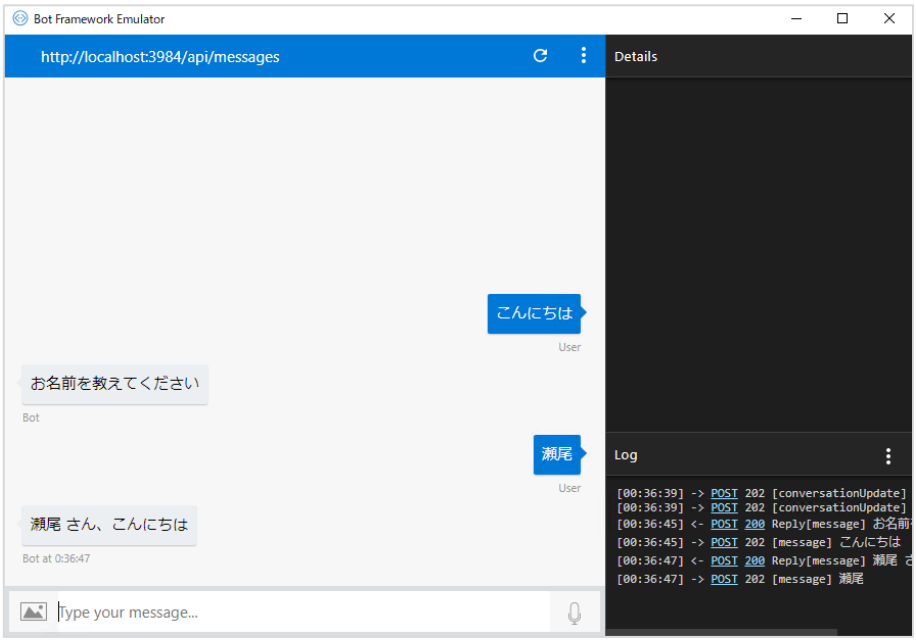
このあとに紹介する手順でも同じようなコードが出てきますが、同様の注意書きは行いません。

3. RootDialog クラスに GreetAsync メソッドを追加します。
ユーザーから取得した名前を UserData に保存したり、名前を保存したので今後は名前を問い合わせる必要がない旨を UserData に保存したりします。その後、挨拶の際にはユーザーの名前を含めます。

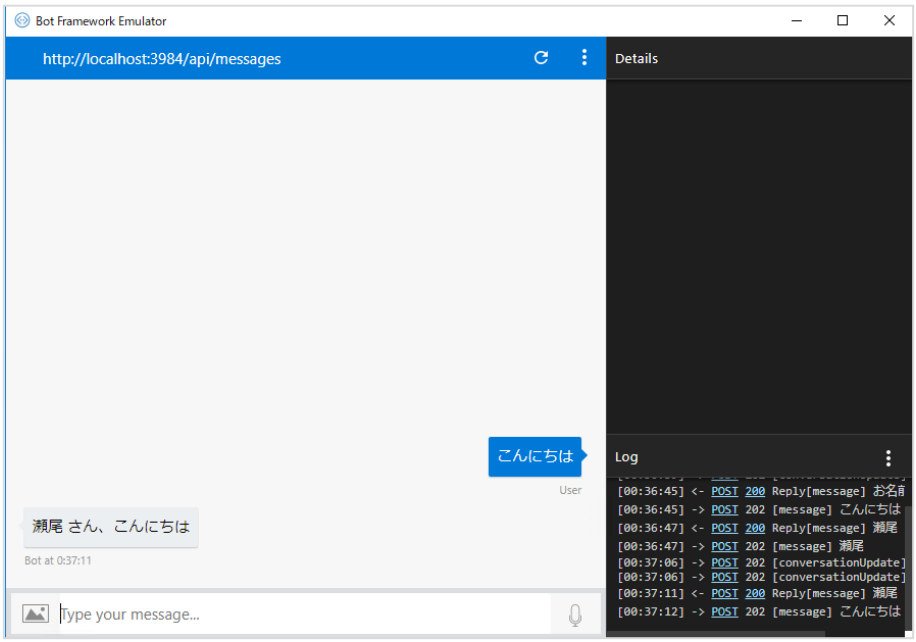
```
private async Task GreetAsync(IDialogContext context, IMessageActivity message)
{
    string userName;
    bool needUserName;
    if (!context.UserData.TryGetValue<string>("UserName", out userName) &&
        context.UserData.TryGetValue<bool>("NeedUserName", out needUserName))
    {
        userName = message.Text;
        context.UserData.SetValue("UserName", userName);
        context.UserData.SetValue("NeedUserName", false);
    }

    if (string.IsNullOrEmpty(userName))
    {
        context.UserData.SetValue("NeedUserName", true);
        await context.PostAsync("お名前を教えてください");
    }
    else
    {
        _needGreet = false;
        await context.PostAsync($"{userName} さん、こんにちは");
    }
}
```

4. ビルドに成功したら、デバッグ実行します。

初回の会話では、Bot アプリケーションはユーザーに名前を尋ねます。



2 回目以降の会話では、ユーザーが「こんにちは」と入力した場合は挨拶を返し、それ以外のメッセージの場合はユーザーのメッセージをオウム返しします。



ここまでで、Bot アプリケーションの基本的な実装は完了です。

STEP 5. 自然言語を理解する

このステップでは、Language Understanding (LUIS) を利用して、Bot アプリケーションが自然言語を処理できるように機能拡張します。

- ✓ Language Understanding (LUIS)
- ✓ Language Understanding アプリケーションの作成
- ✓ BotBuilder SDK から Language Understanding の呼び出し

18. Language Understanding (LUIS)

Language Understanding は、人間が発した文から「価値のある情報」を特定するためのサービスです。

価値のある情報とは、例えば、「今日の予定は何？」というユーザーからのメッセージであれば、

- 「予定の問い合わせ」という意図
- 「今日」という具体的な要素

のことです。

Language Understanding は、このようなユーザーの意図 (Intent)と要素 (Entity) を抽出してくれます。Azure Bot Service とシームレスに結合されているため、自然言語による命令や問い合わせを Bot アプリケーションに実装しやすくなっています。

Intent や Entity は、Language Understanding の管理画面で開発者が定義します。また例文を管理画面で登録して Intent や Entity に紐づけることで、知能を獲得して "賢く" なっていきます。

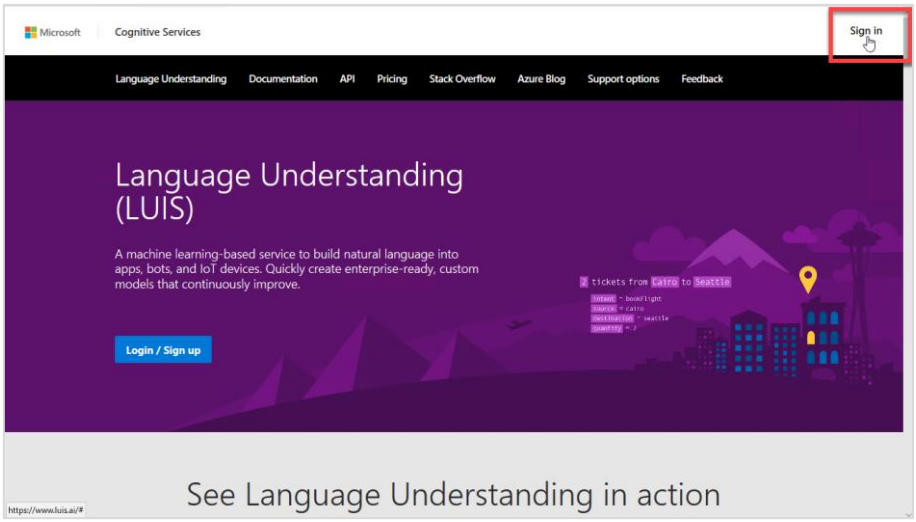
公式サイト (https://azure.microsoft.com/ja-jp/services/cognitive-services/language-understanding-intelligent-service/) には、サンプルやドキュメントが掲載されています。一度全体に目を通しておくことをお勧めします。

ワンポイント
Language Understanding は、以前は LUIS (Language Understanding Intelligent Service) と呼ばれていました。 このため、マイクロソフト公式、またはその他の情報源でも “LUIS” と記載されているものが多数あります。Cognitive Services に含まれるサービスとして LUIS という表記が出てきた場合には、Language Understanding のことだと考えてください。

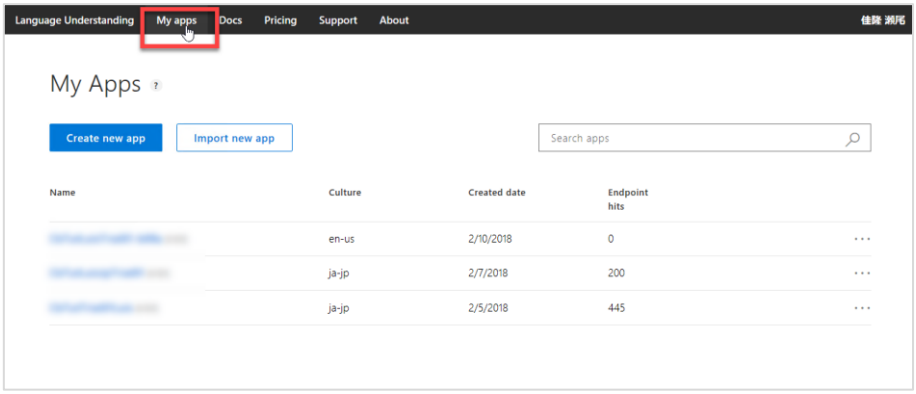
19. Language Understanding アプリケーションの作成

Language Understanding アプリケーションを作成する手順は以下の通りです。

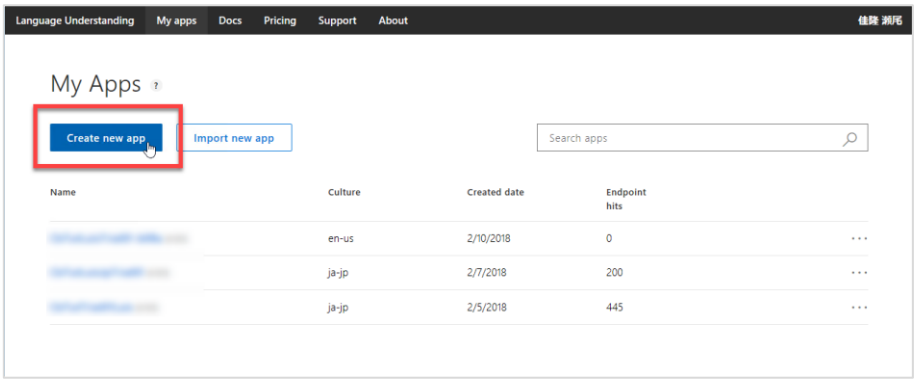
1. Language Understanding のサイト (https://www.luis.ai/home) を Web ブラウザーで開き、サインインします。



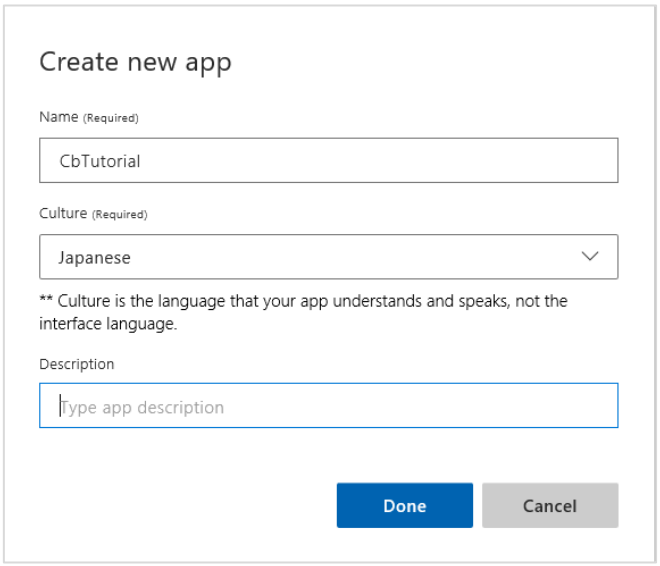
2. サインインに成功すると、“My Apps” ページに自動的に遷移します。もし遷移しない場合はページ上部の [My Apps] をクリックしてください。



3. [Create new app] をクリックして、新しいアプリケーションを作成します。



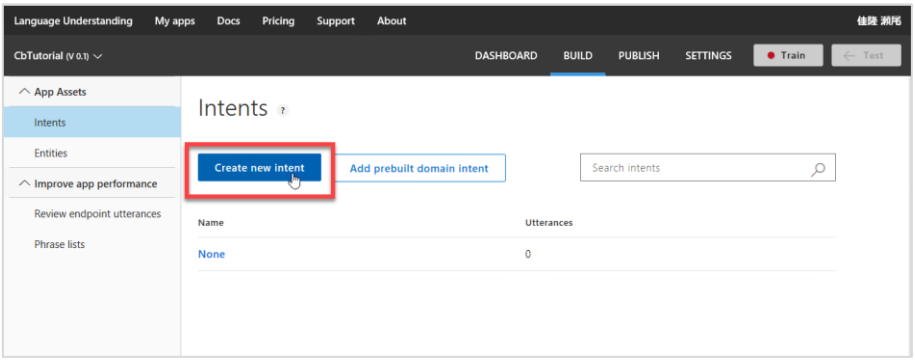
4. [Create new app] 画面で、以下の値で設定します。すべて入力したら [Done] をクリックします。



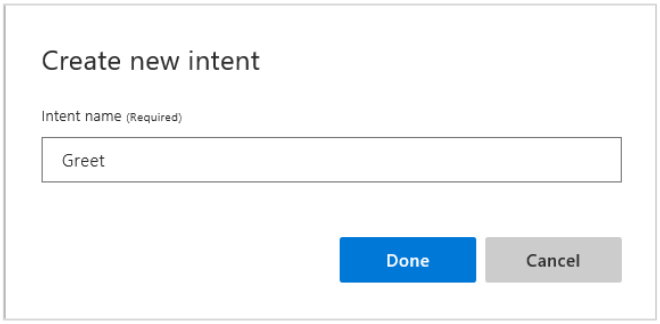
設定項目	設定値
Name	任意の値
Culture	Japanese
Description	任意の値（開発者向けの説明文であり、空白のままでもかまいません）

5. Language Understanding アプリケーションが作成されると [Intent] 画面が自動的に開きます。

6. [Create new intent] をクリックします。

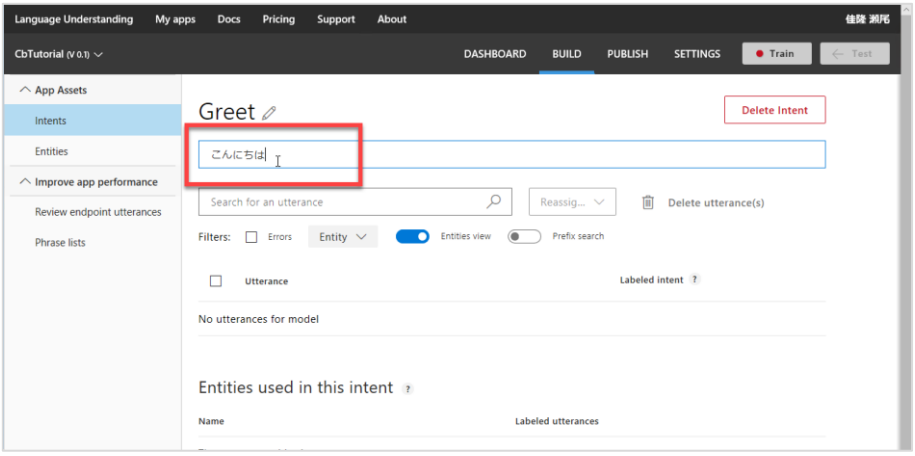


7. [Create new intent] 画面で、"Greet" と入力します。



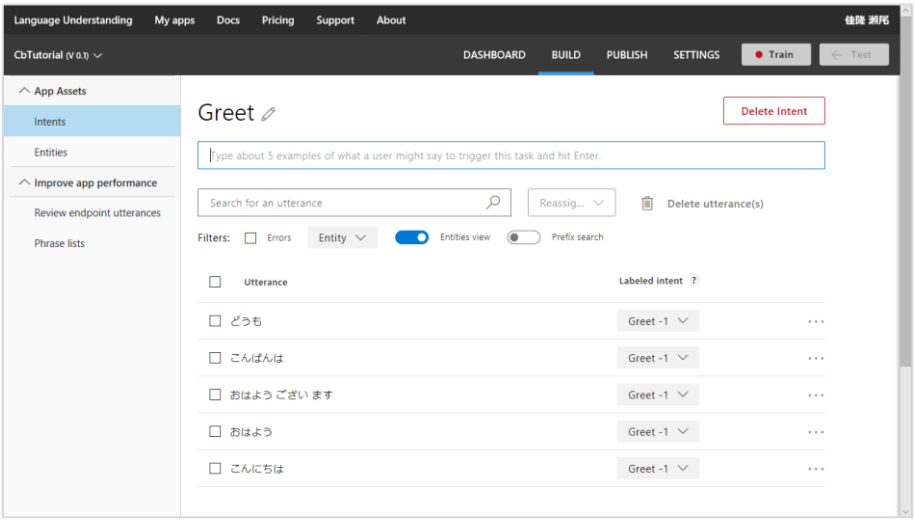
8. [Greet] Intent 画面で、"Type about 5 examples ～" というウォーターマークが表示されている欄に、"こんにちは" と入力してエンターキーを押します。

「こんにちは」という例文は、"Greet" という "Intent" であることを意味します。



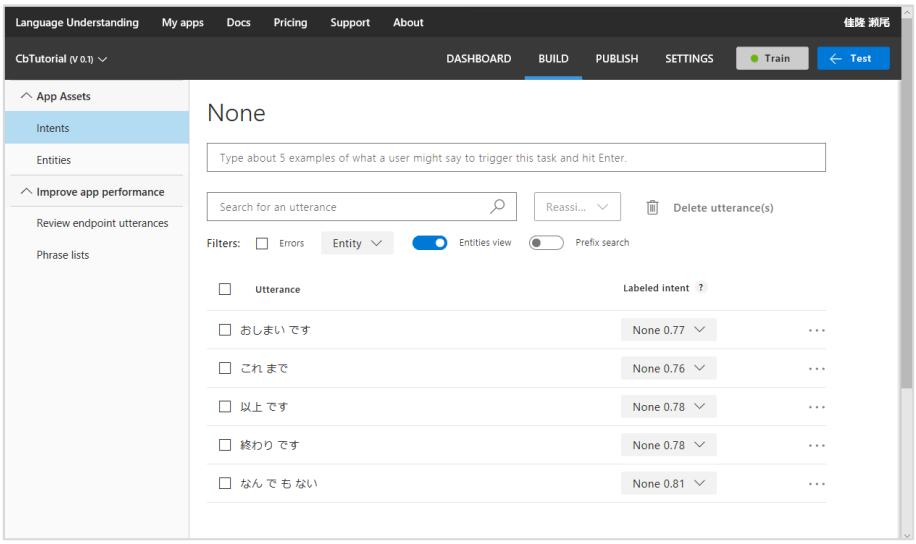
9. 同様に 5 個程度以上の挨拶（おはようございます、こんばんは、など）を登録します。
例えば、以下があります。（同じものでなくてもかまいません）

- こんにちは
- おはよう
- おはようございます
- こんばんは
- どうも



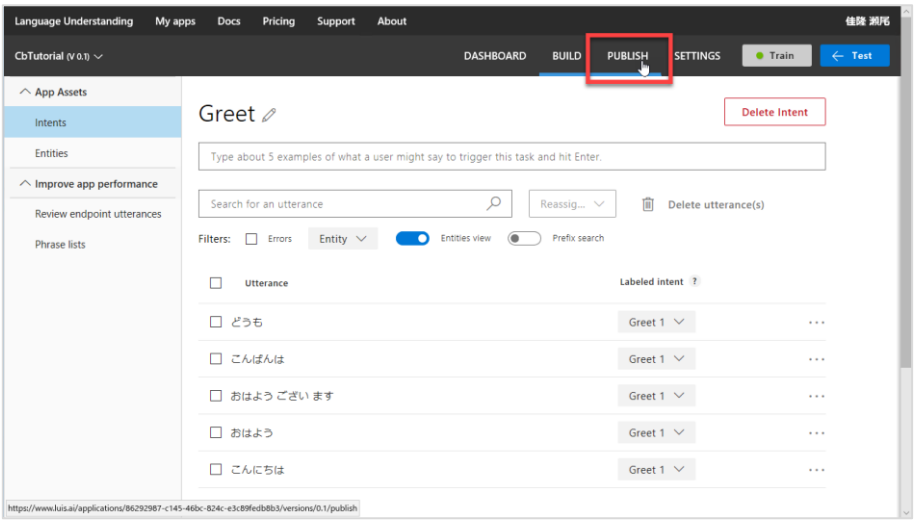
10. None インテントにも 5 個程度の例文を入力します。
例えば、以下を登録します。（同じものでなくてもかまいません）

- なんでもない
- 終わりです
- 以上です
- おしまいです
- これまで



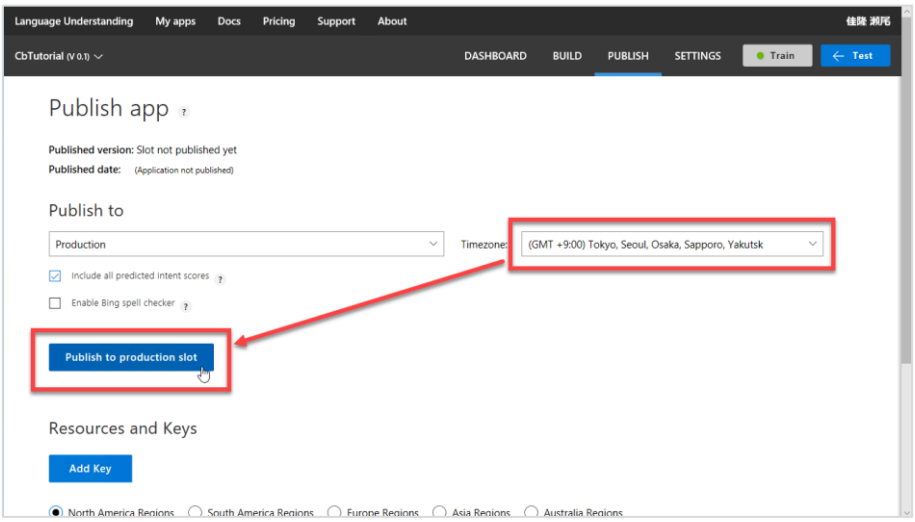
11. [Train] ボタンをクリックして学習します。少し待つと学習が完了します。

12. [PUBLISH] ボタンをクリックして、Language Understanding アプリケーションを発行します。



13. [Timezone] で “GMT +09:00” を選択します。

14. [Publish to production slot] をクリックします。

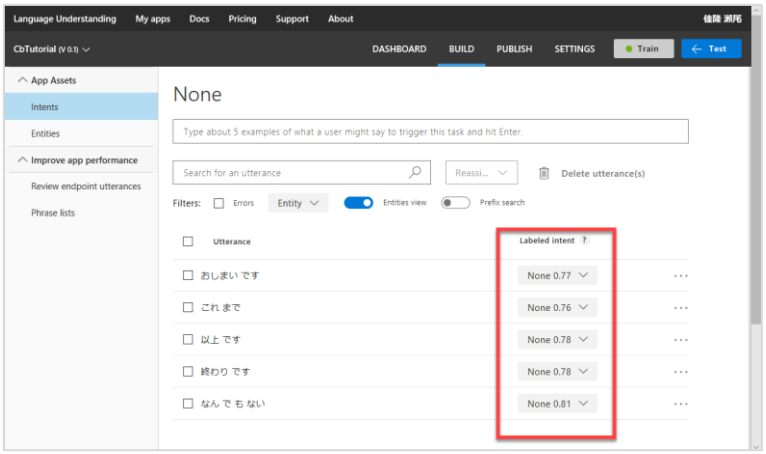


以上で、Language Understanding アプリケーションの発行が完了しました。

ワンポイント

Intent 画面で、登録済みの例文の右側に“インテント名 数値”と表示されています。
これは、その例文がどのインテントであると分類され、その確度はどのくらい（0～1 の数値）であるかを表します。
期待とは異なるインテントであると分類される場合、または確度が低い場合は、さらに例文を登録して学習を進めてください。

なお後述の LuisDialog のコンストラクターでは、” threshold” 属性に 0.7 を指定しています。
これは確度が 0.7 以上の場合にそのインテントとして分類することを意味します。
実際の開発では、学習量と threshold とのバランスを取りながらチューニングしていく必要があります。

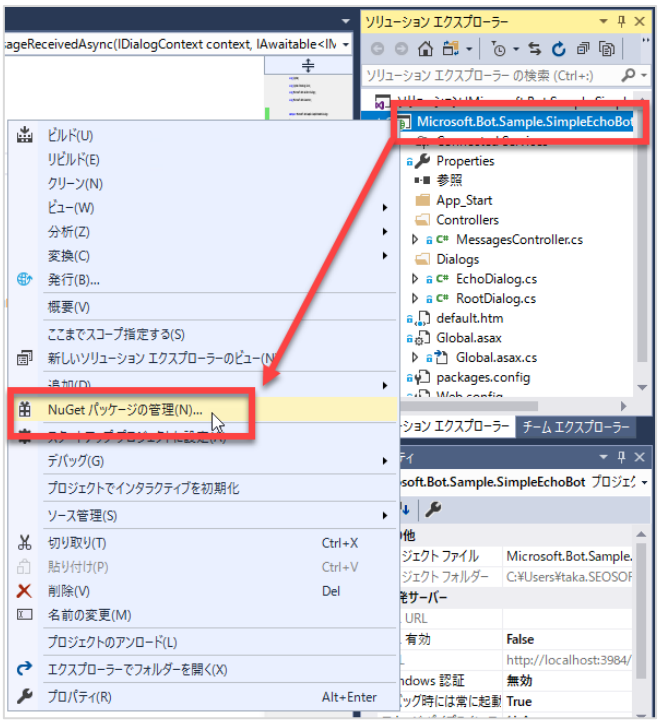


20. BotBuilder SDK から Language Understanding の呼び出し

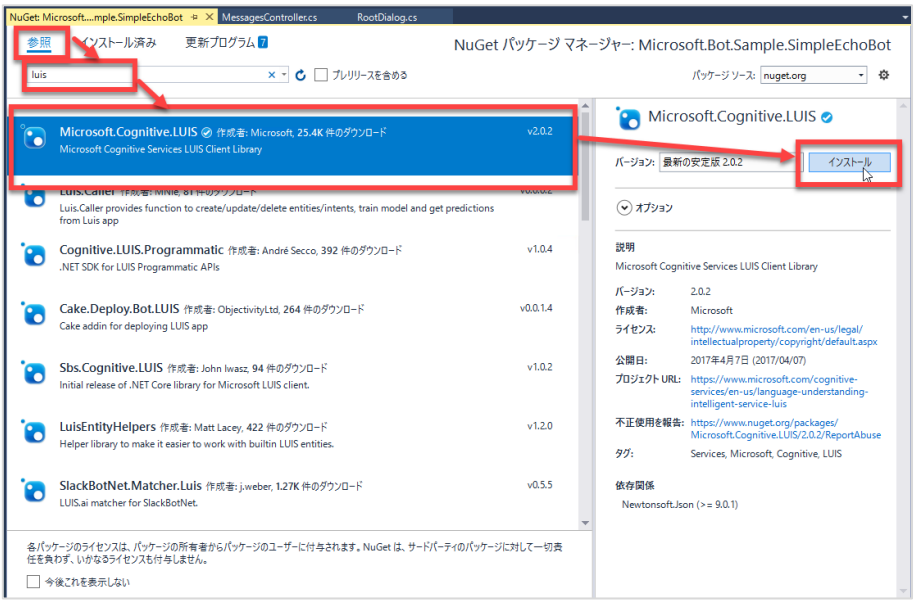
前の手順で、Language Understanding アプリケーションの発行が完了しました。
続いて、開発中の Bot アプリケーションから、この Language Understanding アプリケーションを呼び出してみます。

Language Understanding アプリケーションは RESTful API に対応していますが、今回は Language Understanding 用の NuGet パッケージを利用します。

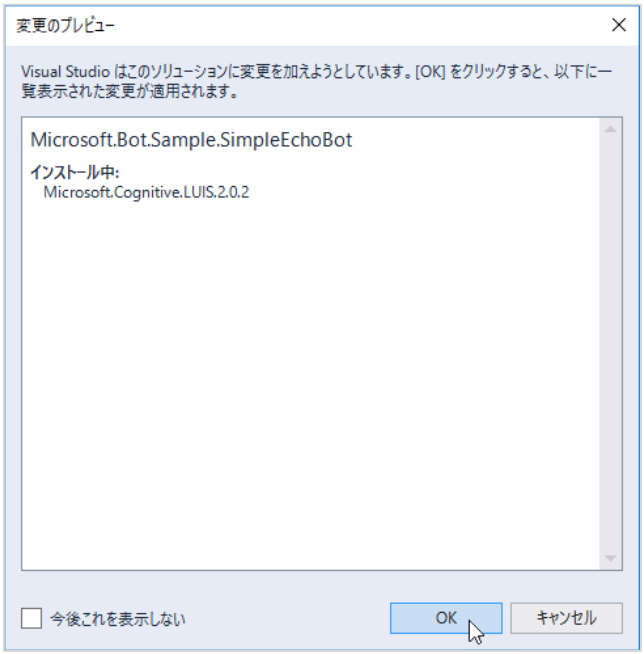
1. Visual Studio 2017 のソリューションエクスプローラーのプロジェクトで右クリックして、[NuGet パッケージの管理] を選択します。



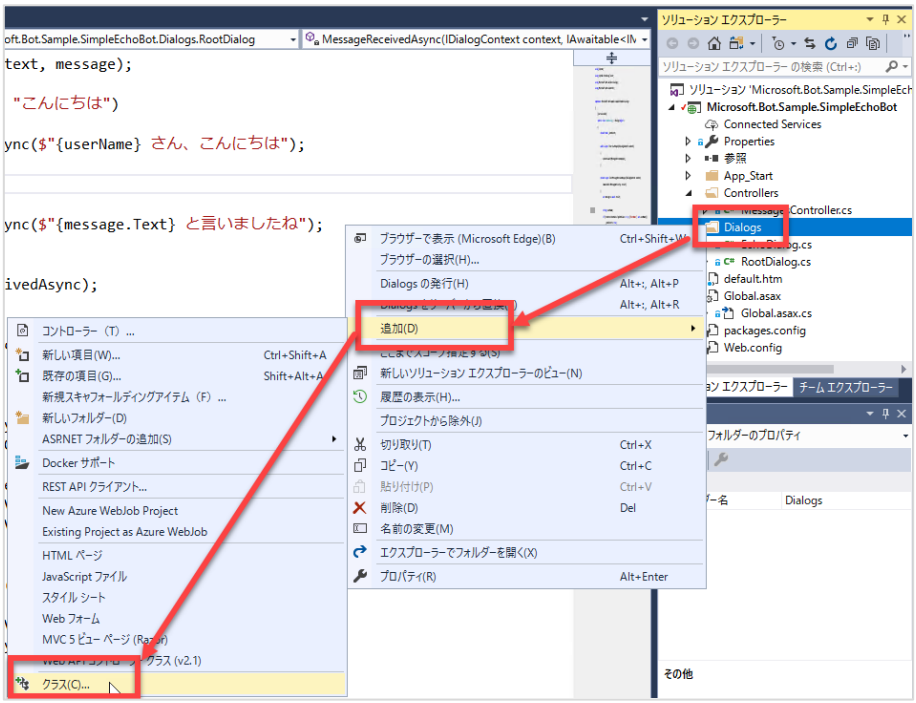
2. NuGet パッケージマネージャーで、[参照] を選択し、テキストボックスに “luis” と入力します。続いて、“Microsoft.Cognitive.LUIS” を選択して [インストール] をクリックします。



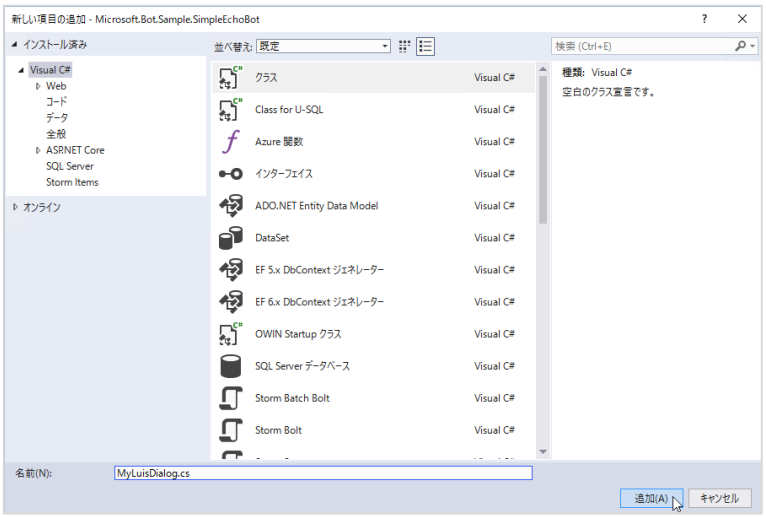
3. [変更のプレビュー] 画面で [OK] をクリックします。Language Understanding の NuGet パッケージがインストールされます。



4. Language UnderStanding を処理する Dialog クラスを追加します。ソリューションエクスプローラーの [Dialogs] フォルダーで右クリック、[追加] – [クラス] を選択します。



5. 名前には ”MyLuisDialog.cs” を指定します。



6. MyLuisDialog.cs 全体を以下のように変更します。

```
using System;
using System.Configuration;
using System.Threading.Tasks;
using Microsoft.Bot.Builder.Dialogs;
using Microsoft.Bot.Builder.Luis;
using Microsoft.Bot.Builder.Luis.Models;
using Microsoft.Bot.Connector;

namespace SimpleEchoBot.Dialogs
{
    [Serializable]
    public class MyLuisDialog : LuisDialog<object>
    {
        public MyLuisDialog() : base(new LuisService(new LuisModelAttribute(
            ConfigurationManager.AppSettings["LuisAppId"],
            ConfigurationManager.AppSettings["LuisAPIKey"],
            apiVersion: LuisApiVersion.V2,
            domain: ConfigurationManager.AppSettings["LuisAPIHostName"],
            threshold: 0.7)))
        {
        }

        [LuisIntent("None")]
        public async Task NoneIntent(IDialogContext context,
            IAwaitable<IMessageActivity> activity, LuisResult result)
        {
            await context.PostAsync($"{result.Query} と言いましたね");
            context.Wait(MessageReceived);
        }

        // 次ページに続く
```

```
        // 前ページから

        [LuisIntent("None")]
        public async Task NoneIntent(IDialogContext context,
            IAwaitable<IMessageActivity> activity, LuisResult result)
        {
            await context.PostAsync($"{result.Query} と言いましたね");
            context.Wait(MessageReceived);
        }

        [LuisIntent("Greet")]
        public async Task GreetIntent(IDialogContext context,
            IAwaitable<IMessageActivity> activity, LuisResult result)
        {
            string userName;
            if (context.UserData.TryGetValue<string>("UserName", out userName))
                await context.PostAsync($"{userName} さん、こんにちは");
            else
                await context.PostAsync("こんにちは");

            context.Done(true);
        }
    }
}
```

これまで RootDialog で記述してきたコードと違い、メッセージを処理するメソッドの最後は、

```
context.Wait(MessageReceivedAsync);
```

ではなく、

```
context.Done(true);
```

とします。

Done メソッドは、RootDialog から Forward されたメッセージ処理をここで終了することを意味します（Forward とは RootDialog でメッセージ処理せずに、MyLuisDialog に処理を委譲するということです）。
これにより、ユーザーの次のメッセージはまた改めて RootDialog で処理されるようになります。

7. RootDialog.cs の先頭付近を以下のように変更します。
“Using SimpleEchoBot.Dialogs;” の行を追加しています。

```
using System;
using System.Threading;
using System.Threading.Tasks;
using Microsoft.Bot.Builder.Dialogs;
using Microsoft.Bot.Connector;
using SimpleEchoBot.Dialogs;

namespace Microsoft.Bot.Sample.SimpleEchoBot.Dialogs
{
        // 以下、省略
```

8. RootDialog.cs の MessageReceivedAsync メソッド全体を以下のように変更します。

```
private async Task MessageReceivedAsync(IDialogContext context,
    IAwaitable<IMessageActivity> result)
{
    var message = await result;

    string userName;
    if (!context.UserData.TryGetValue<string>("UserName", out userName))
        _needGreet = true;

    if (_needGreet)
    {
        await GreetAsync(context, message);
        context.Wait(MessageReceivedAsync);
    }
    else
    {
        await context.Forward(new MyLuisDialog(),
            AfterLuisAnswerAsync, message, CancellationToken.None);
    }
}
```

挨拶が必要な場合は GreetAsync メソッドが呼ばれます。
それ以外の場合（ほとんどの場合はこちらになるでしょう）、Forward メソッドを使って MyLuisDialog に処理を委譲します。

9. RootDialog.cs の最後に、以下の通り AfterLuisAnswerAsync メソッドを追加します。

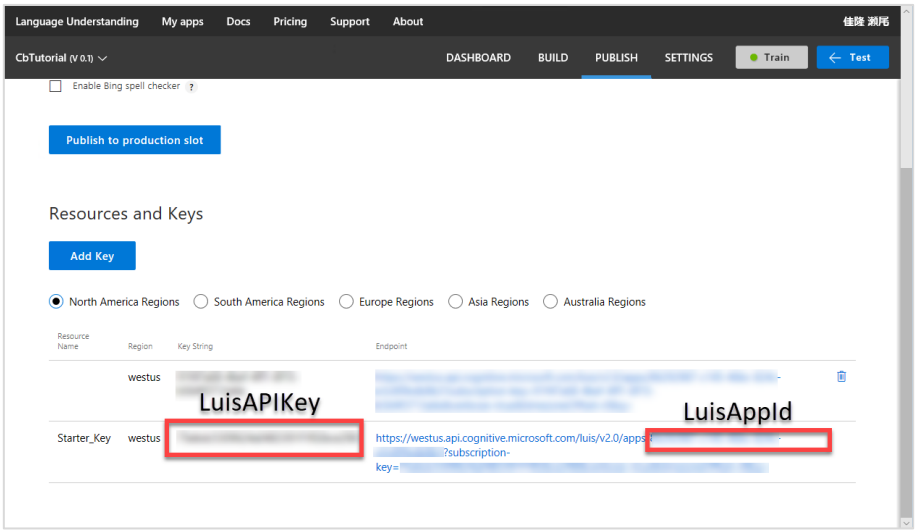
```
private async Task AfterLuisAnswerAsync(IDialogContext context, IAwaitable<object> result)
{
    context.Wait(MessageReceivedAsync);
}
```

10. Web.config を開き、appSettings の中に LuisAPIKey, LuisAppId, LuisAPIHostName を追加します。

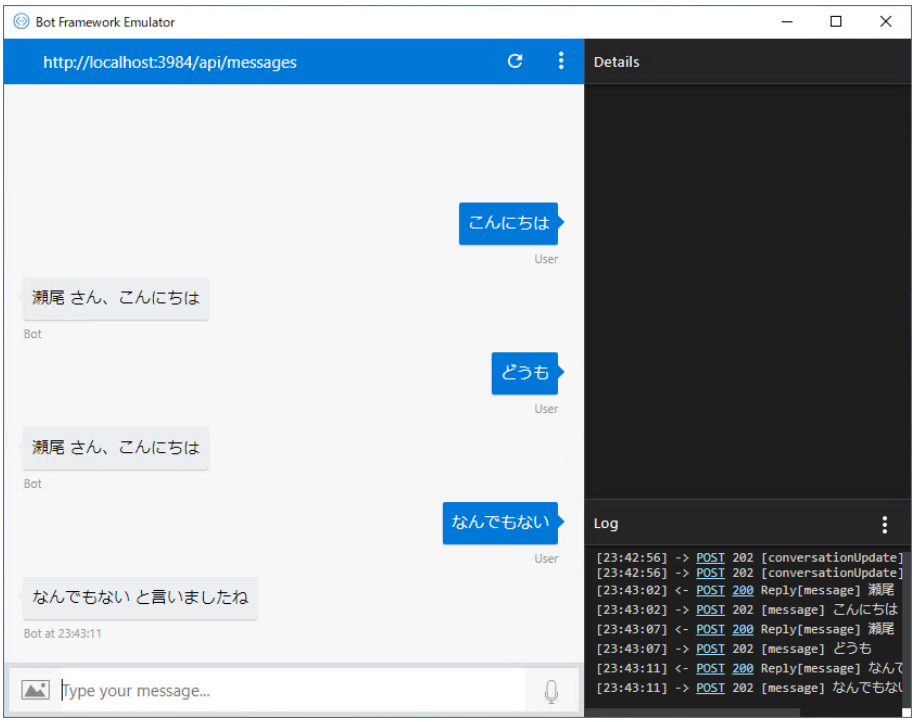
```
<appSettings>
  <!-- update these with your Microsoft App Id and your Microsoft App Password-->
  <add key="MicrosoftAppId" value="" />
  <add key="MicrosoftAppPassword" value="" />
  <add key="AzureWebJobsStorage" value="<接続文字列>" />

  <add key="LuisAPIKey" value="<以下の通り>" />
  <add key="LuisAppId" value="<以下の通り>" />
  <add key="LuisAPIHostName" value="<以下の通り>" />
</appSettings>
```

追加したキーに対する値は、以下の通りです。

key	value
LuisAPIKey	Language Understanding アプリケーションの PUBLISH ページの “Resources and Keys” を参照
LuisAppId	Language Understanding アプリケーションの PUBLISH ページの “Resources and Keys” を参照
LuisAPIHostName	Language Understanding アプリケーションを配置したリージョンに応じた URL。”West US” の場合には ” westus.api.cognitive.microsoft.com”



11. ビルドに成功したらデバッグ実行します。
Bot Framework Emulator で、「こんにちは」、「こんばんは」などの挨拶を入力します。Bot アプリケーションが挨拶を返します。
挨拶以外のメッセージを入力すると、その内容をオウム返しします。



今の時点では、LUIS に十分な例文を登録していないため、ユーザーの意図を適切に汲み取って答えられないことあります。
他の例文を入力していくことで、Language Understanding は“賢く”なっていきます。

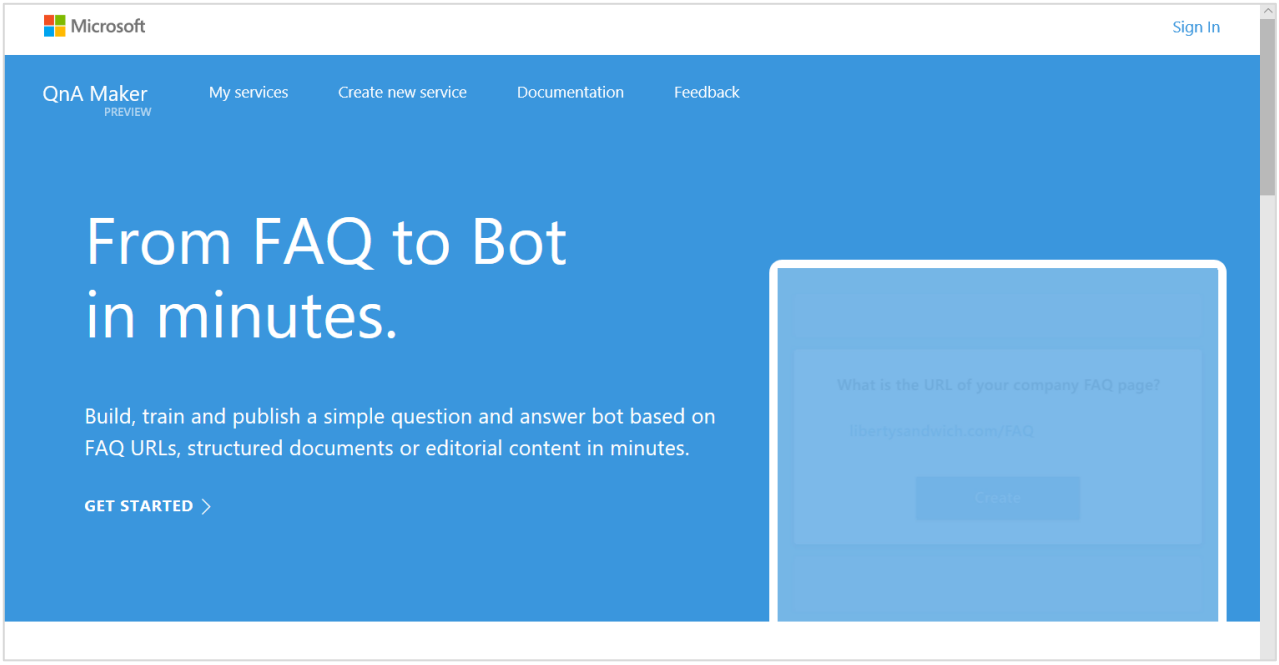
STEP 6. ユーザーの質問に答える

このステップでは、Bot アプリケーションに FAQ 回答機能を追加します。さらに FAQ の回答が見つからない場合は Bing の Web 検索機能を利用してユーザーに応答するようにします。

- ✓ QnA Maker API
- ✓ ナレッジの作成と発行
- ✓ BotBuilder SDK から QnA Maker API の呼び出し
- ✓ Bing Web Search API
- ✓ Bing Web Search API のアクセスキー取得
- ✓ BotBuilder SDK から Bing Web Search API の呼び出し

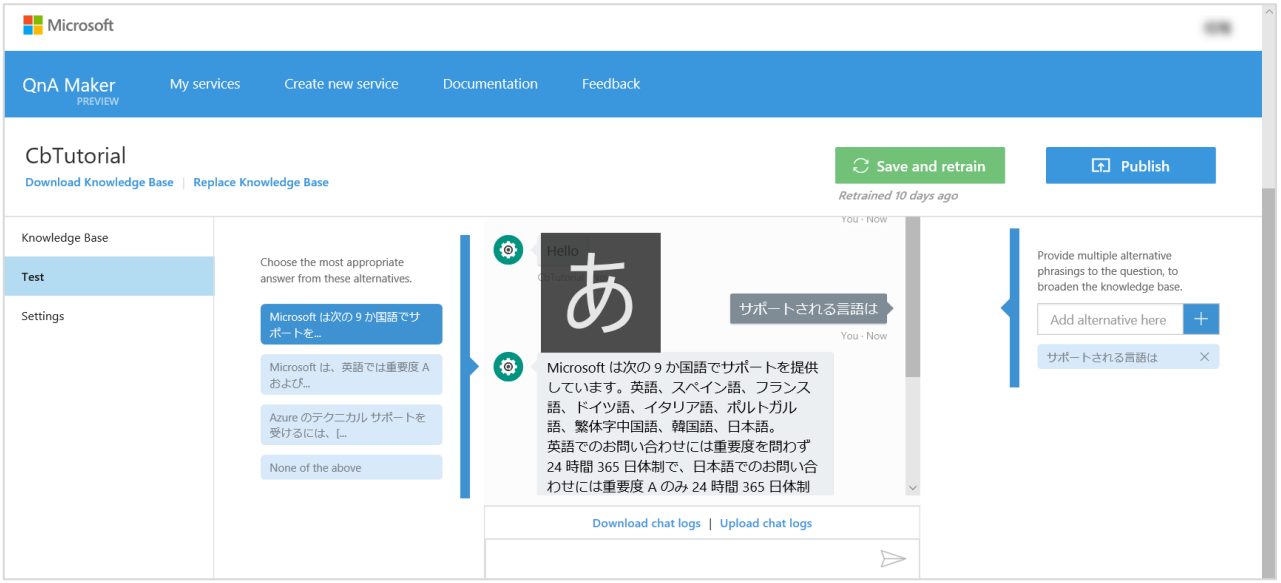
21. QnA Maker API

QnA Maker API (https://qnamaker.ai/) は、FAQ を Bot で実現するためのサービスです。



既存の FAQ ページやファイルをインポートしたり、Excel などで新規の FAQ データを作成したりすることで簡単に FAQ Bot アプリケーションを作成することができます。
インポート後のナレッジ（学習した FAQ）に対して、さらに追加の学習を行うこともできます。

Web ページのインポート、ローカルファイルのインポート、画面で入力する方法は、それぞれ自由に組み合わせることができます。例えば、Web ページを取り込んだ後に、ローカルファイルをインポートする、さらに画面で Q&A を入力して追加する、などの手順を踏むことが可能です。
このようにして、必要に応じてナレッジを強化していきます。



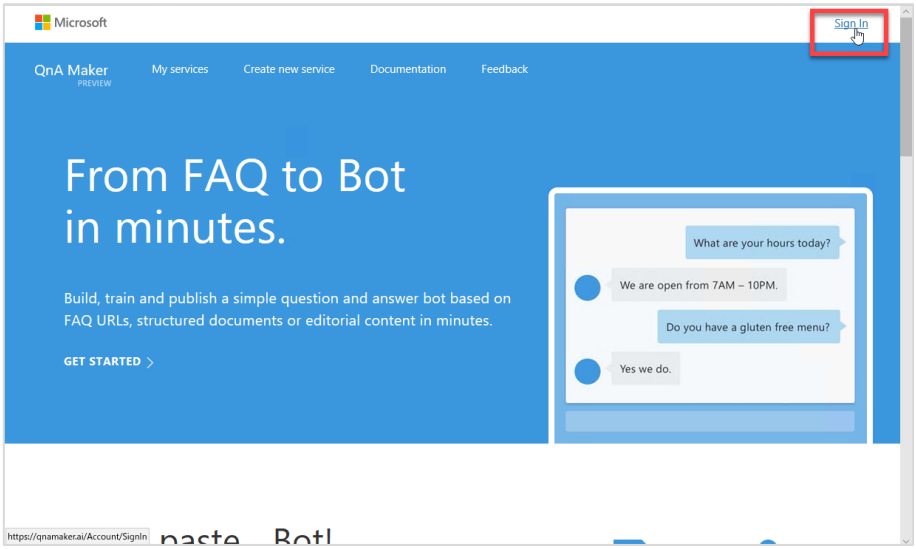
QnA Maker は、Bot Service との連携も考慮されています。

この自習書では多機能の Bot アプリケーションを実装するため、コードの追加変更がありますが、FAQ 専用の Bot の場合は、Bot Service で Bot アプリケーションを作るだけです。ノンコーディングで FAQ Bot が完成します。

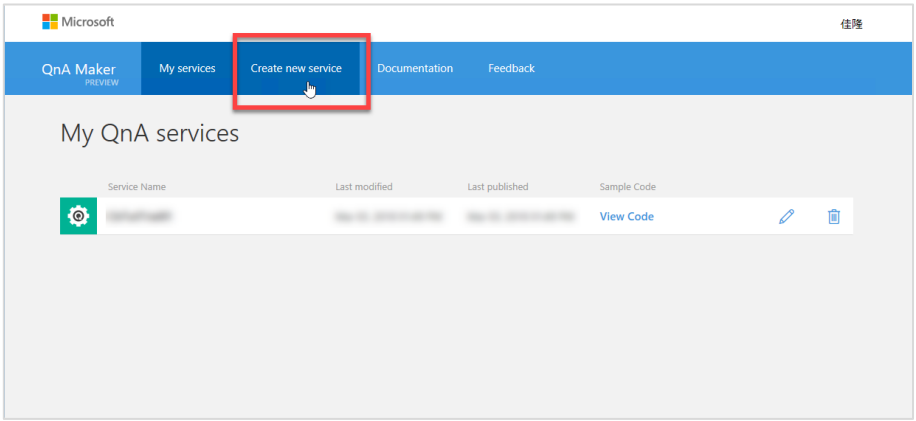
22. ナレッジの作成と発行

QnA Maker を使って、Q&A のナレッジを作成して発行するまでの手順は以下の通りです。

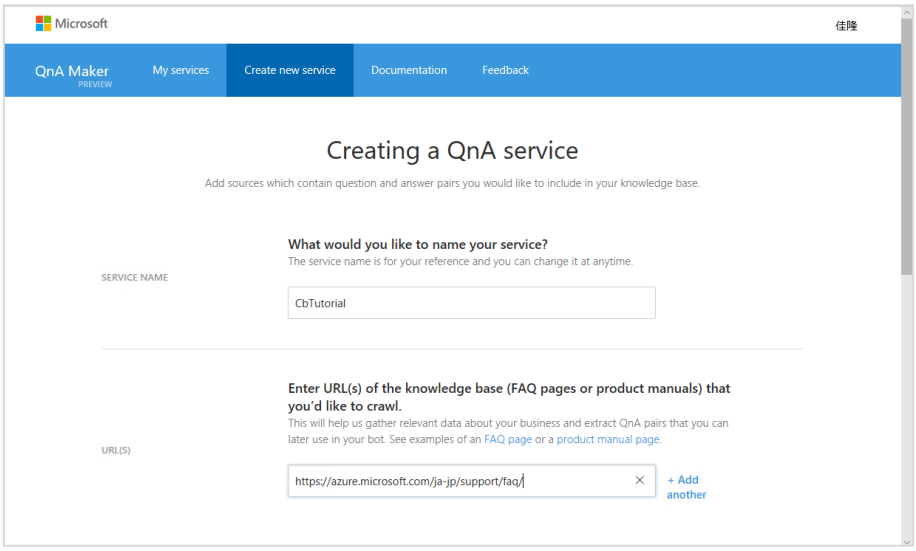
1. QnA Maker のサイト (https://qnamaker.ai/) を Web ブラウザーで開き、サインインします。



2. サインインに成功したら、ページ上部の [Create new service] をクリックしてください。



3. [Create new service] 画面で、以下の値で設定します。すべて入力したら [Create] をクリックします。



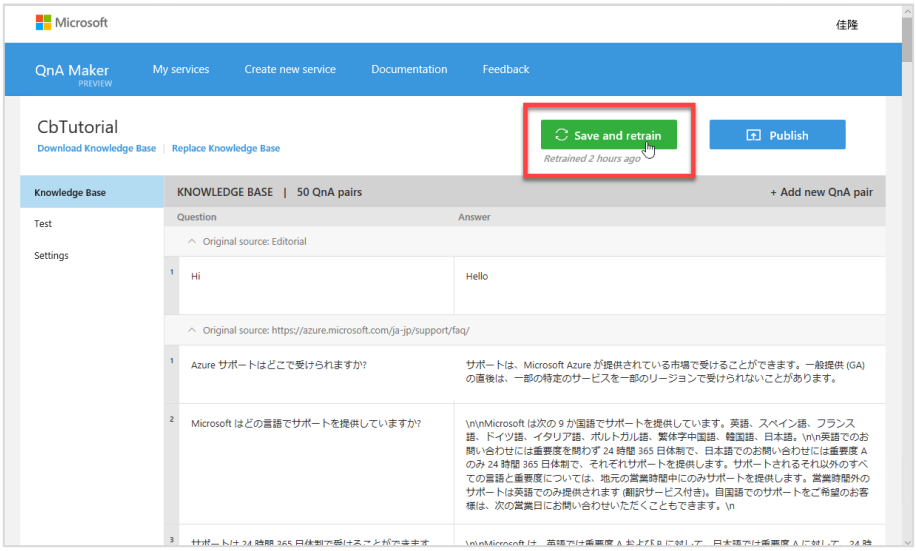
key	value
SERVICE NAME	任意の値
URL(S)	この自習書では、 https://azure.microsoft.com/ja-jp/support/faq/ と入力します。 ※この Web ページは Microsoft Azure の FAQ ページです。

QnA Maker では、既存の Web ページからナレッジを作成することができます。指定した Web ページに Q&A と考えられる構造が見つかった場合、それらをナレッジとして自動的に取り込みます。

ナレッジを作成するには、既存の FAQ ページをインポートするほかに、ローカルに保存したいかのフォーマットのファイルをインポートする方法もあります。

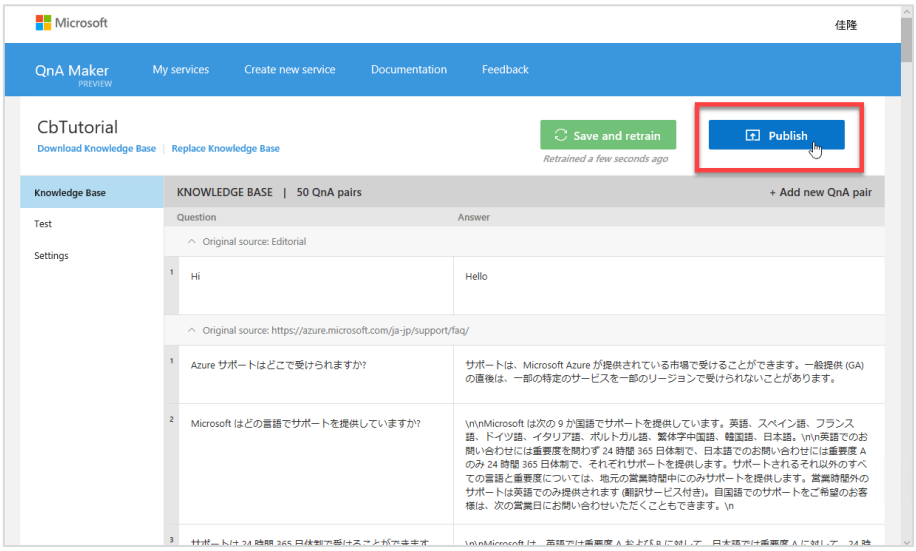
- .tsv
- .pdf
- .doc
- .docx
- .xlsx

4. [Save and retrain] をクリックします。これにより、インポートした Q&A および画面で入力、変更した Q&A を保存して学習します。



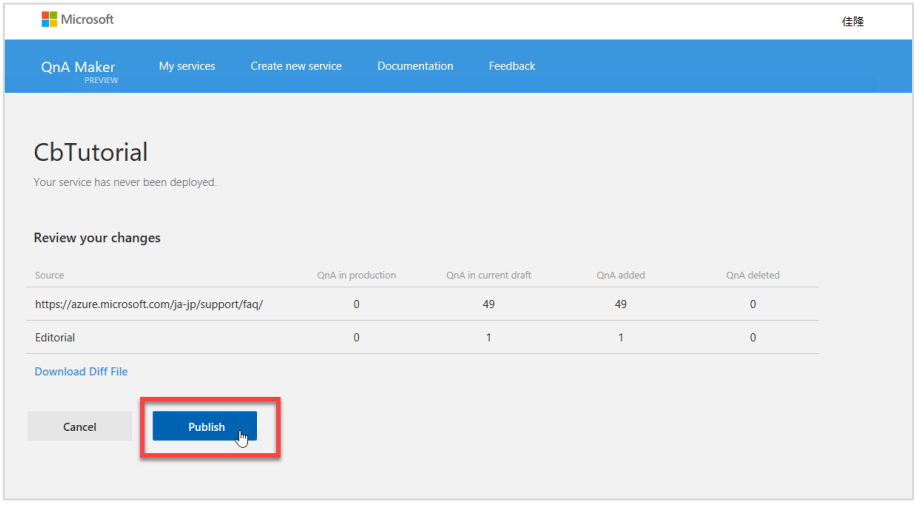
Q&A を編集して [Save and retrain] をクリックすることで、随時ナレッジを更新することができます。

5. 学習に成功したら、[Publish] をクリックして、QnA 発行を始めます。ここではすぐに発行されず、次の手順によって実際に発行が行われます。

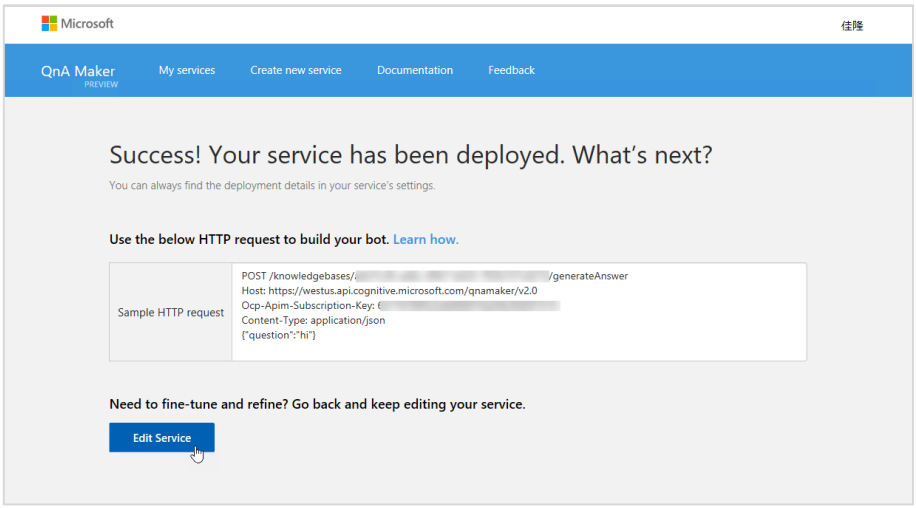


6. Q&A の登録状況、変更内容などが表示されます。

7. [Publish] をクリックします。Q&A が発行されます。



8. 発行に成功すると、結果が表示されます。



QnAKnowledgebaseId および QnASubscriptionKey を含む、サービスの情報が表示されれば、発行が完了しています。

これらの値はあとから [Settings] 画面でも確認できますので、ここではサービスの情報を別の場所に記録しておく必要はありません。

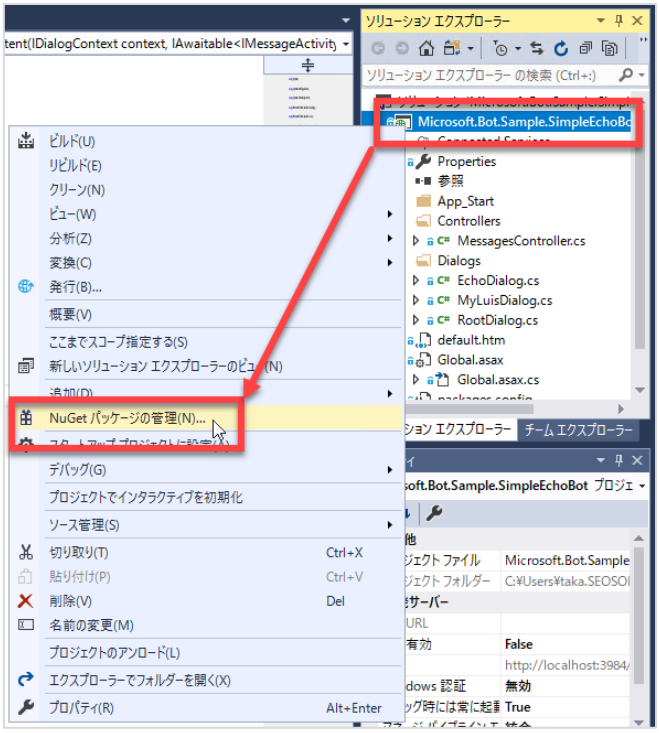
以上で、ナレッジの作成と発行が完了しました。

23. BotBuilder SDK から QnA Maker API の呼び出し

前の手順で、QnA Maker を使ったナレッジの作成と発行が完了しました。
続いて、開発中の Bot アプリケーションから、発行したナレッジを呼び出してみます。

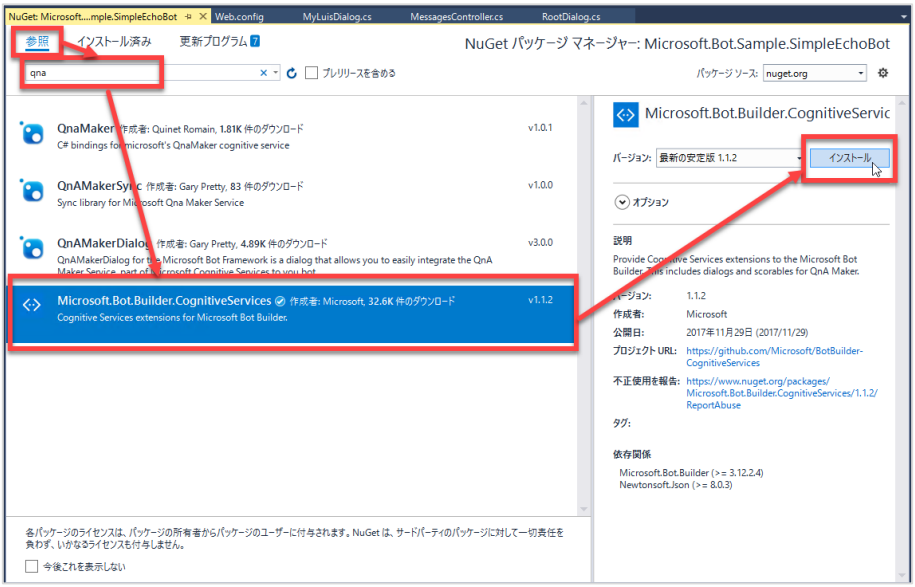
QnA Maker のナレッジは RESTful API に対応しています。この点については、Language Understanding アプリケーションと同様です
この手順書では、QnA Maker API 用の NuGet パッケージを利用して QnA Maker で作ったナレッジを呼び出してみます。

1. Visual Studio 2017 のソリューションエクスプローラーのプロジェクトで右クリックして、[NuGet パッケージの管理] を選択します。

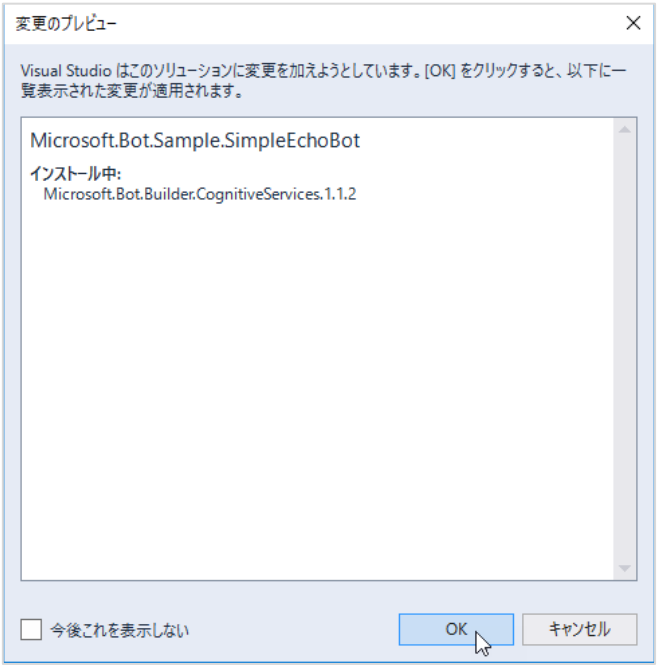


2. NuGet パッケージマネージャーで、[参照] を選択し、テキストボックスに “qna” と入力します。続いて、”Microsoft.Bot.Builder.CognitiveServices” を選択して [インストール] をクリックします。

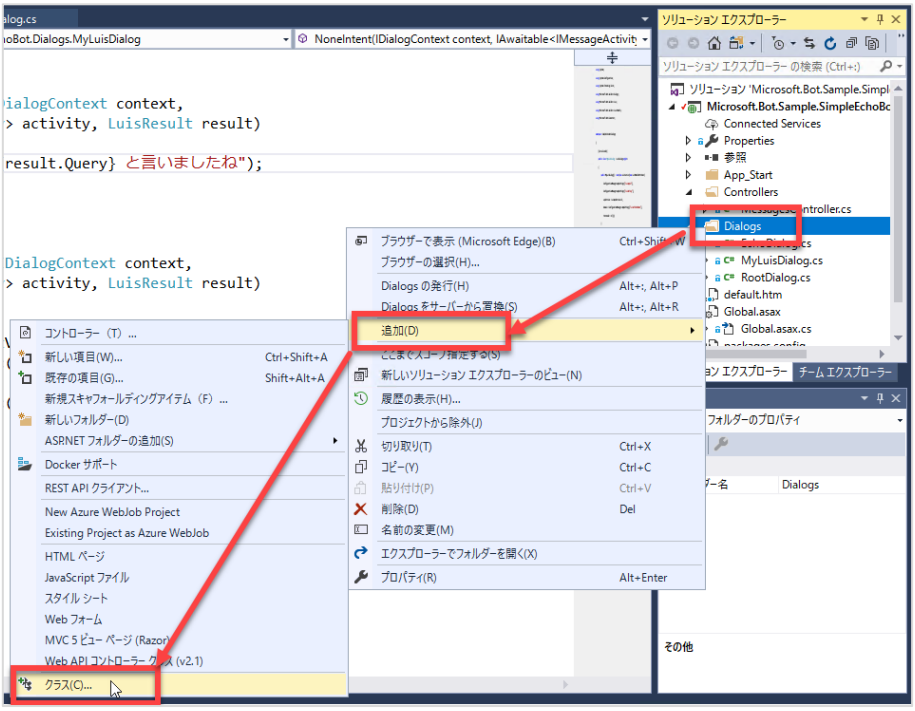
NuGet パッケージマネージャーで “qna” 検索すると複数の候補が見つかります。この手順書ではマイクロソフトが提供する “Microsoft.Bot.Builder.CognitveServices” を使用します。



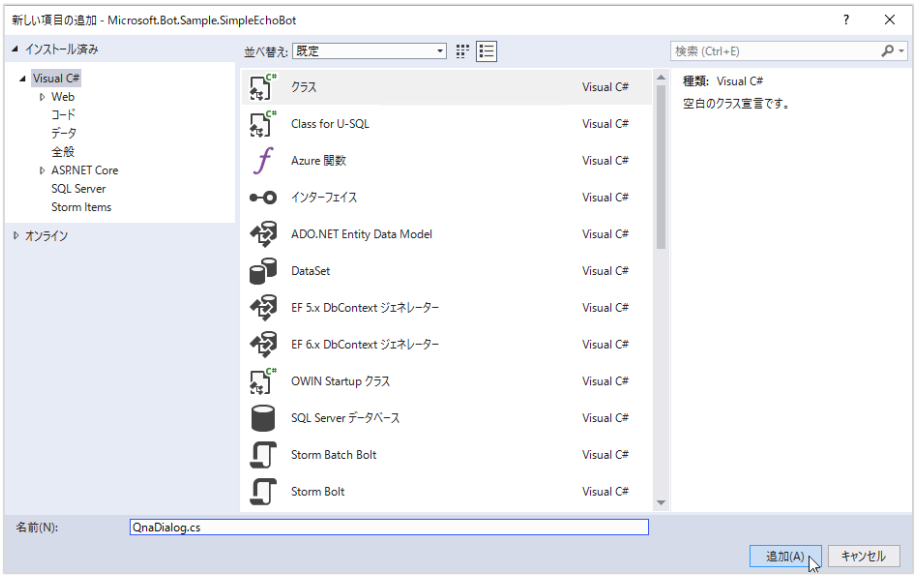
3. [変更のプレビュー] 画面で [OK] をクリックします。QnA Maker API を含む NuGet パッケージがインストールされます。



4. QnA Maker API を利用する Dialog クラスを追加します。ソリューションエクスプローラーの [Dialogs] で右クリック、[追加] –[クラス] を選択します。



5. 名前として “QnaDialog.cs” を指定します。



6. QnaDialog.cs 全体を以下のように変更します。

```
using System;
using System.Configuration;
using System.Threading.Tasks;
using Microsoft.Bot.Builder.CognitiveServices.QnAMaker;
using Microsoft.Bot.Builder.Dialogs;
using Microsoft.Bot.Connector;

namespace SimpleEchoBot.Dialogs
{
    [Serializable]
    public class QnaDialog : QnAMakerDialog
    {
        private bool _qnaAnswered;

        public QnaDialog() : base(new QnAMakerService(new QnAMakerAttribute(
            ConfigurationManager.AppSettings["QnASubscriptionKey"],
            ConfigurationManager.AppSettings["QnAKnowledgebaseId"])))
        {
        }

        protected override async Task
            RespondFromQnAMakerResultAsync(IDialogContext context,
            IMessageActivity message, QnAMakerResults result)
        {
            _qnaAnswered = true;
            await context.PostAsync("Q&A で見つかった回答です");
            await context.PostAsync(result.Answers[0].Answer);
            context.Done(true);
        }

        // 次ページへ
```

```
        // 前ページから

        protected override async Task
            DefaultWaitNextMessageAsync(IDialogContext context,
            IMessageActivity message, QnAMakerResults result)
        {
            if (!_qnaAnswered)
            {
                context.Done(true);
            }
            else
            {
                _qnaAnswered = false;
                context.Wait(MessageReceivedAsync);
            }
        }
    }
}
```

7. MyLuisDialog.cs の先頭付近を以下のように変更します。
"Using System.Threading;" の行を追加しています。

```
using System;
using System.Configuration;
using System.Threading;
using System.Threading.Tasks;
using Microsoft.Bot.Builder.Dialogs;
using Microsoft.Bot.Builder.Luis;
using Microsoft.Bot.Builder.Luis.Models;
using Microsoft.Bot.Connector;

namespace SimpleEchoBot.Dialogs
{
    // 以下、省略
```

8. MyLuisDialog の NoneIntent メソッドを、以下のように変更します。

```
[LuisIntent("None")]
public async Task NoneIntent(IDialogContext context,
    IAwaitable<IMessageActivity> activity, LuisResult result)
{
    var category = result.Entities;
    await context.Forward(new QnaDialog(), AfterQnaAnswerAsync,
        await activity, CancellationToken.None);
}
```

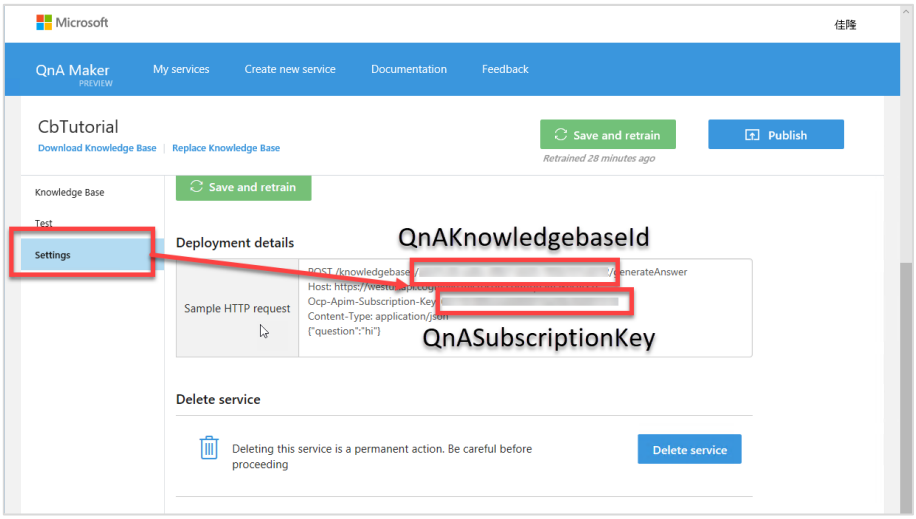
9. MyLuisDialog に、以下の通り AfterQnaAnswerAsync メソッドを追加します。

```
private static async Task AfterQnaAnswerAsync(IDialogContext context, IAwaitable<object> result)
{
    context.Done(true);
}
```

10. Web.config を開き、appSettings の中に QnASubscriptionKey, QnAKnowledgebaseId を追加します。

```
<appSettings>
    <!-- update these with your Microsoft App Id and your Microsoft App Password-->
    <add key="MicrosoftAppId" value="" />
    <add key="MicrosoftAppPassword" value="" />
    <add key="AzureWebJobsStorage" value="<接続文字列>" />

    <add key="LuisAPIKey" value="<設定済み>" />
    <add key="LuisAppId" value="<設定済み>" />
    <add key="LuisAPIHostName" value="<設定済み>" />

    <add key="QnASubscriptionKey" value="<以下の通り>" />
    <add key="QnAKnowledgebaseId" value="<以下の通り>" />
</appSettings>
```

追加したキーに対する値は、QnA Maker の [Settings] 画面で確認することができます。



key	value
QnASubscriptionKey	1 行目の “/knowledgebases/” と “/generateAnswer” との間の値
QnAKnowledgebaseId	“Ocp-Apim-Subscription-Key:” の後の値

11. ビルドに成功したらデバッグ実行します。
ユーザーからのメッセージとそれに対する Bot アプリケーションの応答は以下のようになります。入力するメッセージは全く同じである必要はありません。

- 挨拶

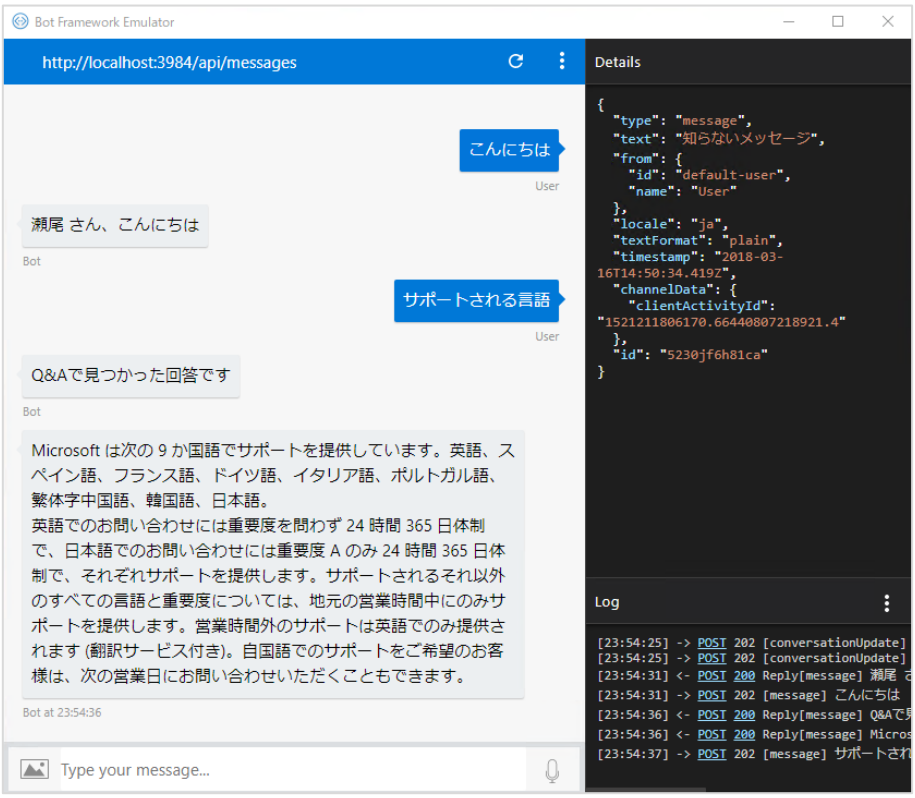
「こんにちは」と入力すると、Bot アプリケーションは挨拶を返します。

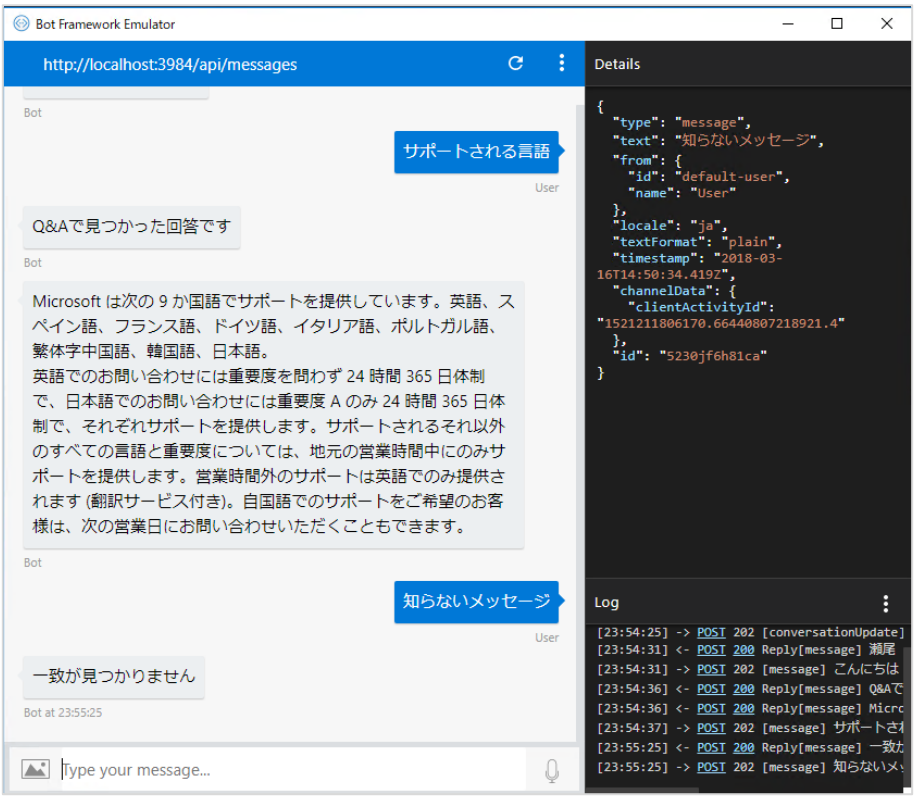
- QnA Maker のナレッジの質問文に似たメッセージ

例えば「サポートされる言語」と入力すると、QnA Maker のナレッジから該当する Answer を返します。

- どちらにも該当しないメッセージ

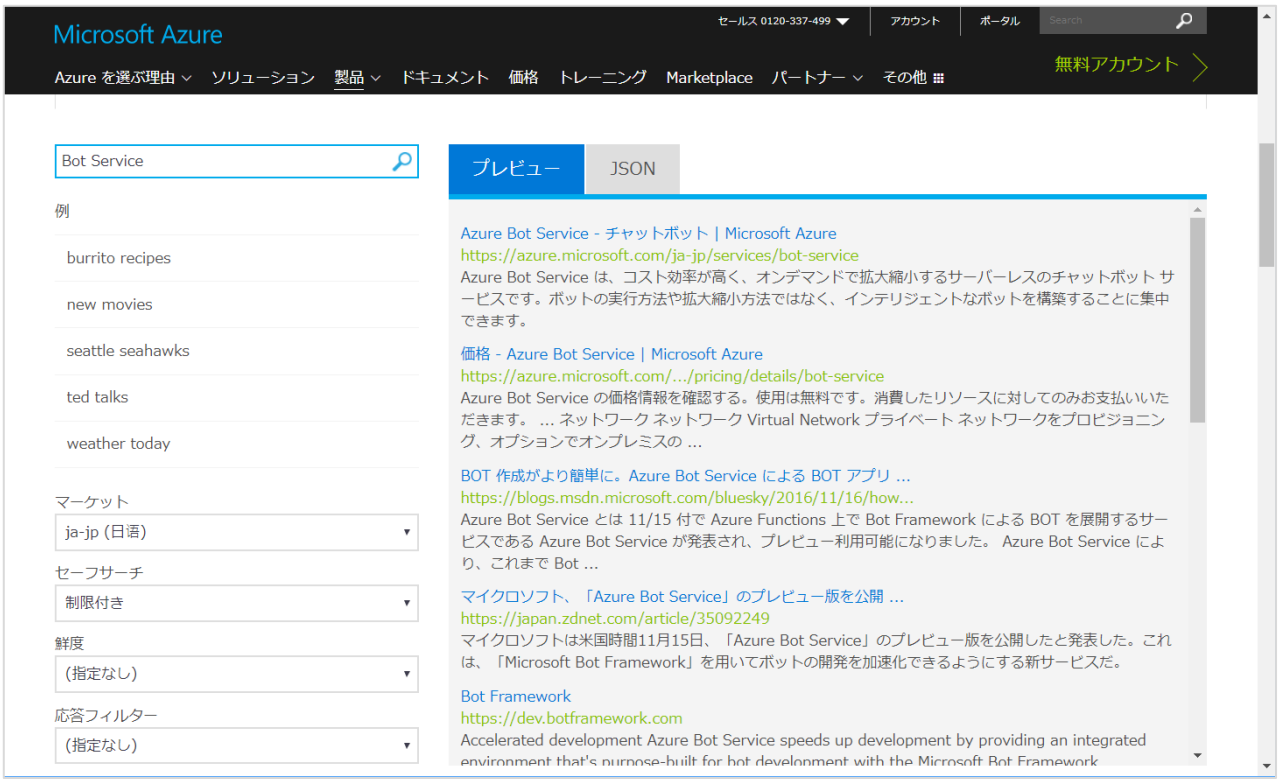
挨拶でも、ナレッジの質問文でもないメッセージを入力した場合は、「一致が見つかりません」というメッセージを返します。このメッセージは QnA Maker API が返す固定メッセージです。
この固定メッセージは変更可能です。この後のステップで紹介します。




Bot Framework Emulator
http://localhost:3984/api/messages
Bot
サポートされる言語
User
Q&Aで見つかった回答です
Bot
Microsoft は次の 9 か国語でサポートを提供しています。英語、スペイン語、フランス語、ドイツ語、イタリア語、ポルトガル語、繁体字中国語、韓国語、日本語。
英語でのお問い合わせには重要度を問わず 24 時間 365 日体制で、日本語でのお問い合わせには重要度 A のみ 24 時間 365 日体制で、それぞれサポートを提供します。サポートされるそれ以外のすべての言語と重要度については、地元の営業時間中にのみサポートを提供します。営業時間外のサポートは英語でのみ提供されます (翻訳サービス付き)。自国語でのサポートをご希望のお客様は、次の営業日にお問い合わせいただくこともできます。
Bot
知らないメッセージ
User
一致が見つかりません
Bot at 23:55:25
Type your message...
Details
{
"type": "message",
"text": "知らないメッセージ",
"from": {
"id": "default-user",
"name": "User"
},
"locale": "ja",
"textFormat": "plain",
"timestamp": "2018-03-16T14:50:34.419Z",
"channelData": {
"clientActivityId": "1521211806170.66440807218921.4"
},
"id": "5230jf6h81ca"
}
Log
[23:54:25] -> POST 202 [conversationUpdate]
[23:54:31] <- POST 200 Reply[message] 瀬尾
[23:54:31] -> POST 202 [message] こんにちは
[23:54:36] <- POST 200 Reply[message] Q&Aで
[23:54:36] <- POST 200 Reply[message] Micro
[23:54:37] -> POST 202 [message] サポートされ
[23:55:25] <- POST 200 Reply[message] 一致が
[23:55:25] -> POST 202 [message] 知らないメッ

24. Bing Web Search API

Bing Web Search API (https://azure.microsoft.com/ja-jp/services/cognitive-services/bing-web-search-api/) は、アプリケーションに Bing Web 検索の機能を追加するものです。

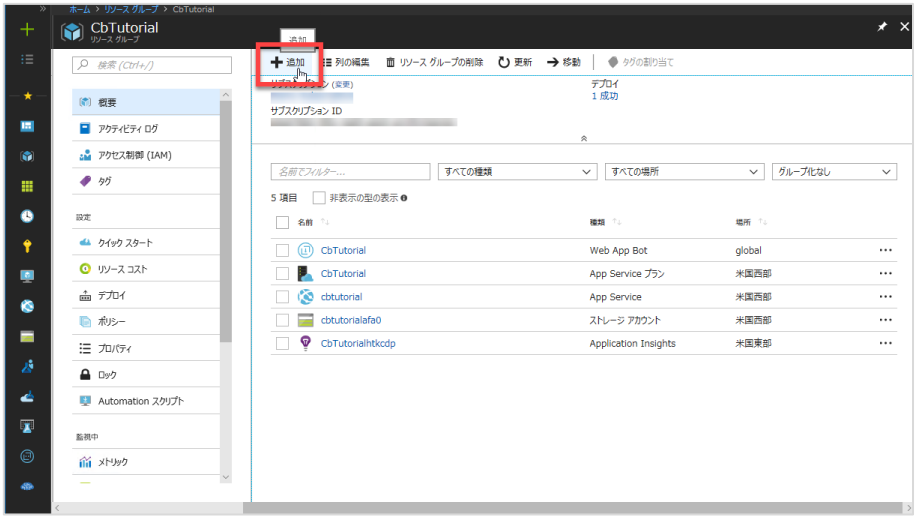


検索結果をランク付けする、位置情報をもとに検索結果を絞り込む、セーフサーチ（適切ではない結果を対象外とする）、スペルミス時の提案などの機能も利用できます。

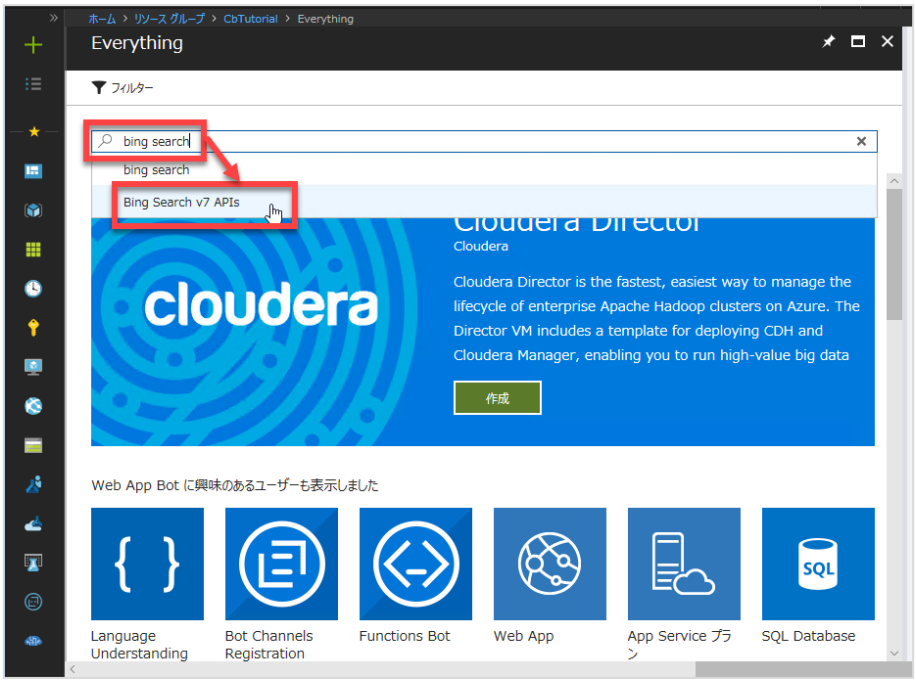
25. Bing Web Search API のアクセスキー取得

Bing Web Search API はアクセスキーの取得のみで利用できます。
ここでは、Bing Web Search API のアクセスキーの取得方法を紹介します。

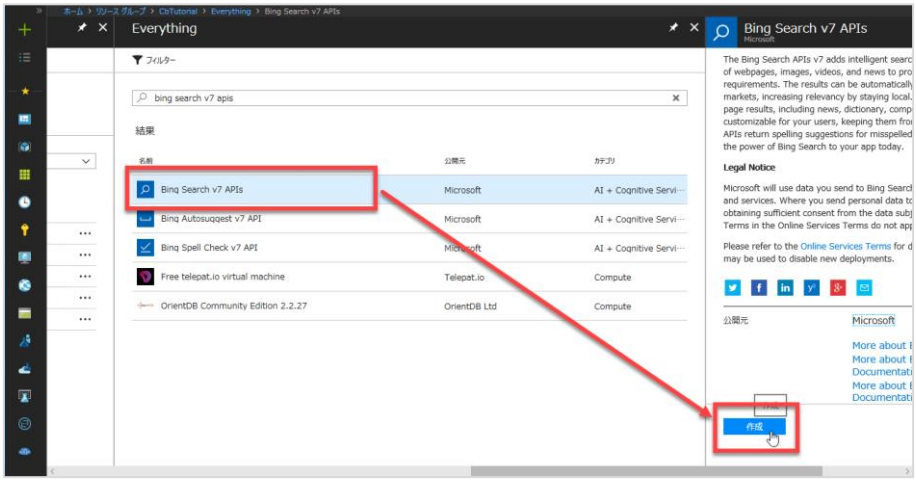
1. Azure 管理ポータルで Bot Service を作ったリソースグループを開き、[+追加] をクリックします。



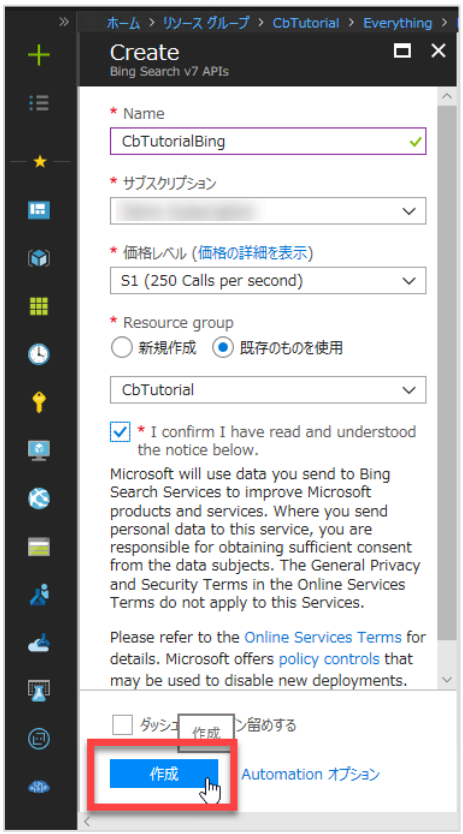
2. “bing search” と入力すると候補が表示されます。“Bing Search v7 APIs” をクリックします。



3. [Everything] ブレードで “Bing Search v7 APIs” をクリックし、新しいブレードが開いたら [作成] をクリックします。

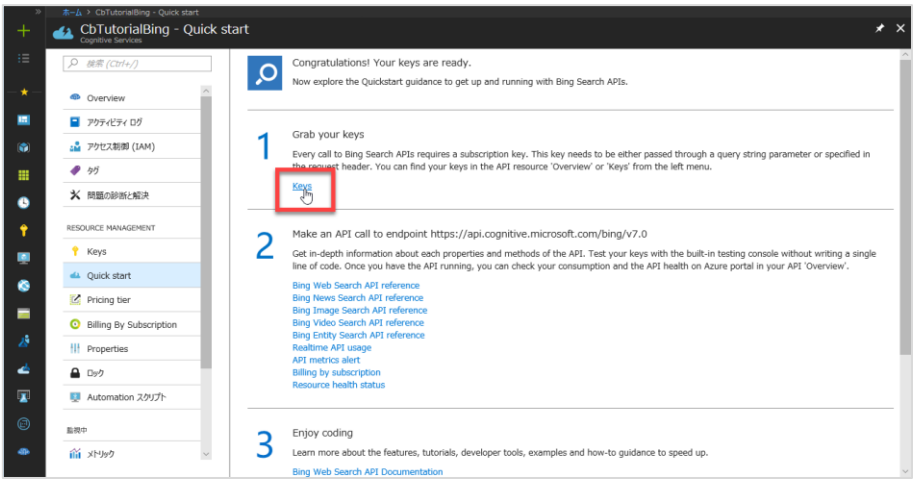


4. [Create] ブレードが開いたら、以下の値で設定します。内容を確認したら [作成] をクリックします。



設定項目	設定値
Name	任意の値
サブスクリプション	Bot Service を作ったものと同じサブスクリプション
価格レベル	S1
Resource Group	既存のものを使用。 ドロップダウンリストでは、Bot Service と同じリソースグループ
(チェックボックス)	“I confirm ～” をチェック

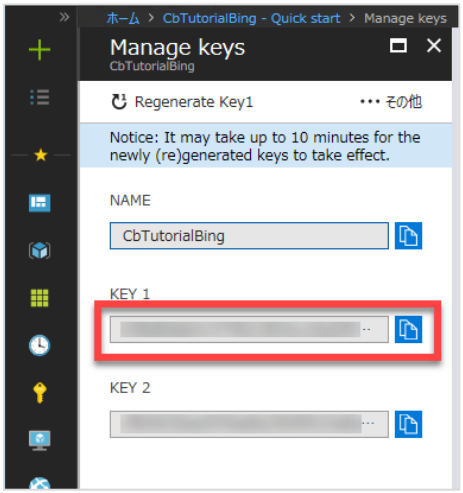
5. Bing Search API が作成されたら、[Key] をクリックします。



6. Bing Search API のアクセスキーが表示されます。

アクセスキーは、開発中の Bot アプリケーションで Bing Web Search API を呼び出す際に利用します。

このページはこのまま開いておきます。

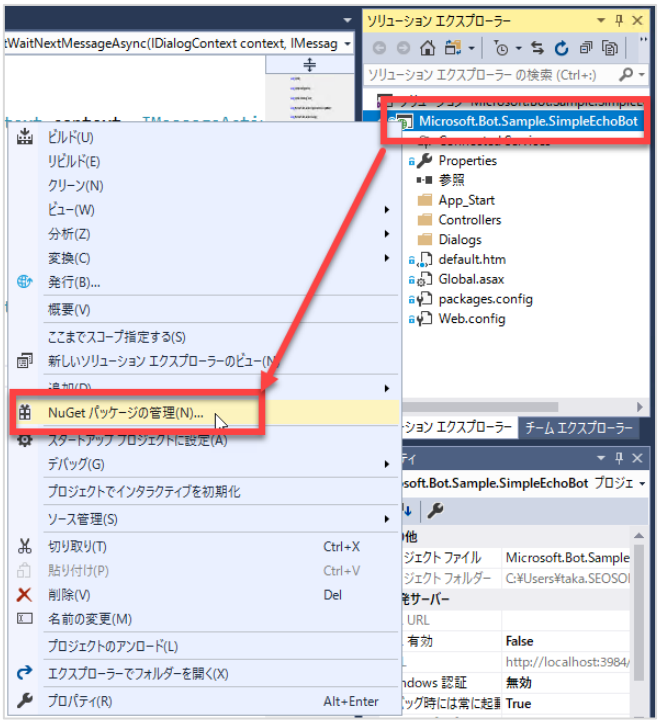


26. BotBuilder SDK から Bing Web Search API の呼び出し

開発中の Bot アプリケーションから、Bing Web Search API を呼び出してみます。

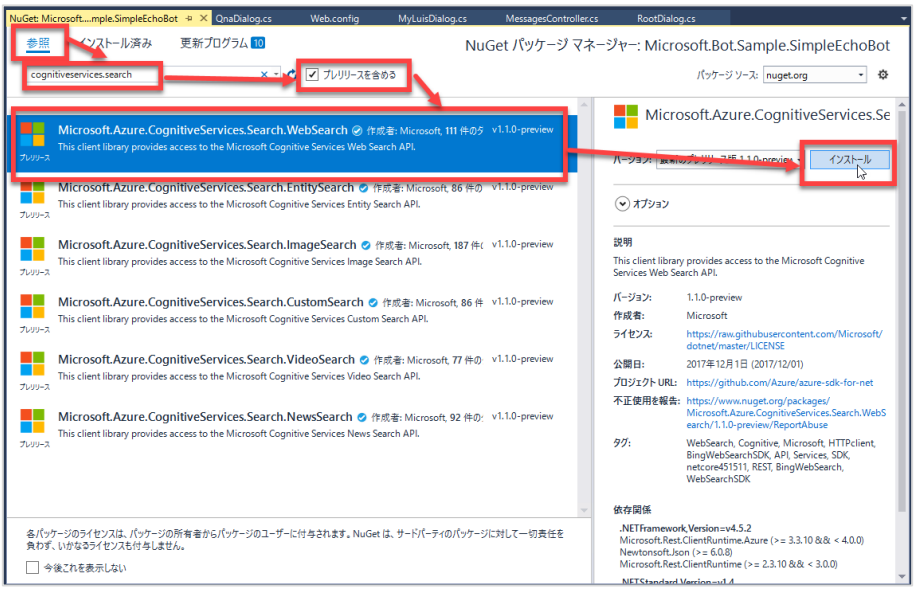
Bot アプリケーションから Bing Web Search API を呼び出すにはキーが必要です。このキーは前の手順で取得しました。
以下の手順では、NuGet パッケージの利用方法を紹介します。

1. Visual Studio 2017 のソリューションエクスプローラーのプロジェクトで右クリックして、[NuGet パッケージの管理] を選択します。

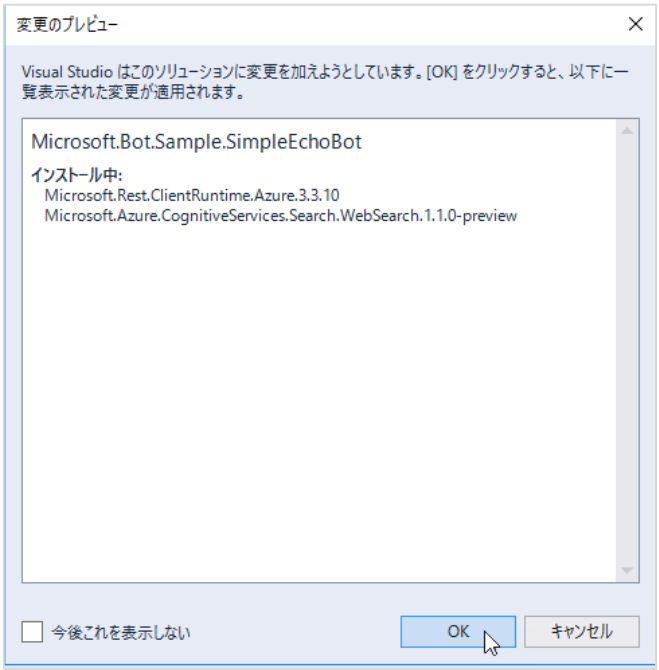


2. NuGet パッケージマネージャーで [参照] を選択し、テキストボックスに “cognitiveservices.search” と入力します。

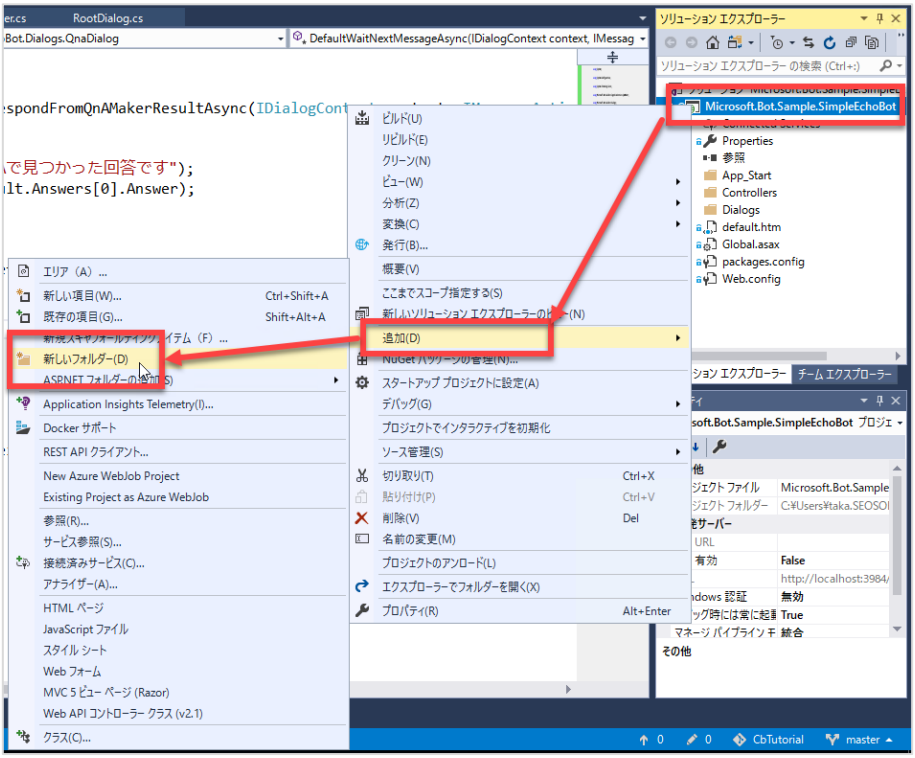
3. 現在、Bing Search API の NuGet パッケージはプレビュー版です。[プレリリースを含める] をチェックしてパッケージを検索します。”Microsoft.Azure.CognitiveServices.Search.WebSearch” を選択して [インストール] をクリックします。



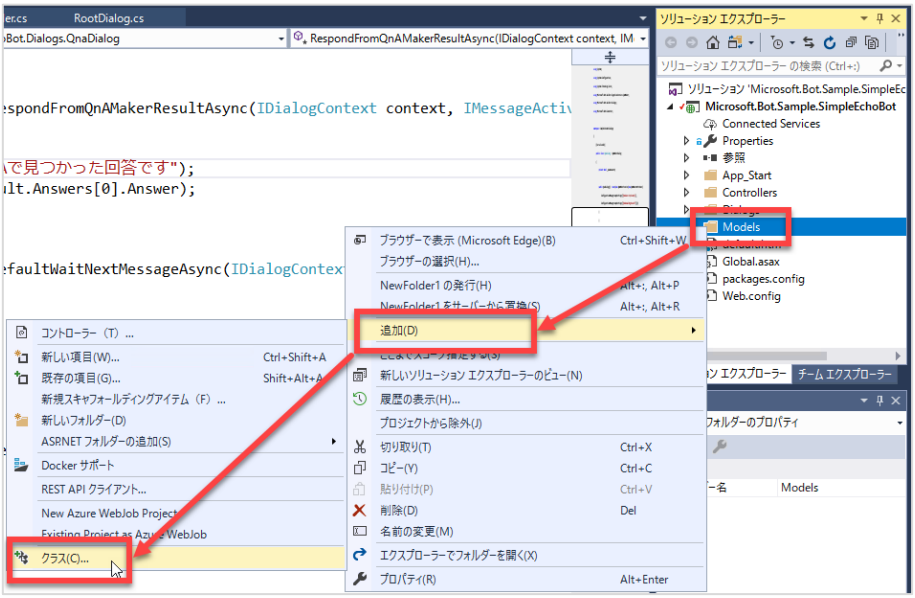
4. [変更のプレビュー] 画面で [OK] をクリックします。Bing Search API を含む NuGet パッケージがインストールされます。



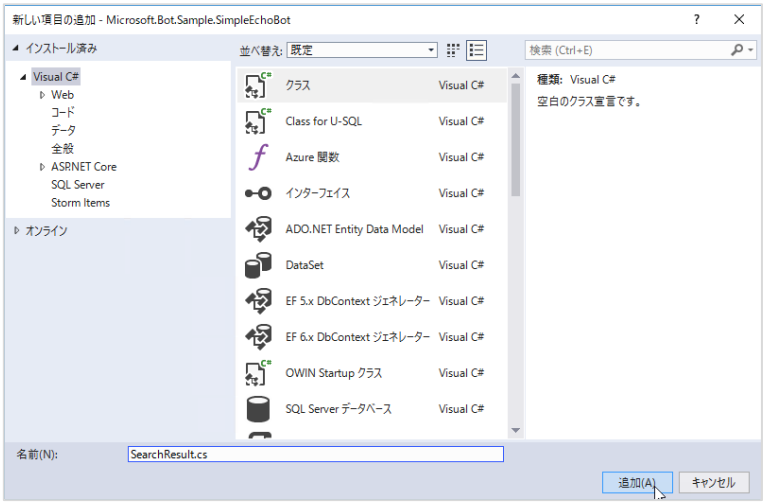
5. Visual Studio 2017 のソリューションエクスプローラーのプロジェクトで右クリックして、[追加] – [新しいフォルダー] を選択します。フォルダーができたら “Models” に名前を変更します。



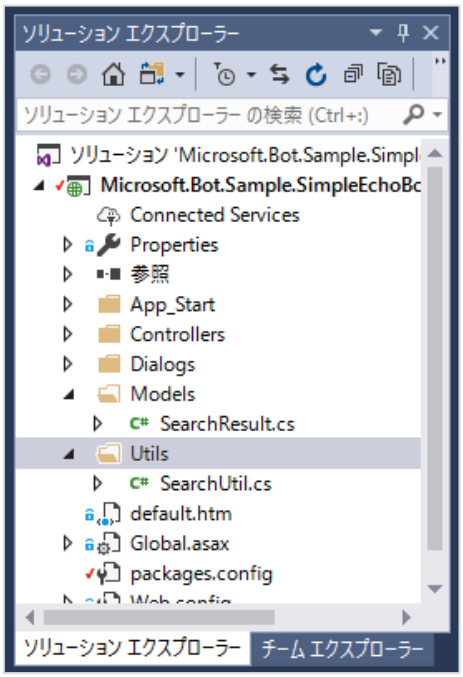
6. 今作った “Models” フォルダーで右クリックして、[追加] – [クラス] を選択します。



7. 名前として“SearchResult.cs”を指定します。



8. 同様の手順で、“Utils”フォルダーを作り、その中に“SearchUtil.cs”ファイルを作成します。



9. SearchResult.cs 全体を以下のように変更します。

```
namespace SimpleEchoBot.Models
{
    public class SearchResult
    {
        public string Title { get; set; }
        public string Description { get; set; }
        public string Url { get; set; }
    }
}
```

10. SearchUtil.cs 全体を以下のように変更します。

```
using System.Collections.Generic;
using System.Configuration;
using System.Linq;
using System.Threading.Tasks;
using Chronic;
using Microsoft.Azure.CognitiveServices.Search.WebSearch;
using SimpleEchoBot.Models;
using ApiKeyServiceClientCredentials
    = Microsoft.Azure.CognitiveServices.Search.WebSearch.ApiKeyServiceClientCredentials;

namespace SimpleEchoBot.Utils
{
    public static class SearchUtil
    {
        public static async Task<IEnumerable<SearchResult>> SearchWebPagesAsync(string
searchTerm)
        {
            var accessKey = ConfigurationManager.AppSettings["BingSearchAccessKey"];
            var client = new WebSearchAPI(new ApiKeyServiceClientCredentials(accessKey));

            var searchResults = new List<SearchResult>();

            var webData = await client.Web.SearchAsync(searchTerm);
            webData.WebPages.Value.Take(3).ForEach(p => searchResults.Add(
                    new SearchResult { Title = p.Name, Description = p.Snippet, Url = p.Url }));

            return searchResults;
        }
    }
}
```

この自習書では、Bing で検索した結果の上位 3 件を返すようにします。"Take(3)" の箇所がそれにあたります。

11. QnaDialog.cs の先頭付近を以下のように変更します。
“using SimpleEchoBot.Utils;” の行を追加しています。

```
using System;
using System.Configuration;
using System.Threading.Tasks;
using Chronic;
using Microsoft.Bot.Builder.CognitiveServices.QnAMaker;
using Microsoft.Bot.Builder.Dialogs;
using Microsoft.Bot.Connector;
using SimpleEchoBot.Utils;

namespace SimpleEchoBot.Dialogs
{
    // 以下、省略
```

12. QnaDialog.cs を開き、コンストラクターを以下のように変更します。
QnA Maker のナレッジで指定のスコア以上の回答が得られなかった場合にユーザーに返すメッセージ、およびスコアのしきい値を指定する属性を追加しています。

```
public QnaDialog() : base(new QnAMakerService(new QnAMakerAttribute(
    ConfigurationManager.AppSettings["QnASubscriptionKey"],
    ConfigurationManager.AppSettings["QnAKnowledgebaseId"],
    "Q&A で見つからないため、Bing で検索します", 0.5)))
{
}
```

13. QnaDialog クラスの最後に、DefaultWaitNextMessageAsync メソッドを追加します。

```
protected override async Task DefaultWaitNextMessageAsync(IDialogContext context,
    IMessageActivity message, QnAMakerResults result)
{
    if (!_qnaAnswered)
    {
        var searchResults = await SearchUtil.SearchWebPagesAsync(message.Text);
        searchResults.ForEach(async r => await context.PostAsync(
            $"{r.Title}¥r¥n{r.Description}¥r¥n{r.Url}"));

        context.Done(true);
    }
    else
    {
        _qnaAnswered = false;
        context.Wait(MessageReceivedAsync);
    }
}
```

14. Web.config を開き、appSettings の中に BingSearchAccessKey を追加します。
以下の“<前の手順で取得したキー1>”の箇所には、Bing Search API を作成した際に表示された“KEY1”の値で置換します。

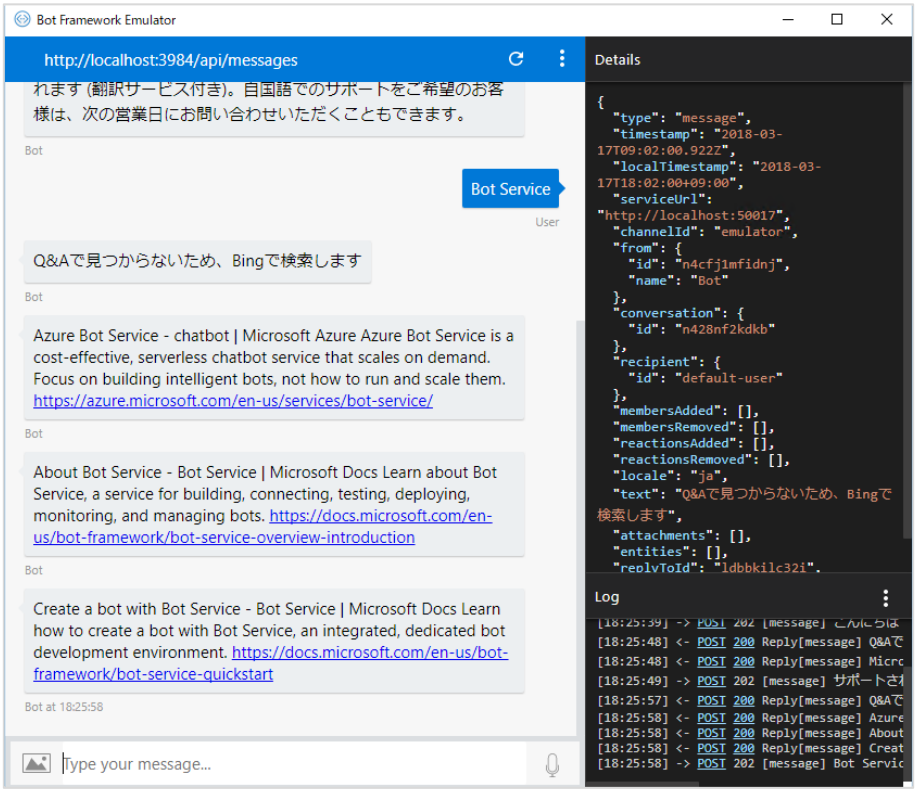
```
<appSettings>
  <!-- update these with your Microsoft App Id and your Microsoft App Password-->
  <add key="MicrosoftAppId" value="" />
  <add key="MicrosoftAppPassword" value="" />
  <add key="AzureWebJobsStorage" value="<接続文字列>" />

  <add key="LuisAPIKey" value="<設定済み>" />
  <add key="LuisAppId" value="<設定済み>" />
  <add key="LuisAPIHostName" value="<設定済み>" />

  <add key="QnASubscriptionKey" value="<設定済み>" />
  <add key="QnAKnowledgebaseId" value="<設定済み>" />
  <add key="BingSearchAccessKey" value="<前の手順で取得した Bing Search API のキー 1>" />
</appSettings>
```

15. ビルドに成功したらデバッグ実行します。

挨拶、および QnA Maker ナレッジでヒットする回答については、これまでと同じように動作します。

ユーザーがどちらでもないメッセージを入力した場合は、Bing で検索した結果を返します。

例えば「Bot Service」と入力すると結果が分かります。



STEP 7. よりリッチな応答をする

このステップでは、ユーザーに負担をかけずに情報収集する手段を実装します。さらにカード表示を使用して、チャットクライアントの UI を改善します。

- ✓ Language Understanding の拡張
- ✓ FormFlow
- ✓ FormFlow の実装
- ✓ Thumbnail Card の実装

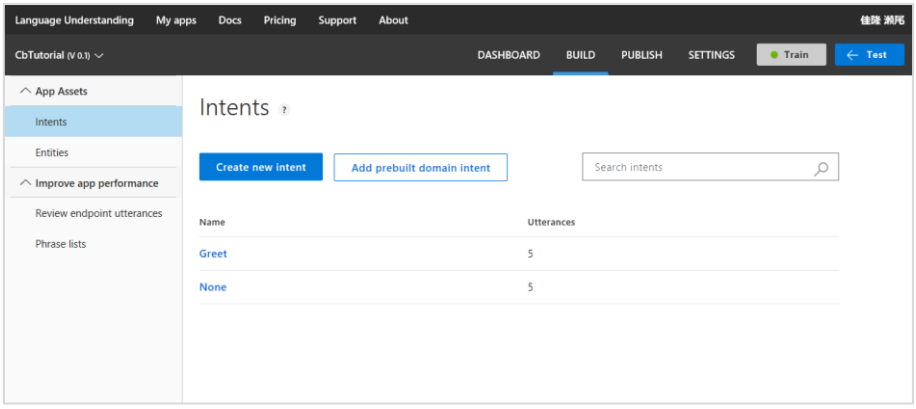
27. Language Understanding の拡張

このステップでは、以下の拡張を行います。

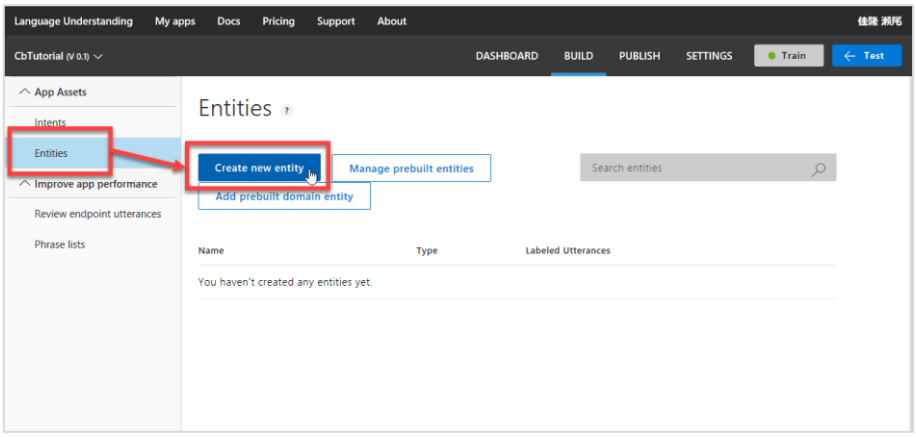
- Language Understanding に Intent を追加して、Bot アプリケーションに機能を追加する準備
- FormFlow でユーザーからの情報収集の操作性を向上
- ユーザーから収集した情報を Azure Storage に保存
- Thumbnail Card で Bot からの応答の見た目を改善

Language Understanding に Intent を追加するところから始めます。

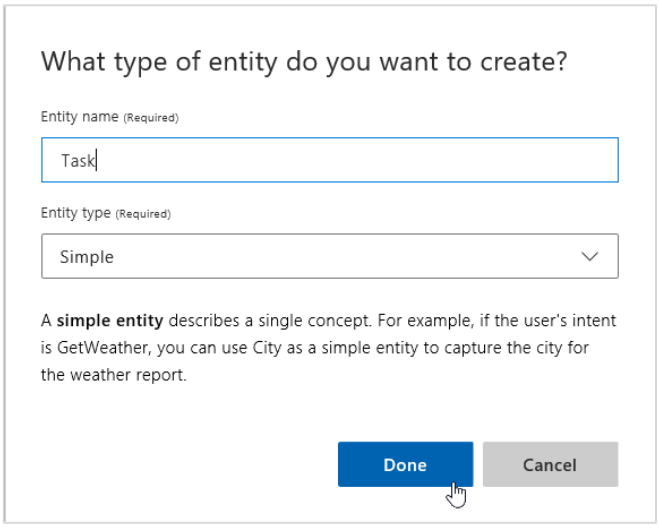
1. Language Understanding のサイト (https://www.luis.ai/home) でサインインし、この手順書で作った Language Understanding アプリケーションを開きます。



2. [Entities] を選択して [Create new entity] をクリックします。

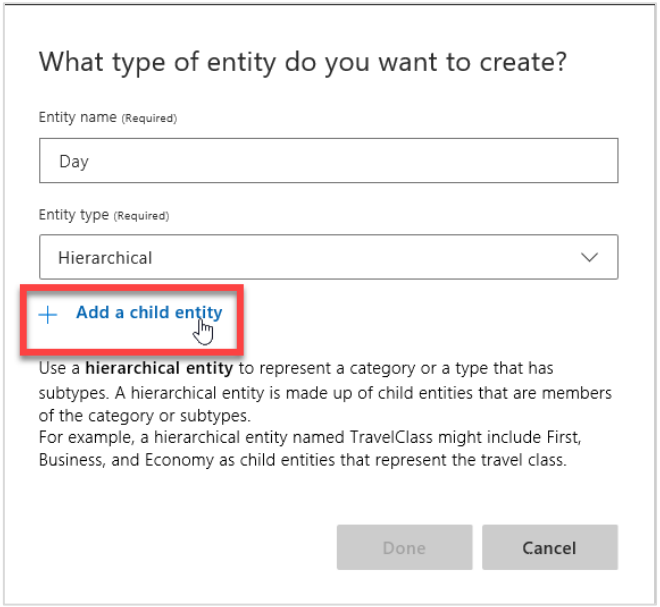


3. Entity の作成画面で、以下の値で設定します。入力が終わったら [Done] をクリックします。



設定項目	設定値
Entity name	Task
Entity Type	Simple

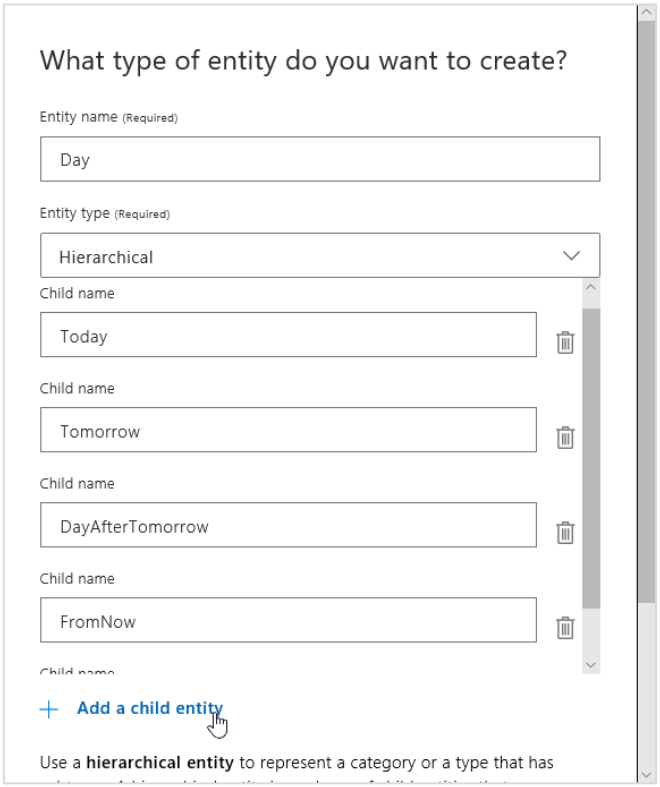
4. もう一度 [Entities] を選択して [Create new entity] をクリックします。入力は以下の通りです。



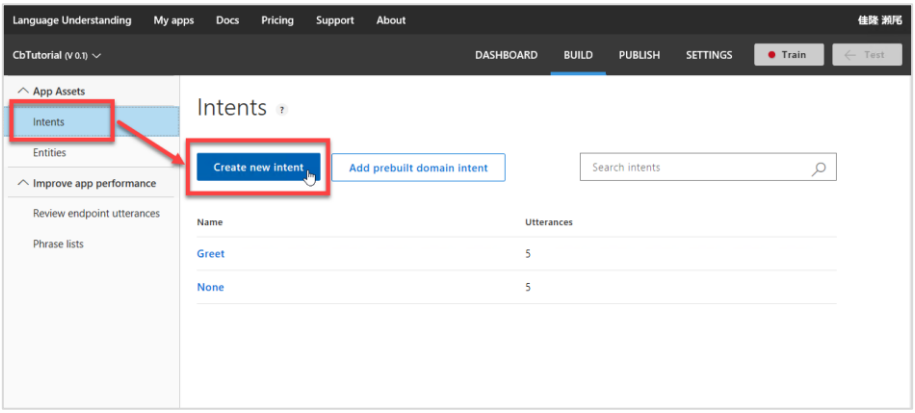
設定項目	設定値
Entity name	Day
Entity Type	Hierarchical

5. [Add a child entity] をクリックして以下の入力を行います。
子エンティティは 1 個ずつ入力することができます。[Add a child Entity] を繰り返しながら、今回は以下の 5 個のエンティティを登録します。
最後に [Done] をクリックして、Day エンティティと子エンティティを保存します。

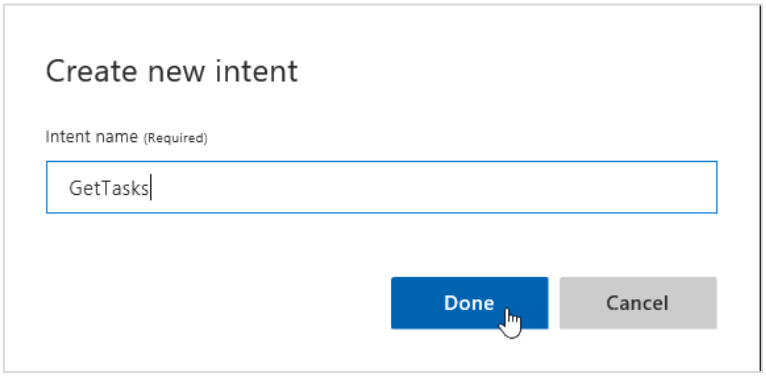
- Today
- Tomorrow
- DayAfterTomorrow
- FromNow
- Yesterday



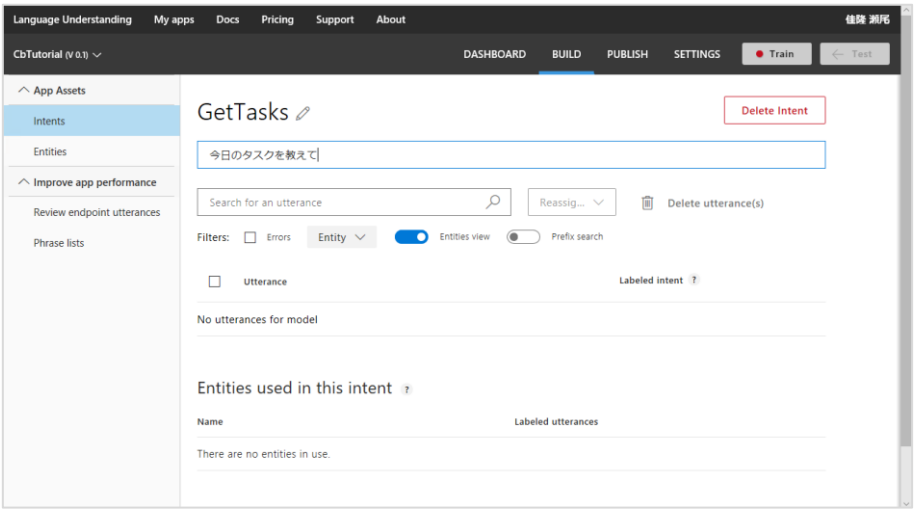
6. [Intents] を選択して [Create new intent] をクリックします。



7. [Create new intent] 画面で、"GetTasks" と入力します。



8. [GetTasks] 画面で、例文を入力します。
最初に「今日の作業を教えて」と入力して Enter キーを押します。

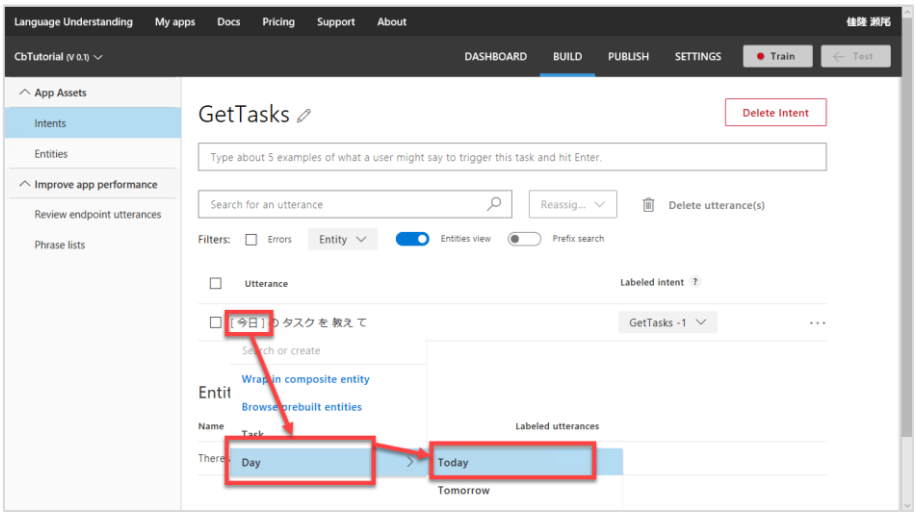


9. 入力した「今日のタスクを教えて」が入力済みの例文として表示されます。ここで “[今日]” の部分をクリックして、[Day] を選択、次に [Today] を選択します。

この操作により、

- 「今日のタスクを教えて」が GetTasks インテントの例文となり
- “今日” は “Day” エンティティの “Today” 子エンティティである

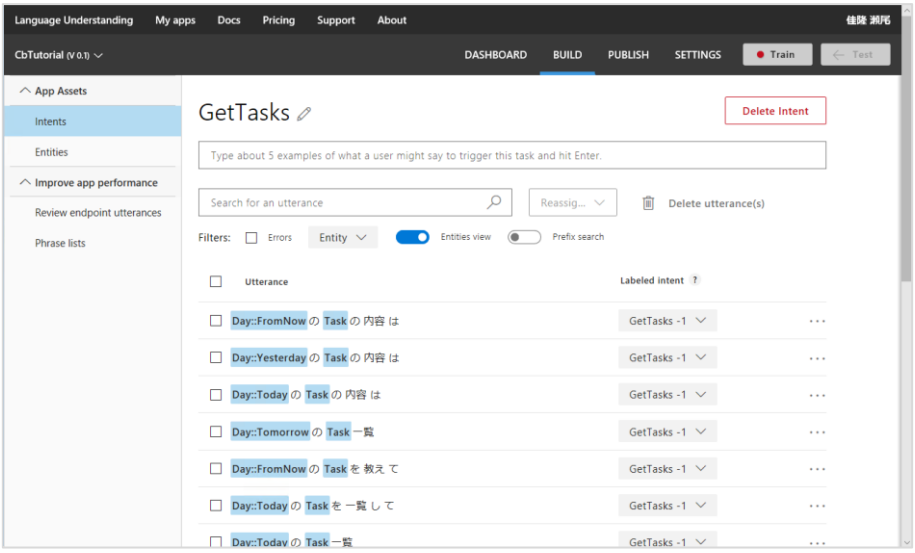
以上の内容を指定したことになります。



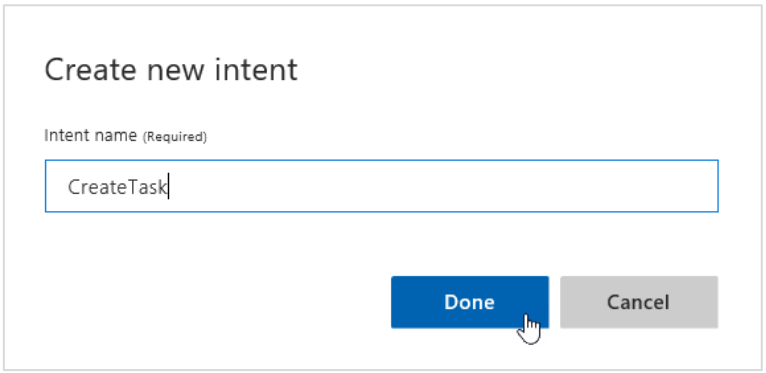
10. 同様に、以下の例文を入力します。

例文	Entity の指定
今日のタスクを教えて	今日→Day::Today タスク→Task
明日の作業は何かな	明日→Day::Tomorrow 作業→Task
明後日のタスクは何	明後日→Day::DayAffterTomorrow タスク→Task
今日のタスク一覧	今日→Day::Today タスク→Task
今日の作業を一覧して	今日→Day::Today 作業→Task
今後の作業を教えて	今後→Day::FromNow 作業→Task
昨日の作業一覧	昨日→Day::Yesterday 作業→Task
今日のタスクの内容は	今日→Day::Today タスク→Task
昨日の作業の内容は	昨日→Day::Yesterday 作業→Task
今後の予定の内容は	今後→Day::FromNow 予定→Task

例文の入力とエンティティの指定が完了すると、以下のようになります。



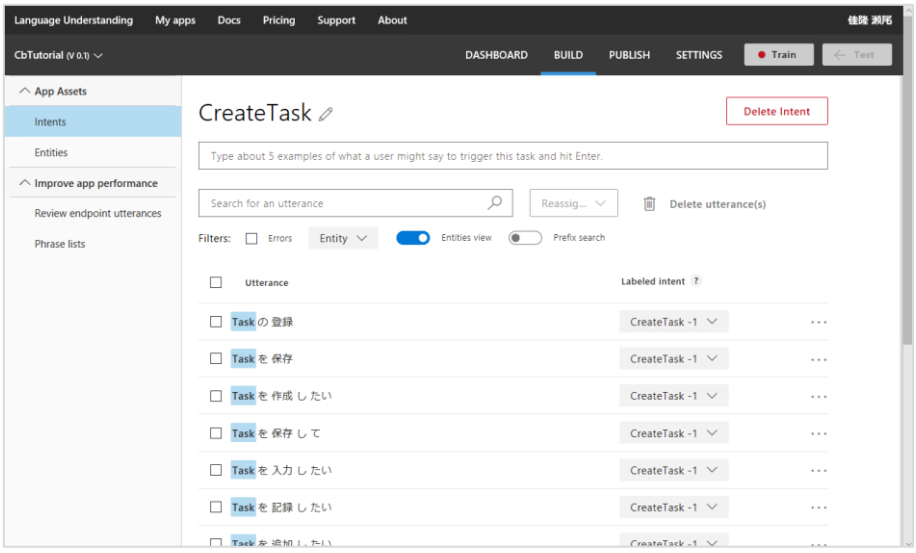
11. 新しい Intent を作ります。Intent は “CreateTask” という名前にします。



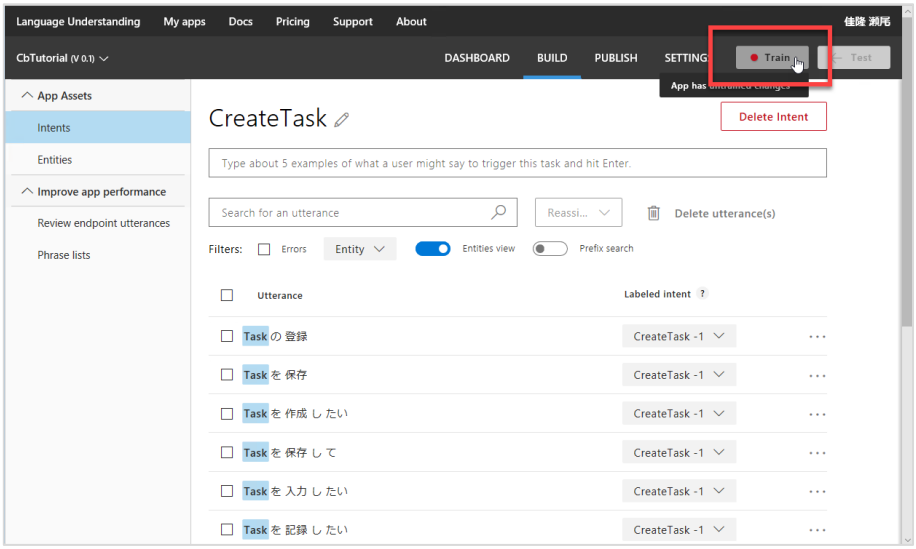
12. 例文を入力して、エンティティを指定します。

例文	Entity の指定
予定の追加	予定→Task
タスクを登録したい	タスク→Task
作業を追加した	作業→Task
タスクを記録したい	タスク→Task
作業を入力したい	作業→Task
タスクを保存して	タスク→Task
作業を作成したい	作業→Task
作業を保存	作業→Task
タスクの登録	タスク→Task

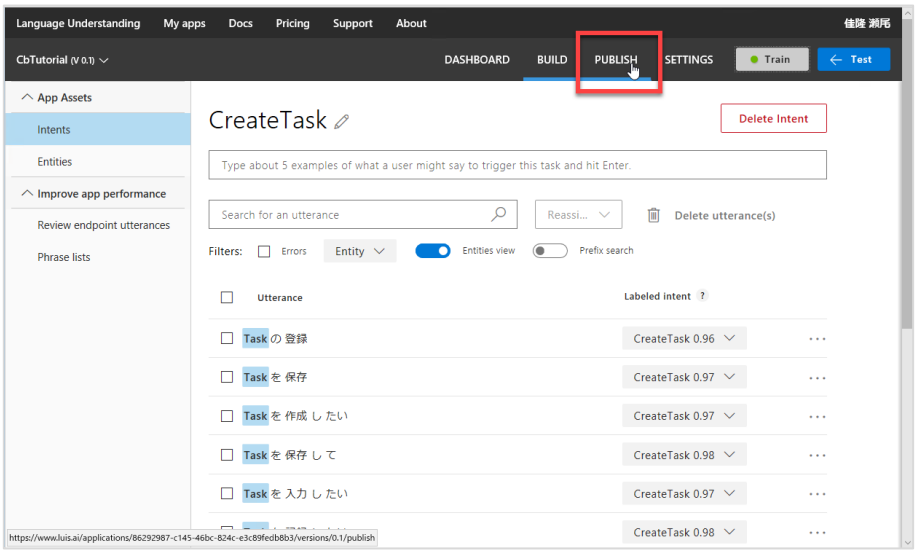
例文の入力とエンティティの指定が完了すると、以下のようになります。



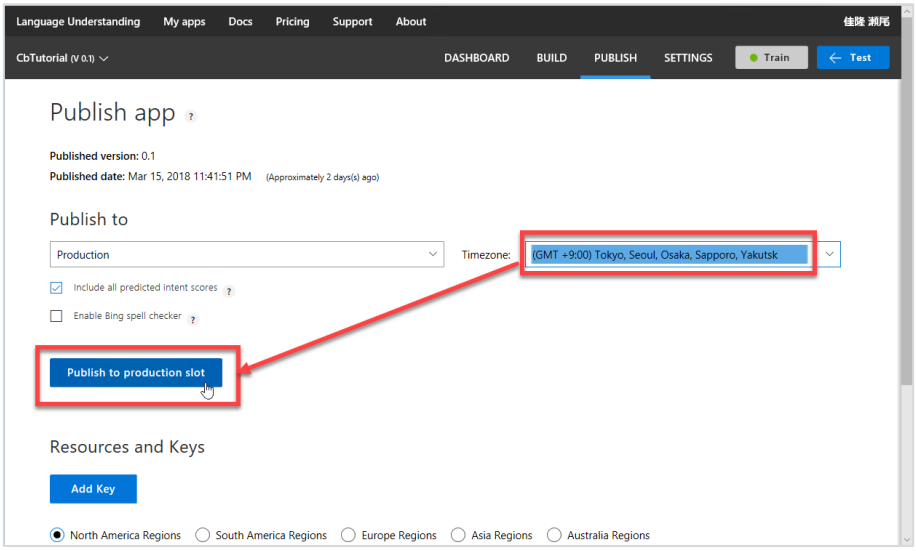
13. [Train] をクリックして、入力した内容を学習します。



14. [PUBLISH] をクリックして、配置を始めます。



15. [Timezone] を “GMT +9:00” にして、[Publish to production slot] をクリックします。



以上で、Language Understanding の拡張が完了しました。エンティティを使用することで、より詳細なユーザーの意図を汲み取ることができるようになります。

28. FormFlow

ここまでのステップで、Bot アプリケーションが自然言語を理解することで、ユーザーに対して柔軟な対応ができるようになることが分かってきたと思います。

しかし自由な会話の中から常に人間の要求を理解するのは難しいことがあります。また技術的には可能でも、必ずしもユーザーの負担を減らす効果がないこともあります。

例えば、電車の切符を買いたい場合、

「明日の 8 時以降の電車で、禁煙車の窓側指定席を 1 枚予約して」

は語順や語尾の揺らぎで何通りもの表現があるため、それらを正確に理解するのは難しそうです。また人間が必要な情報を漏れなく教えてくれるとも限りません。
そこで、それよりは、

人間「切符を予約したい」
Bot 「いつの電車ですか」
人間「明日の 8 時」
Bot 「禁煙車、喫煙車の、どちらの車両がご希望ですか」
・・・

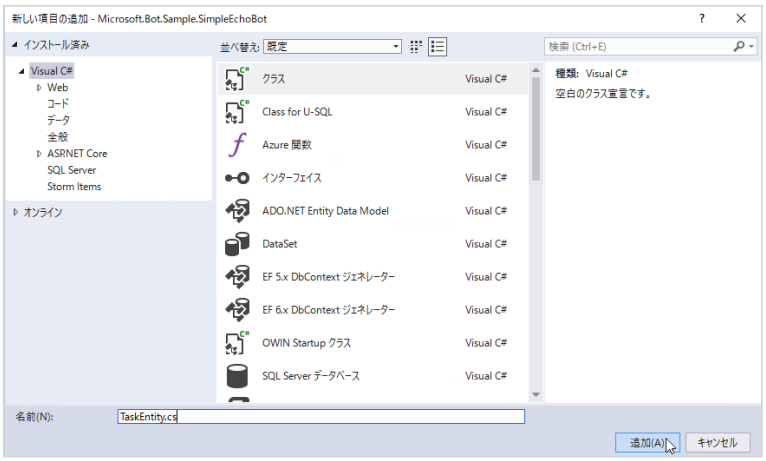
のように Bot が誘導して必要な情報を収集するほうが、人間の負担を減らすことができるケースもあります。

このように、Bot がユーザーに対して、次に教えてほしい情報を順にリクエストするための技術が FormFlow です。

29. FormFlow の実装

FormFlow を使って、ユーザーに必要な入力を流す処理を実装していきます。

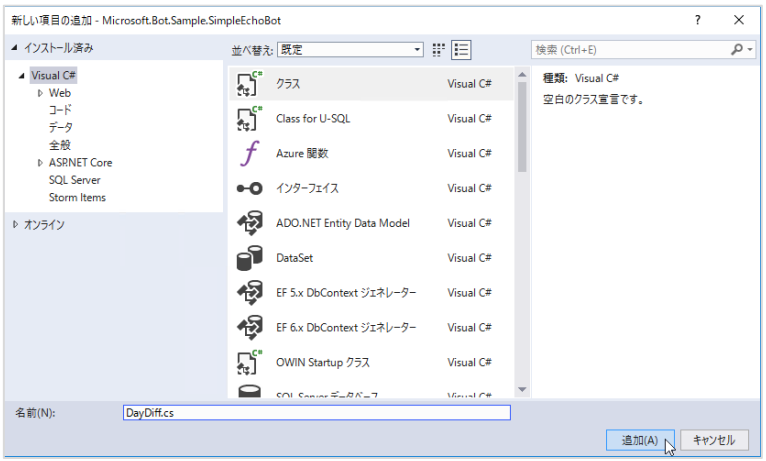
1. Visual Studio 2017 のソリューションエクスプローラーの [Models] で右クリックして、新しいクラスを追加します。ファイル名は “TaskEntity.cs” とします。



2. TaskEntity.cs 全体を以下のように変更します。

```
using System;
using Microsoft.WindowsAzure.Storage.Table;

namespace SimpleEchoBot.Models
{
    public class TaskEntity : TableEntity
    {
        public TaskEntity(string userId, string subject, DateTimeOffset taskDay, string details)
        {
            PartitionKey = userId;
            RowKey = Guid.NewGuid().ToString();
            Subject = subject;
            TaskDay = taskDay;
            Details = details;
        }

        public TaskEntity() { }

        public string Subject { get; set; }

        public DateTimeOffset TaskDay { get; set; }

        public string Details { get; set; }
    }
}
```

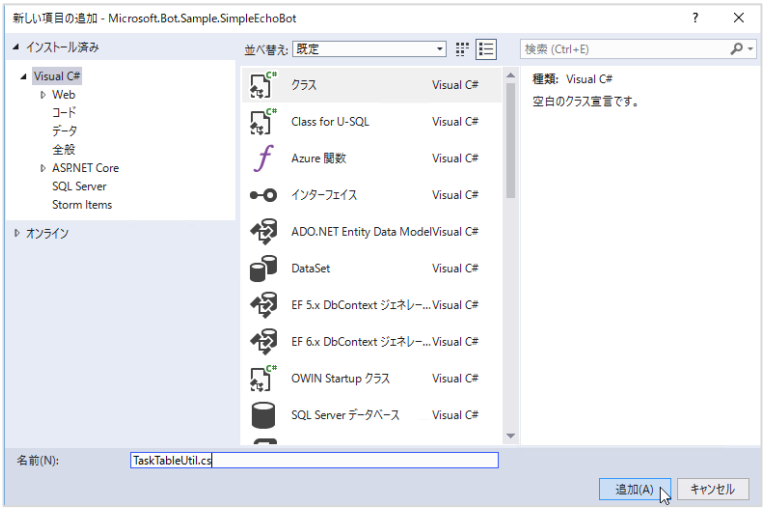
3. Visual Studio 2017 のソリューションエクスプローラーの [Utils] で右クリックして、新しいクラスを追加します。ファイル名は “DayDiff.cs” とします。



4. DayDiff.cs 全体を以下のように変更します。

```
using System;

namespace SimpleEchoBot.Utils
{
    public static class DayDiff
    {
        public static int? GetDiff(string diffString)
        {
            switch (diffString)
            {
                case "Day::Today":
                    return 0;
                case "Day::Tomorrow":
                    return 1;
                case "Day::DayAfterTomorrow":
                    return 2;
                case "Day::Yesterday":
                    return -1;
                case "Day::FromNow":
                    return null;
                default:
                    throw new ApplicationException("No Match");
            }
        }
    }
}
```

5. Visual Studio 2017 のソリューションエクスプローラーの [Utils] で右クリックして、新しいクラスを追加します。ファイル名は “TaskTableUtil.cs” とします。



6. TaskTableUtil.cs 全体を以下のように変更します。

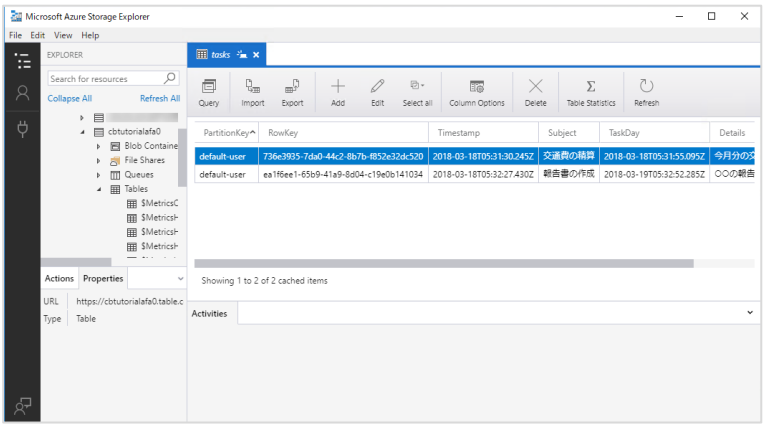
```
using System;
using System.Collections.Generic;
using System.Configuration;
using Microsoft.WindowsAzure.Storage;
using Microsoft.WindowsAzure.Storage.Table;
using SimpleEchoBot.Models;

namespace SimpleEchoBot.Utils
{
    public static class TaskTableUtil
    {
        public static CloudTable GetTasksTable()
        {
            var storageAccount = CloudStorageAccount
                .Parse(ConfigurationManager.AppSettings["AzureWebJobsStorage"]);
            var tableClient = storageAccount.CreateCloudTableClient();
            var table = tableClient.GetTableReference("tasks");
            table.CreateIfNotExists();

            return table;
        }

        // 次に続く
```

```
        // 前から続く

        public static IEnumerable<TaskEntity> GetTasks(string userId, int? dayDiff)
        {
            TableQuery<TaskEntity> tasks;
            if (dayDiff.HasValue)
            {
                var queryDate = TableQuery.CombineFilters(
                    TableQuery.GenerateFilterConditionForDate("TaskDay",
                        QueryComparisons.GreaterThanOrEqual,
DateTime.Now.AddDays(dayDiff.Value).Date),
                    TableOperators.And,
                    TableQuery.GenerateFilterConditionForDate("TaskDay",
                        QueryComparisons.LessThan,
DateTime.Now.Date.AddDays(dayDiff.Value + 1)));

                tasks = new TableQuery<TaskEntity>().Where(
                    TableQuery.CombineFilters(
                        TableQuery.GenerateFilterCondition("PartitionKey",
QueryComparisons.Equal, userId),
                        TableOperators.And,
                        queryDate));
            }
            else
            {
                tasks = new TableQuery<TaskEntity>().Where(
                    TableQuery.CombineFilters(
                        TableQuery.GenerateFilterCondition("PartitionKey",
                            QueryComparisons.Equal, userId),
                        TableOperators.And,
                        TableQuery.GenerateFilterConditionForDate("TaskDay",
                            QueryComparisons.GreaterThanOrEqual, DateTime.Now.Date)));
            }

            return GetTasksTable().ExecuteQuery(tasks);
        }
    }
}
```

ワンポイント

このあと実装する機能により、「タスク」を保存することができるようになります。

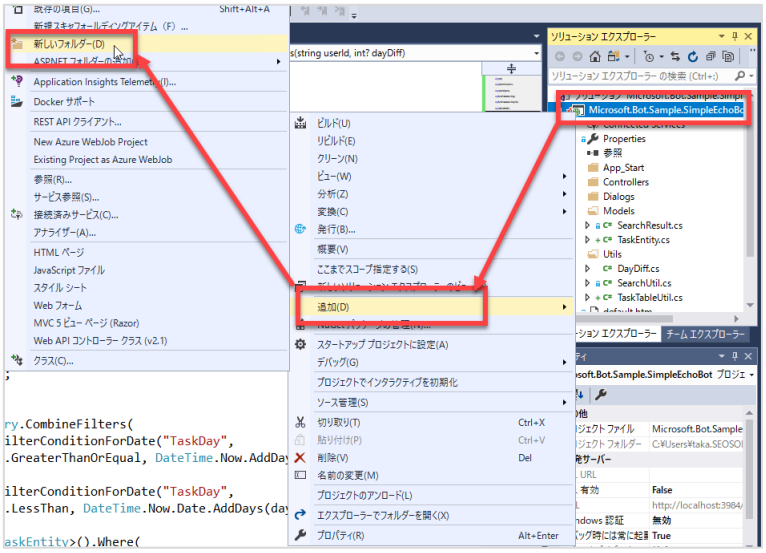
この自習書では、「タスク」のデータストアとして、Azure Table ストレージを使用します。テーブル名は “tasks” なので、デバッグ時にはこれを参照できると便利です。
このような用途には Azure Storage Explorer が便利です。



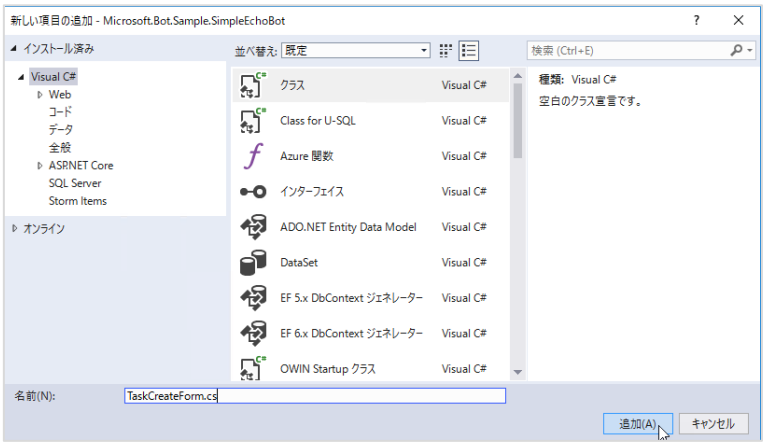
Azure Storage Explorer は、Table ストレージだけではなく、Azure の他のストレージ（Blob、キュー、Cosmos DB、Data Lake ストレージ）の表示と編集も可能です。

Azure Storage Explorer のインストーラーは、
https://azure.microsoft.com/ja-jp/features/storage-explorer/
でダウンロード可能です。

7. ソリューションエクスプローラーのプロジェクトで右クリックして、[追加] – [新しいフォルダー] を選択します。追加したフォルダー名は “Forms” に変更します。



8. ソリューションエクスプローラーの“Forms”で右クリックして、新しいクラスを追加します。ファイル名は“TaskCreateForm.cs”とします。



9. TaskCreateForm.cs 全体を以下のように変更します。

```
using System;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.Bot.Builder.Dialogs;
using Microsoft.Bot.Builder.FormFlow;
using Microsoft.Bot.Connector;
using Microsoft.WindowsAzure.Storage.Table;
using SimpleEchoBot.Models;
using SimpleEchoBot.Utils;

namespace SimpleEchoBot.Forms
{
    public enum TaskDay
    {
        今日 = 1,
        明日,
        明後日
    };

    [Serializable]
    public class TaskCreateForm
    {
        [Describe("タスク名")]
        public string Subject;
        [Describe("登録する日")]
        public TaskDay TaskDay;
        [Describe("タスクの内容")]
        public string Details;

        // 次に続く
```

```
// 前から続く

public static IForm<TaskCreateForm> BuildForm()
{
    var formBuilder = new FormBuilder<TaskCreateForm>();
    formBuilder.Configuration.NoPreference = new[] { "None", "変更なし" };

    formBuilder.Message("タスクを作成します")
        .Field(nameof(Subject))
        .Field(nameof(TaskDay))
        .Field(nameof(Details))
        .Confirm(ConfirmTaskAsync)
        .OnCompletion(CompleteCreateAsync);

    formBuilder.Configuration.Templates
        .Single(t => t.Usage == TemplateUsage.NoPreference)
        .Patterns = new[] { "変更なし" };

    return formBuilder.Build();
}

private static async Task<PromptAttribute> ConfirmTaskAsync(TaskCreateForm state)
{
    return new PromptAttribute($"以下の内容で登録しますか¥n¥n" +
                               $"タスク名: {state.Subject}¥n¥n" +
                               $"登録する日: {state.TaskDay}¥n¥n" +
                               $"タスクの内容: {state.Details}¥n¥n" + "{||}");
}

// 次に続く
```

```
        // 前から続く

        private static async Task CompleteCreateAsync(IDialogContext context,
            TaskCreateForm state)
        {
            var userId = ((IMessageActivity)context.Activity).From.Id;
            var day = DateTime.Now.AddDays((int)state.TaskDay - 1);

            var entity = new TaskEntity(userId, state.Subject, day, state.Details);
            var operation = TableOperation.Insert(entity);
            var table = TaskTableUtil.GetTasksTable();
            try
            {
                table.Execute(operation);
                await context.PostAsync($"以下の内容で登録しました¥n¥n" +
                                        $"タスク名: {state.Subject}¥n¥n" +
                                        $"登録する日: {state.TaskDay}¥n¥n" +
                                        $"タスクの内容: {state.Details}");
            }
            catch (Exception)
            {
                await context.PostAsync("タスクの登録に失敗しました");
            }

        }
    }
}
```

FormFlow について少し説明を加えましょう。

```
formBuilder.Message("タスクを作成します")
        .Field(nameof(Subject))
        .Field(nameof(TaskDay))
        .Field(nameof(Details))
        .Confirm(ConfirmTaskAsync)
        .OnCompletion(CompleteCreateAsync);
```

の部分が、FormFlow の処理フローを決めている個所です。
上から順に、

- 「タスクを作成します」というメッセージをユーザーに返す
- Subject (件名) をユーザーに入力させる
- TaskDay (日付) をユーザーに入力させる
- Details (内容) をユーザーに入力させる
- 入力に間違いがないかユーザーに確認して、必要に応じて個別に再入力させる
- 入力のフローが完了したら後処理（ここでは CompleteCreateAsync）を呼び出す

という処理を行います。
追加の記述をすることで挙動をカスタマイズすることもできますが、これだけの記述で基本的な流れを実装できます。

10. MyLuisDialog.cs の先頭付近を以下のように変更します。
以下の 4 行を追加しています。

- using System.Linq;
- using Microsoft.Bot.Builder.FormFlow;
- using SimpleEchoBot.Forms;
- using SimpleEchoBot.Utils;

```
using System;
using System.Configuration;
using System.Linq;
using System.Threading;
using System.Threading.Tasks;
using Microsoft.Bot.Builder.Dialogs;
using Microsoft.Bot.Builder.FormFlow;
using Microsoft.Bot.Builder.Luis;
using Microsoft.Bot.Builder.Luis.Models;
using Microsoft.Bot.Connector;
using SimpleEchoBot.Forms;
using SimpleEchoBot.Utils;

namespace SimpleEchoBot.Dialogs
{
    // 以下、省略
```

11. MyLuisDialog クラスに、以下の通り GetTasksIntent メソッドを追加します。

```
[LuisIntent("GetTasks")]
public async Task GetTasksIntent(IDialogContext context, IAwaitable<IMessageActivity> activity,
LuisResult result)
{
    var dayDisplayString = GetDayEntity(result)?.Entity;
    if (string.IsNullOrEmpty(dayDisplayString))
    {
        await PostUnknownQuestionMessageAsync(context);
        return;
    }

    var diff = await GetDayDiffAsync(context, GetDayEntity(result)?.Type);

    var message = await activity;
    var tasks = TaskTableUtil.GetTasks(message.From.Id, diff).OrderBy(t => t.TaskDay).ToList();
    if (tasks.Count > 0)
    {
        tasks.ForEach(async t =>
        {
            await context.PostAsync($"{t.Subject}¥n¥n" +
                $"{t.TaskDay.ToString("yyyy 年 M 月 d 日")}¥n¥n" +
                $"{t.Details}");
        });
    }
    else
    {
        await context.PostAsync($"{dayDisplayString}のタスクはありません");
    }

    context.Done(true);
}
```

12. MyLuisDialog にいくつかメソッドを追加します。これらは GetTasksIntent メソッドから呼び出されるものです。

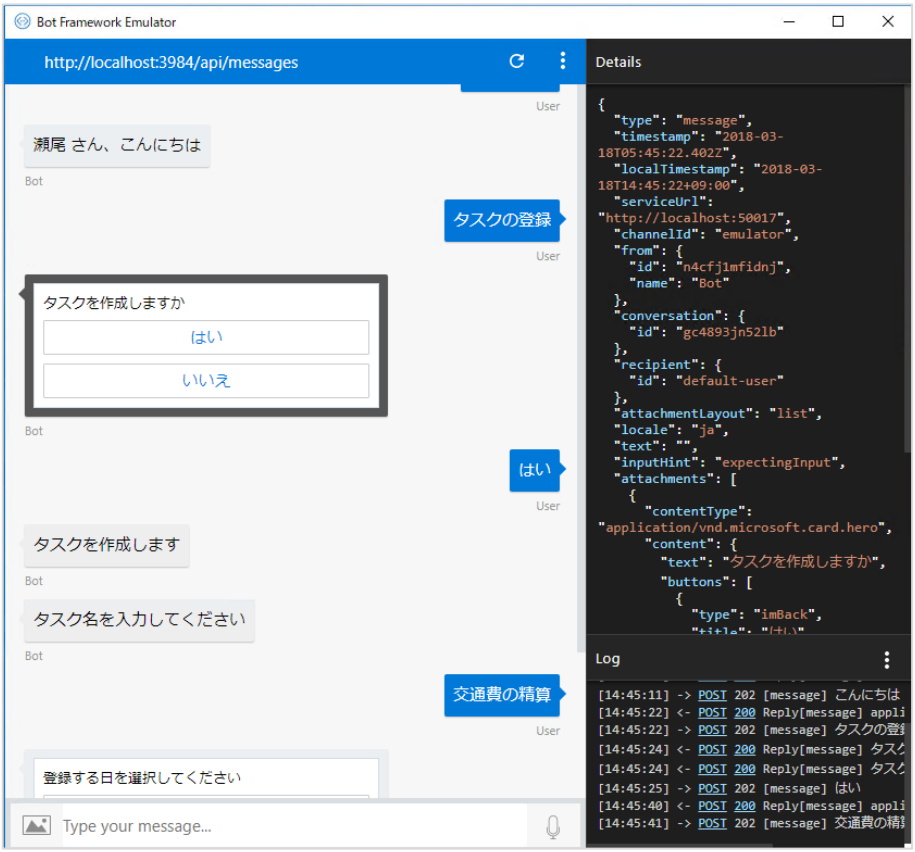
```
private static EntityRecommendation GetDayEntity(LuisResult result) =>
    result.Entities.SingleOrDefault(e => e.Type.StartsWith("Day::"));

private static async Task PostUnknownQuestionMessageAsync(IDialogContext context)
{
    await context.PostAsync("わかりません。質問しなおしてください");
    context.Done(true);
}

private static async Task<int?> GetDayDiffAsync(IDialogContext context, string dayString)
{
    try
    {
        return DayDiff.GetDiff(dayString);
    }
    catch (Exception)
    {
        await context.PostAsync("質問しなおしてください");
        context.Done(true);
        return null;
    }
}
```

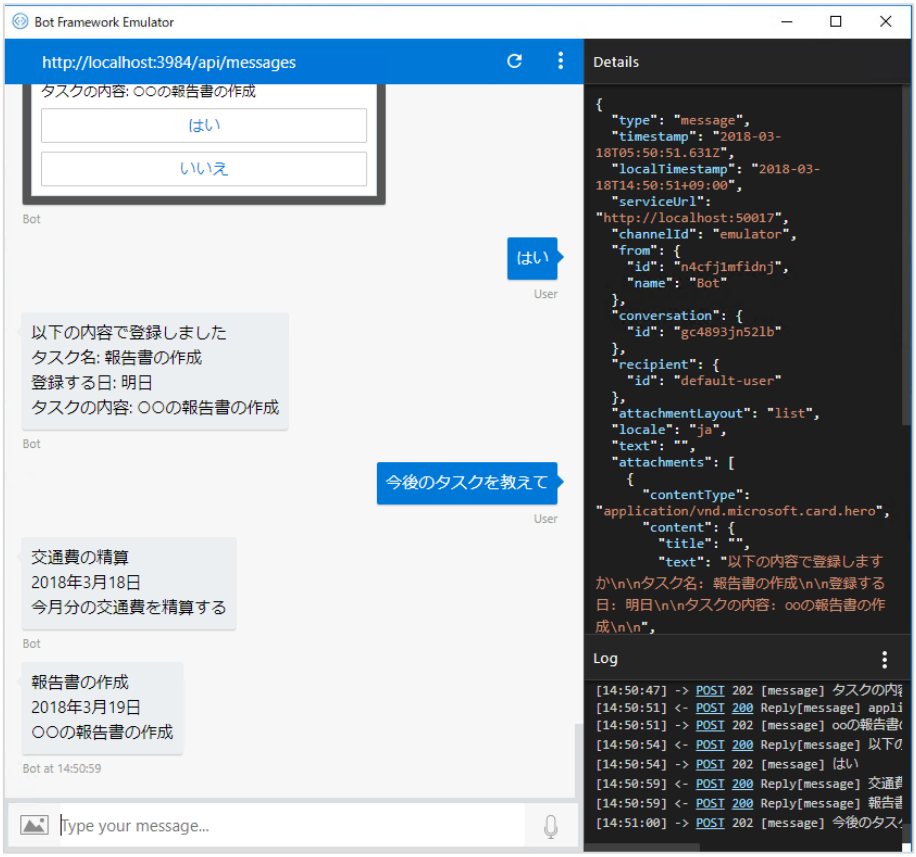
13. MyLuisDialog クラスに、以下の通り GetTasksIntent メソッド、ConfirmCreateTaskAsync メソッド、および AfterCreateTask メソッドを追加します。

```
[LuisIntent("CreateTask")]
public async Task CreateTaskIntent(IDialogContext context, IAwaitable<IMessageActivity> activity,
LuisResult result)
{
    PromptDialog.Confirm(context, ConfirmCreateTaskAsync, "タスクを作成しますか");
}

private async Task ConfirmCreateTaskAsync(IDialogContext context, IAwaitable<bool> result)
{
    if (await result)
    {
        await context.Forward(
            Chain.From(() => FormDialog.FromForm(TaskCreateForm.BuildForm)),
            AfterCreateTask, context.Activity, CancellationToken.None);
    }
    else
    {
        await context.PostAsync("タスクの作成を中止します");
        context.Done(true);
    }
}

private static async Task (IDialogContext context, IAwaitable<object> result)
{
    context.Done(true);
}
```

14. ビルドに成功したらデバッグ実行します。
これまでの機能の他に、以下の機能が追加されました。

- 「今日のタスクは何」、「今後の作業を教えて」などとユーザーが入力すると、登録済みのタスクを Bot アプリケーションが返します。
- 「タスクの登録」、「作業の保存」などとユーザーが入力すると、タスクを登録することができます。この時、Bot アプリケーションがフロー形式でユーザーに入力を促し、ストレスなくタスクの情報を入力できます。





ワンポイント

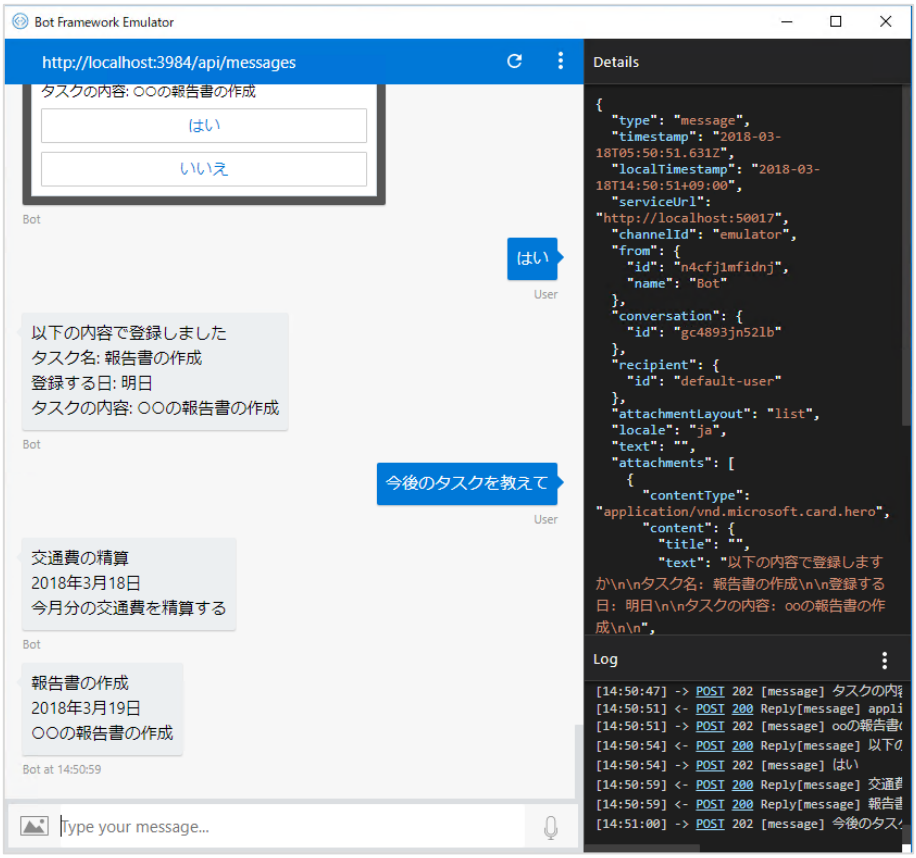
本来ならば、タスクの登録、参照などは Office 365 に含まれる Exchange の予定表と連動するのがよいでしょう。

今回は Office 365 なしでも実習を完了できるように、Azure Table ストレージを使用しています。

また、Bot アプリケーション開発の要点を理解するには、この自習書内の範囲でも十分であると判断し、自由な日程のタスク登録やタスク完了フラグなどの機能を実装していません。
Bot アプリケーションの開発が分かってきたら、ぜひ各自でチャレンジしてみてください。

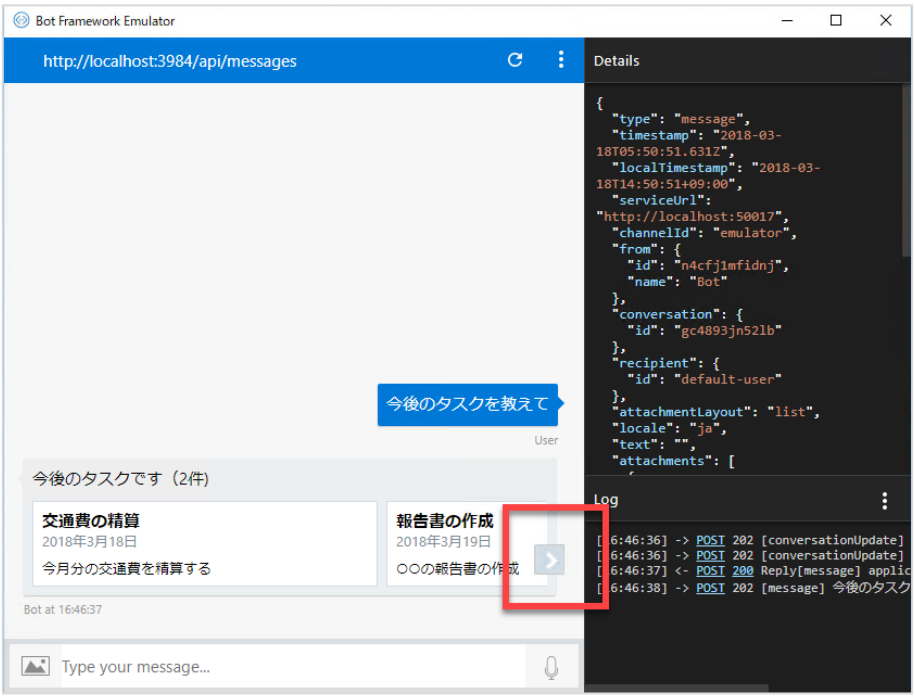
30. Thumbnail Card の実装

ここまでの実装で、Bot アプリケーションの機能的な実装は完了しました。
ただ、UX (ユーザーエクスペリエンス) で一つ課題が残っています。



ユーザーが「今後のタスクを教えて」と入力した場合、該当するタスクが多数あると、会話の全体がスクロールしてしまい、大事な情報を見落とす、または上に戻って確認しなおすなどの問題が発生します。結果としてユーザーに負担を強いるかもしれません。

そこで、ここまでは「吹き出し」の形で表示していたものを「カード」の形にして、かつ一連のカードを横スクロールする表示にします。



1. MyLuisDialog.cs を開き、先頭部分を以下のように変更します。

“using System.Collections.Generic;” および “using SimpleEchoBot.Models;” の 2 行を追加しています。

```
using System.Threading.Tasks;
using Microsoft.Bot.Builder.Dialogs;
using Microsoft.Bot.Builder.FormFlow;
using Microsoft.Bot.Builder.Luis;
using Microsoft.Bot.Builder.Luis.Models;
using Microsoft.Bot.Connector;
using SimpleEchoBot.Forms;
using SimpleEchoBot.Models;
using SimpleEchoBot.Utils;

namespace SimpleEchoBot.Dialogs
{
    // 以下、省略
```

2. MyLuisDialog の GetTasksIntent メソッド全体を以下のように変更します。

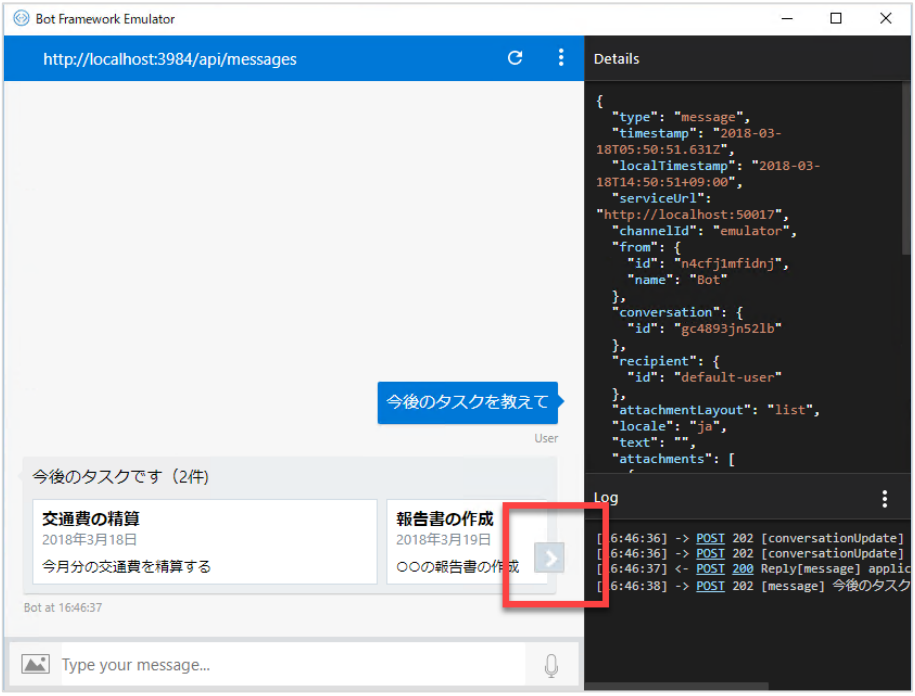
変更箇所は、"if (tasks.Count > 0)" のブロックの中です。
変更前は、各タスクを順に PostAsync していましたが、変更後はカードをまとめて作成してユーザーに返すようにします。

```
[LuisIntent("GetTasks")]
public async Task GetTasksIntent(IDialogContext context, IAwaitable<IMessageActivity> activity,
LuisResult result)
{
    var dayDisplayString = GetDayEntity(result)?.Entity;
    if (string.IsNullOrEmpty(dayDisplayString))
    {
        await PostUnknownQuestionMessageAsync(context);
        return;
    }

    var diff = await GetDayDiffAsync(context, GetDayEntity(result)?.Type);

    var message = await activity;
    var tasks = TaskTableUtil.GetTasks(message.From.Id, diff).OrderBy(t => t.TaskDay).ToList();
    if (tasks.Count > 0)
    {
        var reply = MakeCardsReply(context, tasks, dayDisplayString);
        await context.PostAsync(reply);
    }
    else
    {
        await context.PostAsync($"{dayDisplayString}のタスクはありません");
    }

    context.Done(true);
}
```

3. MyLuisDialog に MakeCardsReply メソッドを追加します。

```
private static IMessageActivity MakeCardsReply(IDialogContext context, List<TaskEntity> tasks, string dayDisplayString)
{
    var reply = context.MakeMessage();
    reply.Text = $"{dayDisplayString}のタスクです（{tasks.Count}件)";
    reply.AttachmentLayout = AttachmentLayoutTypes.Carousel;

    tasks.ForEach(t =>
    {
        reply.Attachments.Add(
            new ThumbnailCard
            {
                Title = t.Subject,
                Subtitle = t.TaskDay.ToString("yyyy 年 M 月 d 日"),
                Text = t.Details
            }.ToAttachment());
    });

    return reply;
}
```

4. ビルドに成功したらデバッグ実行します。
「今後のタスクを教えて」などと入力すると、該当するタスクが横方向にスクロールする“Carousel”で表示されます。



以上で、この自習書での Bot アプリケーションの実装（コーディング）はすべて完了しました。

続いて、Bot アプリケーションをクラウドに配置して、Skype などのチャットクライアントで利用する手順を紹介します。

STEP 8. チャットクライアントに対応する

このステップでは、開発した Bot アプリケーションを Microsoft Azure にデプロイして、作業 PC 以外の PC、スマートフォンなどから Bot アプリケーションを利用できるようにします。

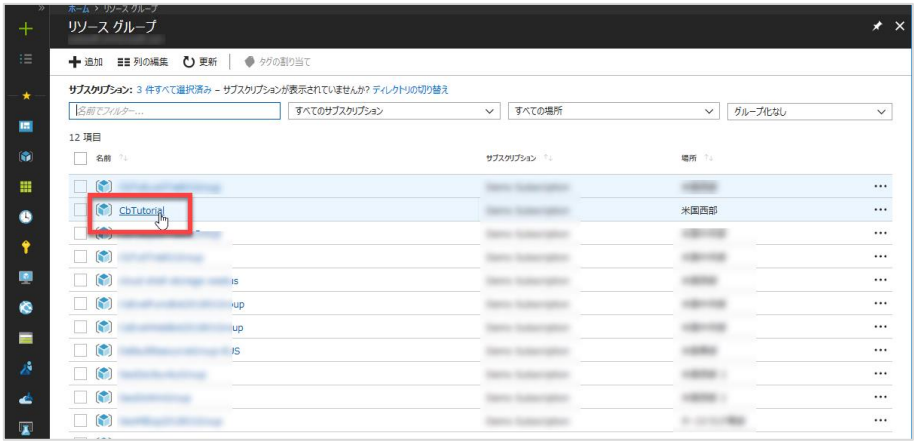
- ✓ Bot アプリケーションのデプロイ
- ✓ Bot Framework Emulator から接続確認
- ✓ Bot Channels
- ✓ Web Chat
- ✓ Skype
- ✓ Microsoft Teams

31. Bot アプリケーションのデプロイ

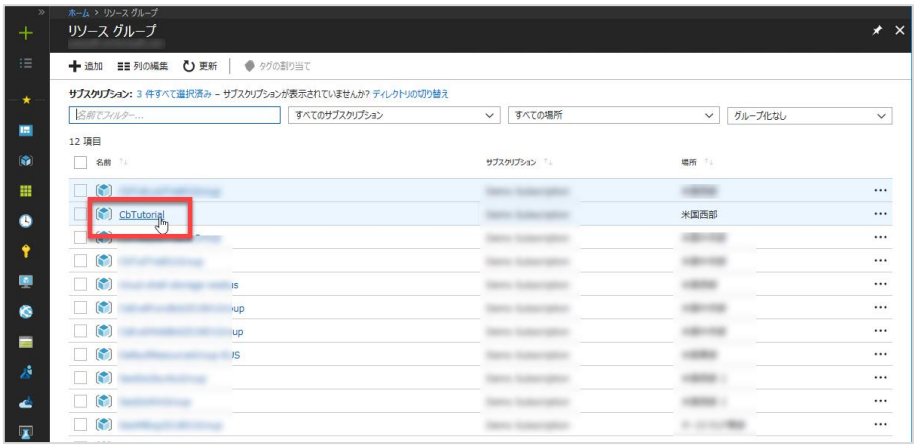
Bot アプリケーションをクラウドに発行することで、Skype などのチャットアプリケーションから利用できるようになります。

1. Microsoft Azure 管理ポータル (https://portal.azure.com/) を開いて、サインインします。

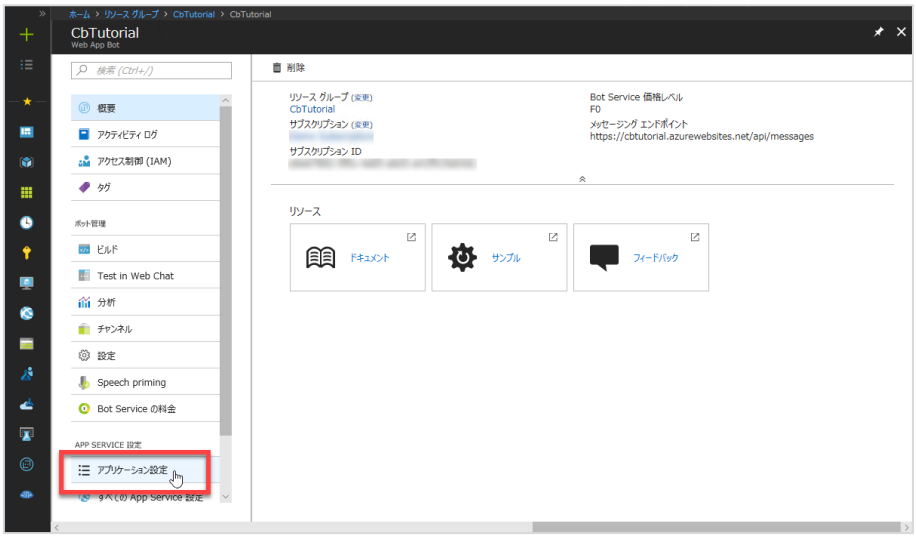
2. この自習書で使っているリソースグループを開きます。



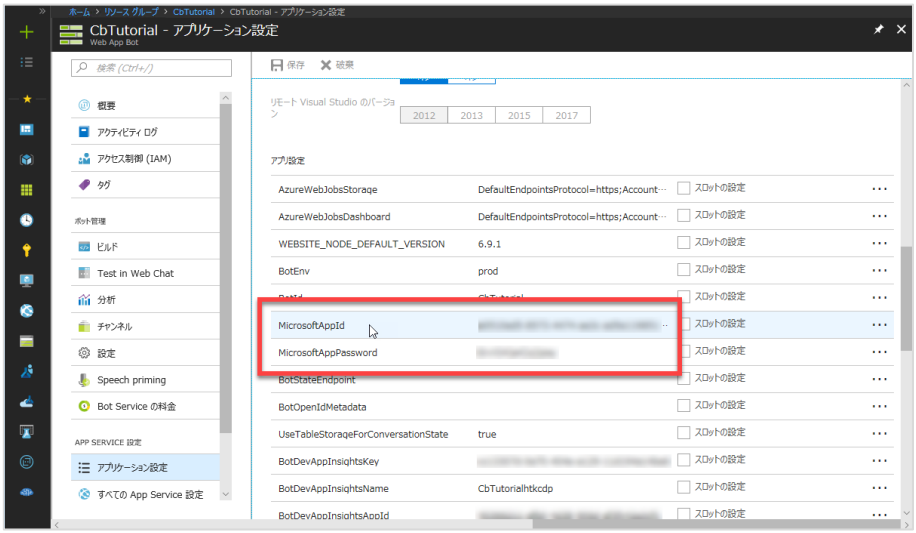
3. [Web App Bot] の名前を選択します。



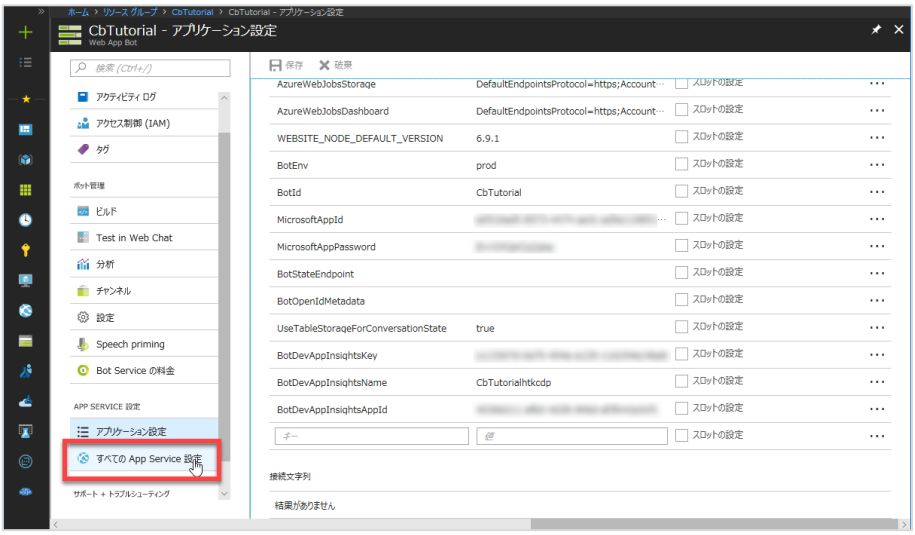
4. [アプリケーション設定] をクリックします。



5. ブレードの中ほどにある [アプリ設定] で、“MicrosoftAppId” および “MicrosoftAppPassword” の 2 項目の値をメモ帳などにコピーします。

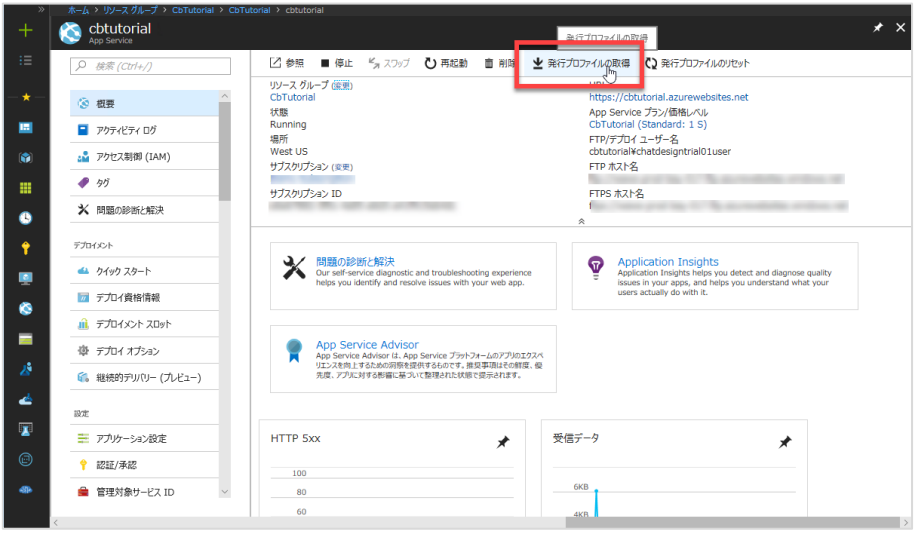


6. [すべての App Service 設定] をクリックします。



7. [発行プロファイルの取得] をクリックします。

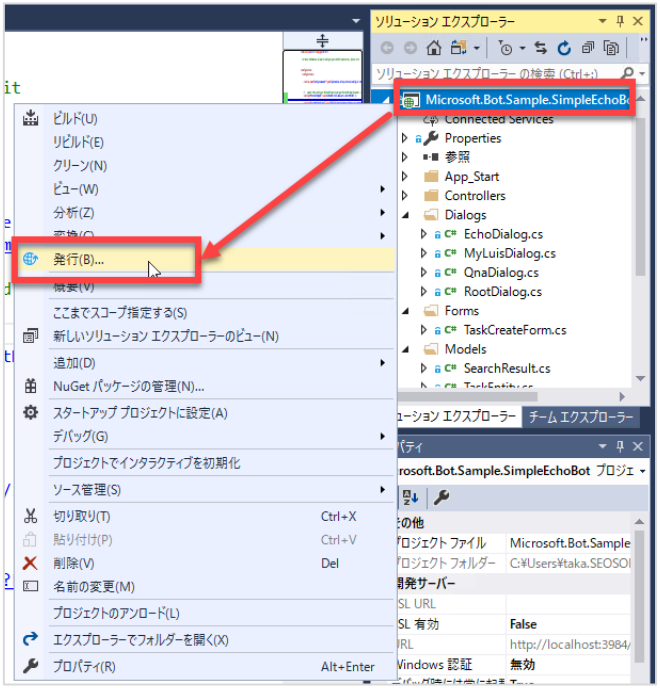
発行プロファイルは、作業 PC の任意のフォルダーに保存します。



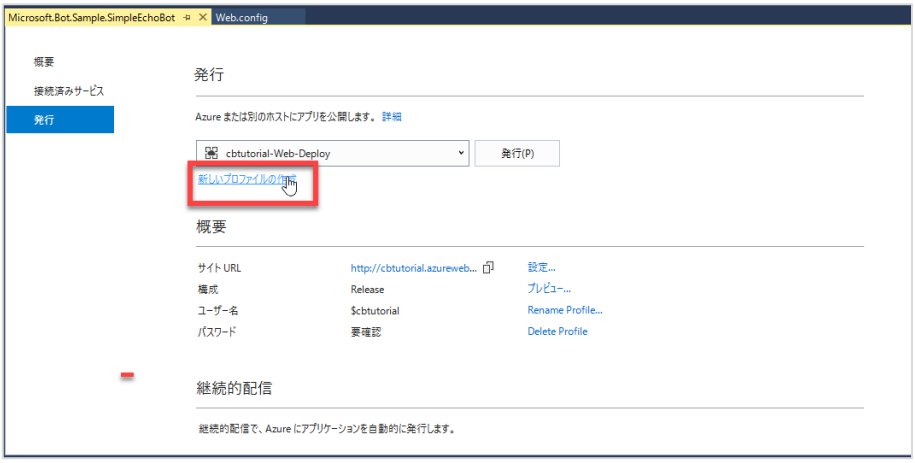
8. Visual Studio 2017 で Web.config を開き、“MicrosoftAppId” および “MicrosoftAppPassword” の値を Azure 管理ポータルで取得した値で書き換えます。

```
<appSettings>
    <!-- update these with your Microsoft App Id and your Microsoft App Password-->
    <add key="MicrosoftAppId" value="<管理ポータルで取得した値>" />
    <add key="MicrosoftAppPassword" value="<管理ポータルで取得した値>" />
    <add key="AzureWebJobsStorage" value="<接続文字列>" />

    <add key="LuisAPIKey" value="<設定済み>" />
    <add key="LuisAppId" value="<設定済み>" />
    <add key="LuisAPIHostName" value="<設定済み>" />

    <add key="QnASubscriptionKey" value="<設定済み>" />
    <add key="QnAKnowledgebaseId" value="<設定済み>" />
    <add key="BingSearchAccessKey" value="<設定済み>" />
</appSettings>
```

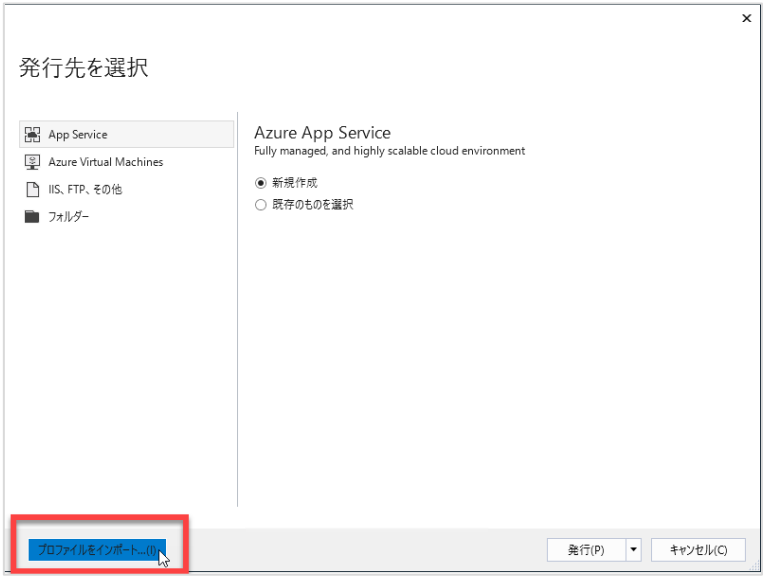
9. ソリューションエクスプローラーのプロジェクトで右クリックして、[発行] を選択します。



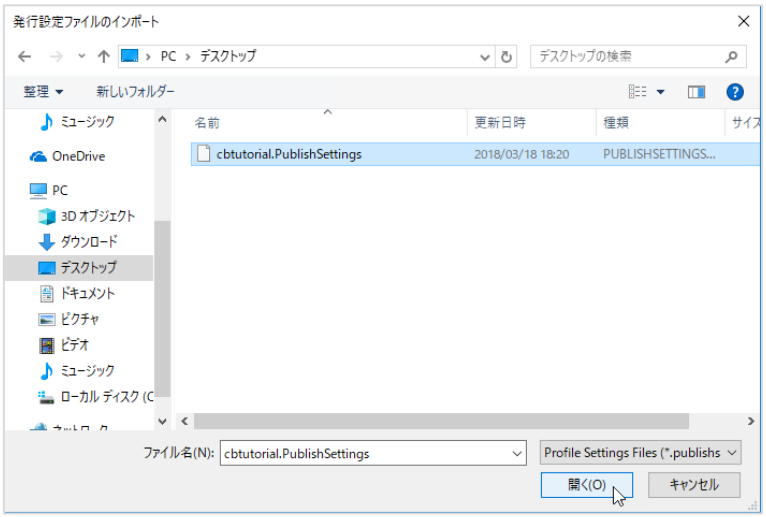
10. 発行ページで [新しいプロファイルの作成] をクリックします。



11. [プロファイルをインポート] をクリックします。

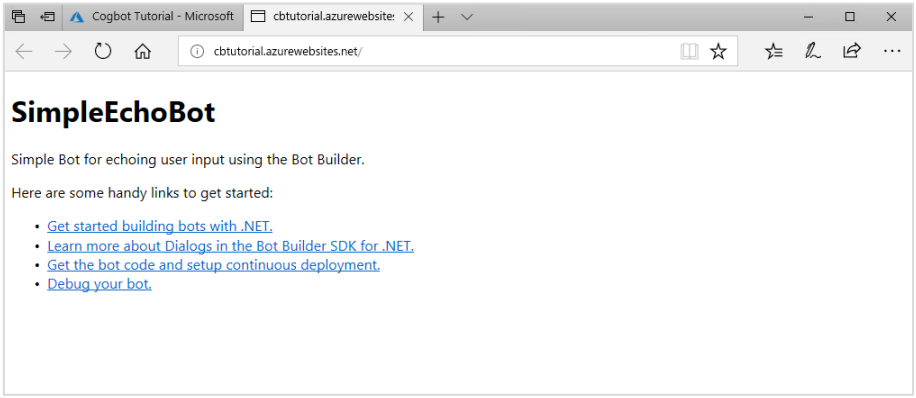


12. Azure 管理ポータルからダウンロードした発行プロファイルを選択します。



13. 自動的に発行が始まります。

14. Web ブラウザーが開き、発行済みの Web サイトが表示されれば発行は成功です。



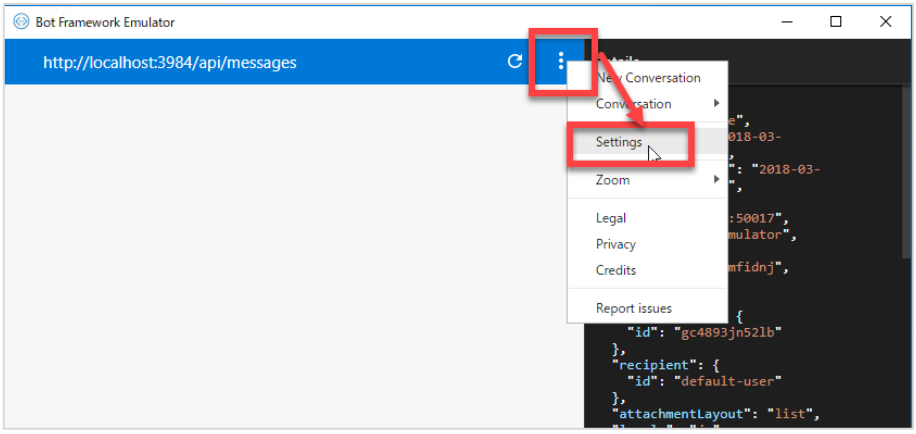
ワンポイント

Azure 管理ポータルから発行プロファイルをダウンロードし、プロファイルのインポートで発行を行う際、発行に失敗することがあります。
この場合は、プロファイルのインポートからもう一度やり直したり、Visual Studio 2017 を再起動したりすることを試してみてください。

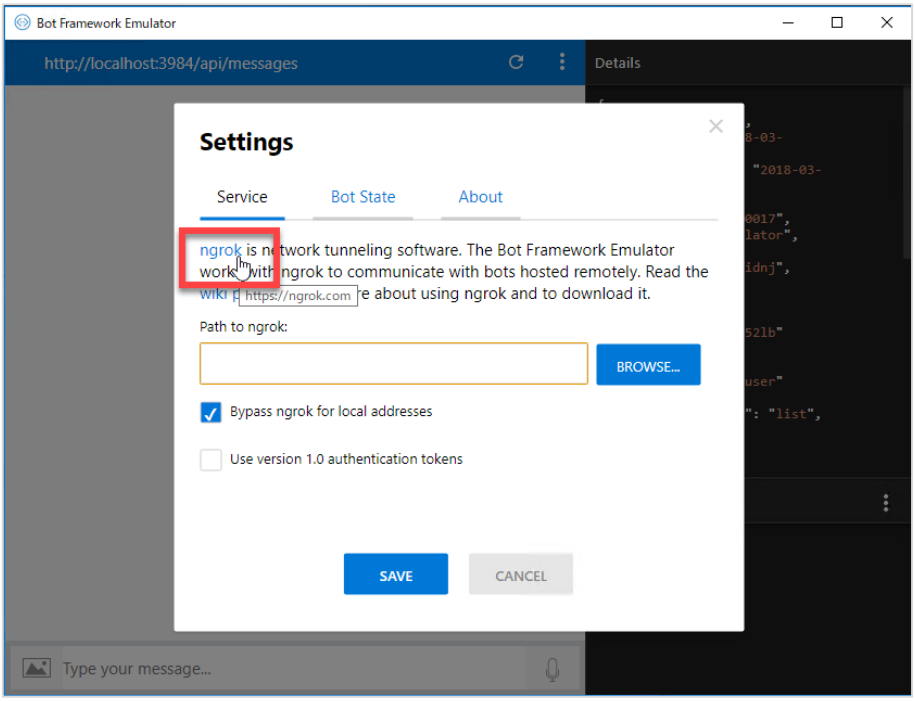
32. Bot Framewor Emulator からの接続確認

Skype などのチャットクライアントから接続する前に、Bot Framework Emulator から接続確認をしてみます。
ローカルの Web サーバーでデバッグ実行している時は、ポートアドレスの間違いに注意する程度で接続できましたが、クラウドにデプロイした Bot アプリケーションと通信するには、追加の手順が必要です。

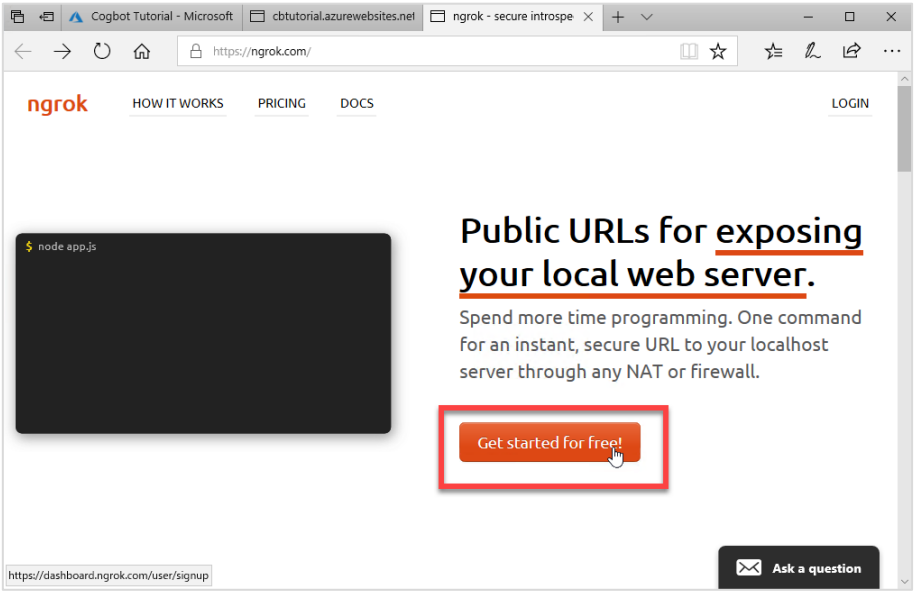
1. Bot Framework Emulator のメニュー (縦の 3 点リーダー) を開き、[Settings] を選択します。



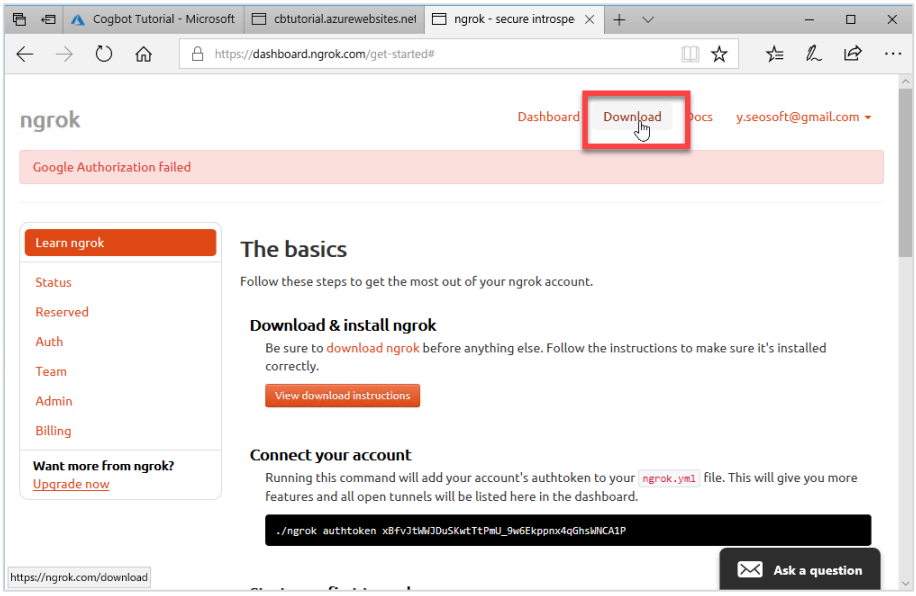
2. Web ブラウザーで ngrok (http://ngrok.com/) のサイトを開きます。
ngrok は、Azure に発行した Bot アプリケーションと Bot Framework Emulator が通信する際に利用するサービスです。



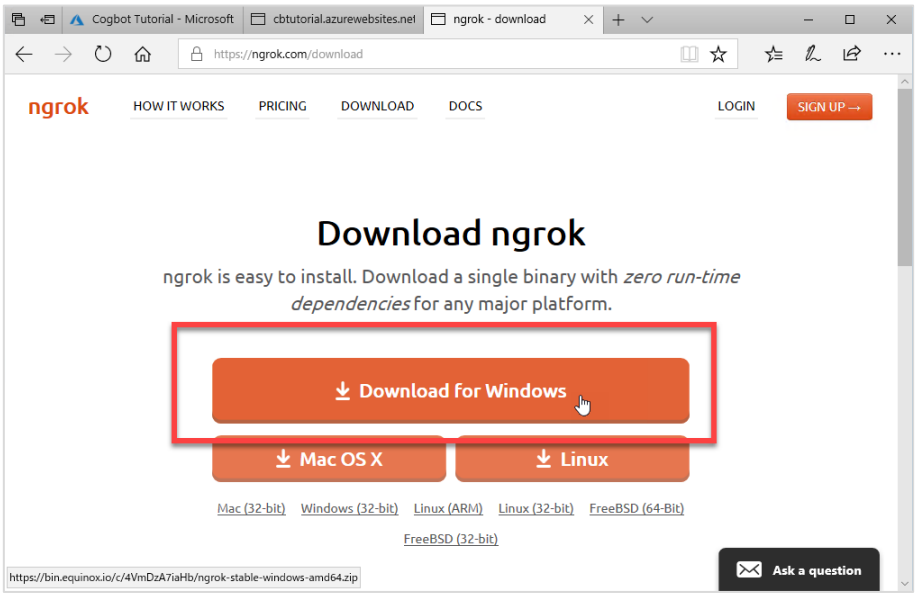
3. ngrok のサイトで、[Get started for free] をクリックします。



4. アカウントを作成します。作成後に [Download] をクリックします。

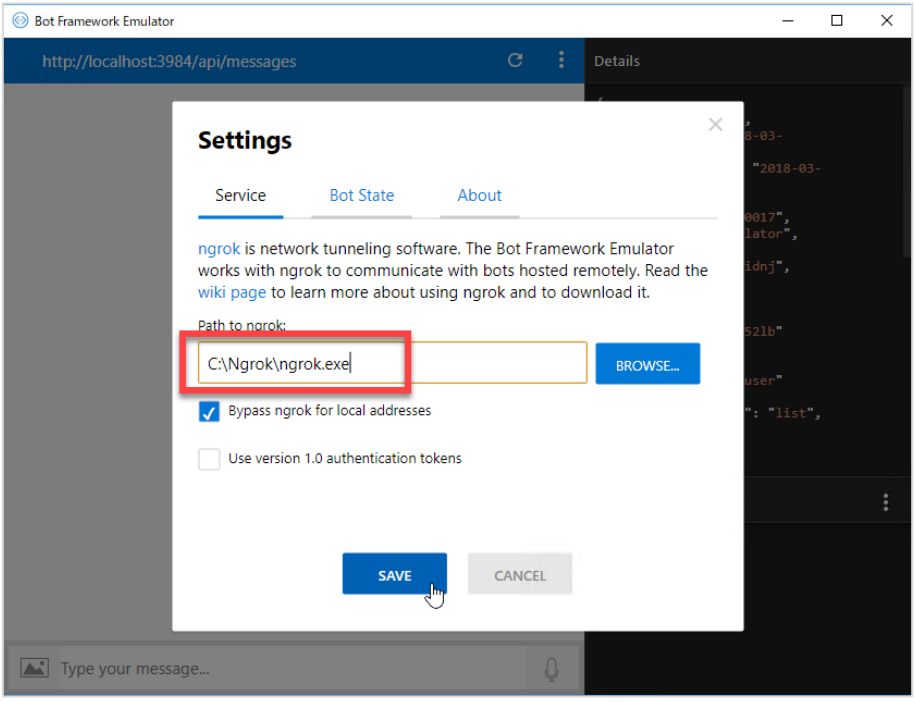


5. [Download for Windows] をクリックして、Windows 版 ngrok をダウンロードします。

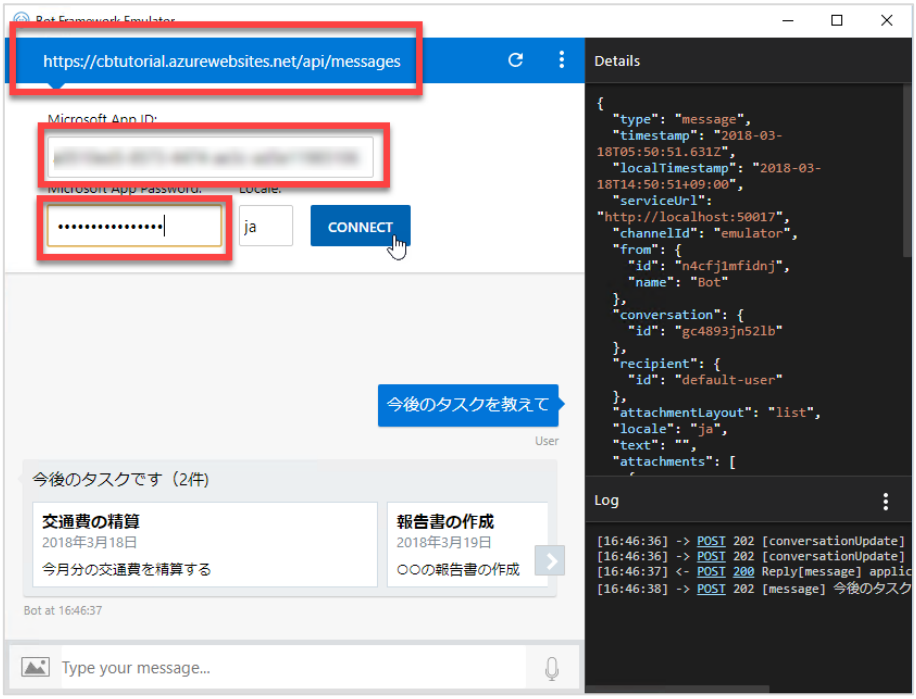


6. ダウンロードした zip ファイルを、作業 PC の任意のフォルダーに展開します。
Zip ファイルには “ngrok.exe” のみが圧縮されています。

7. Bot Framework Emulator の Settings 画面で、ローカルに展開した ngrok.exe のパスを指定します。

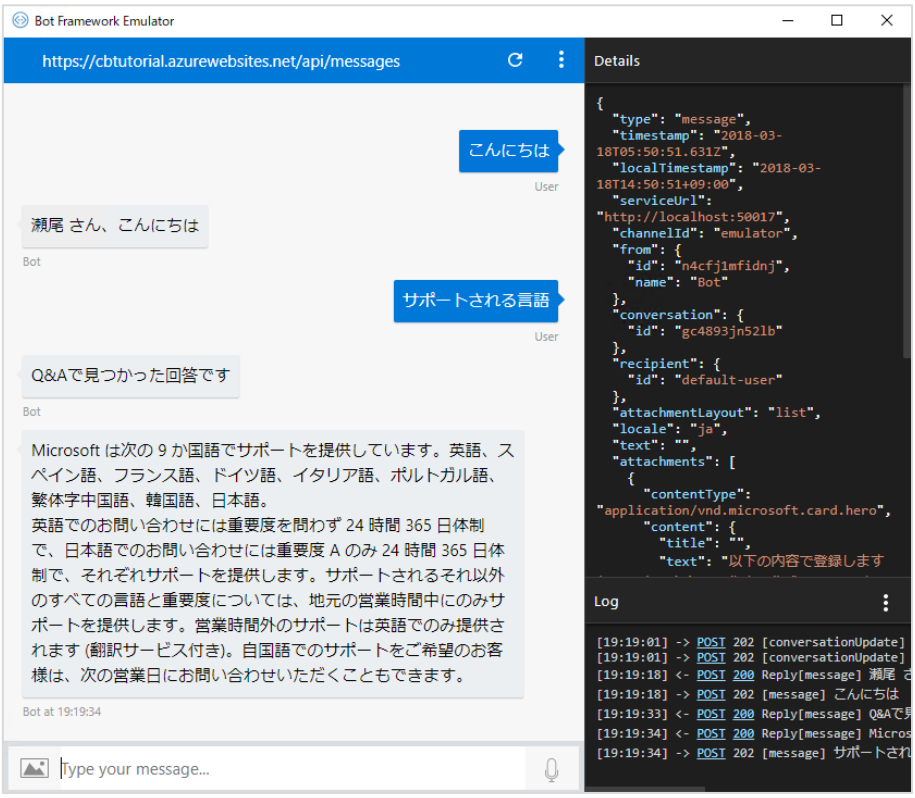


8. Bot Framework Emulator の接続先を以下のように変更します。



設定項目	設定値
URL	https://<Azure に発行した Bot Web App のアドレス>/api/messages
Microsoft App ID	Azure 管理ポータルで取得した App ID (Web.config に設定したものと同じ)
Micorosoft App Password	Azure 管理ポータルで取得した App Password (Web.config に設定したものと同じ)
Locale	ja

9. ローカル環境で Bot アプリケーションをデバッグ実行していた時と同様に、ユーザーと Bot アプリケーションとが会話できることを確認します。



33. Bot Channels

Bot Service の特徴の一つに、利用するチャットクライアントを限定しない点があります。Web Chat、Skype、Facebook Messanger など複数のクライアントから利用できます。
これによりユーザーが使用するデバイスや OS などに依存しなくなり、ユーザーに利用してもらう場所や時間帯の自由度が上がります。

Bot Service のこの機能を、Bot Channels と言います。
Azure 管理ポータルで Bot Service を作った直後は、Web Chat のみ接続されています。チャットクライアントごとにチャンネル設定することで、複数のチャットクライアントから Bot アプリケーションを利用できるようになります。

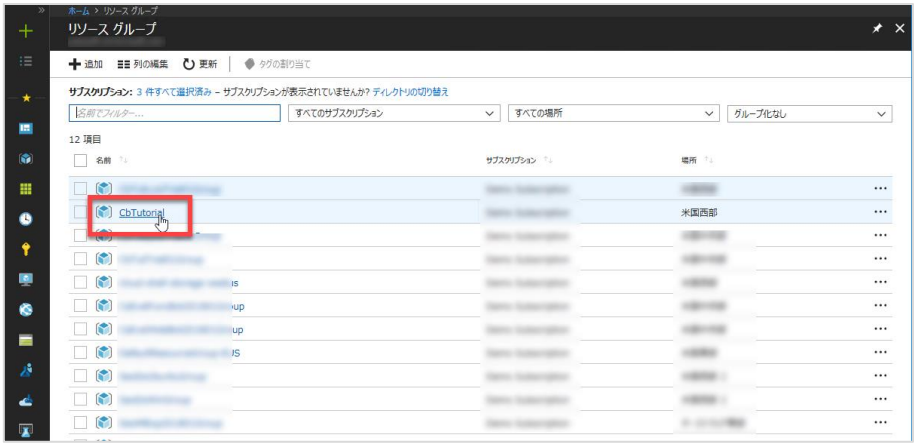
34. Web Chat

Bot Service の Channel 設定は、すべて Azure 管理ポータルからできます。

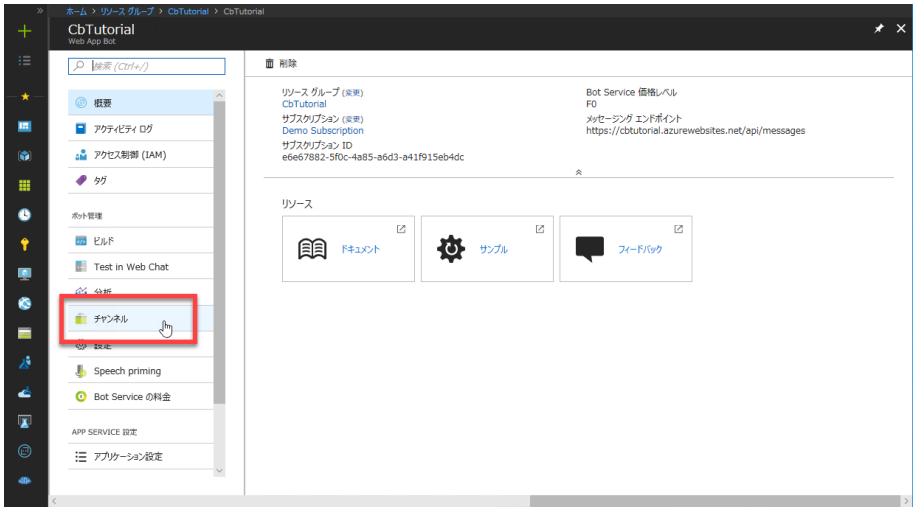
Web Chat は最初から Channel 設定されています。ただし接続に必要なシークレットキーの取得は必要です。
ここではシークレットキーを取得して、実際に Web ブラウザーから Bot アプリケーションに接続するまでの手順を紹介します。

1. Microsoft Azure 管理ポータル (https://portal.azure.com/) を開いて、サインインします。

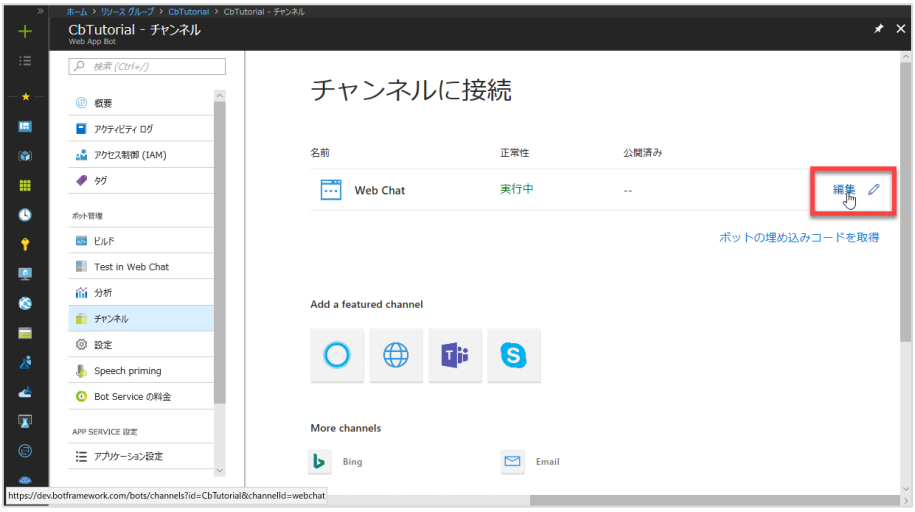
2. この自習書で使っているリソースグループを開きます。



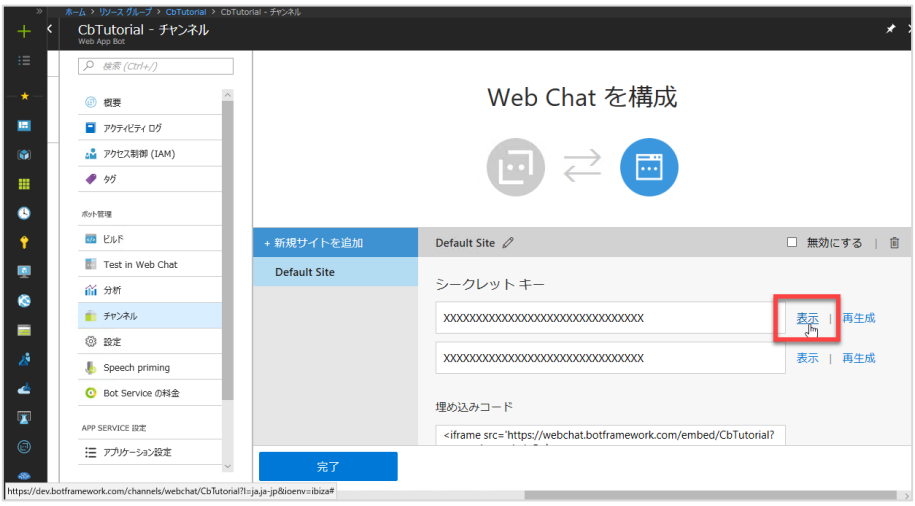
3. [チャンネル] をクリックします。

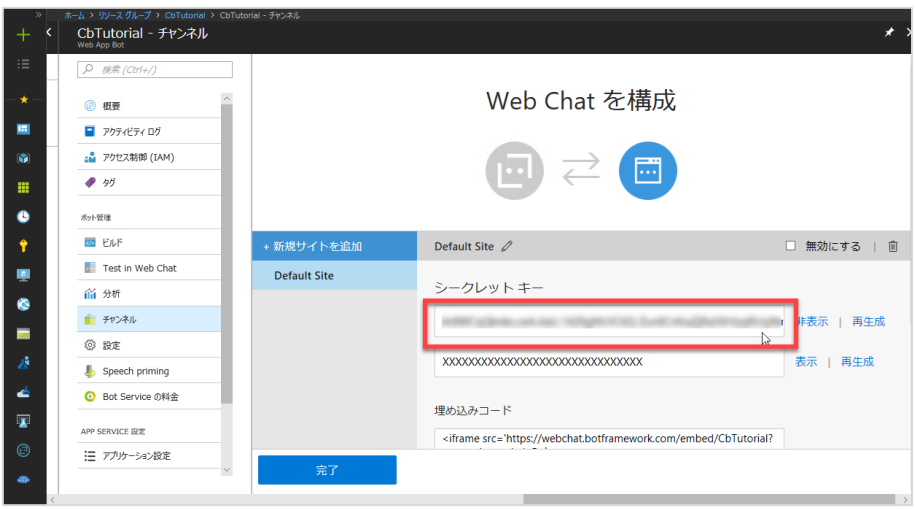


4. [Web Chat] 行の [編集] をクリックします。

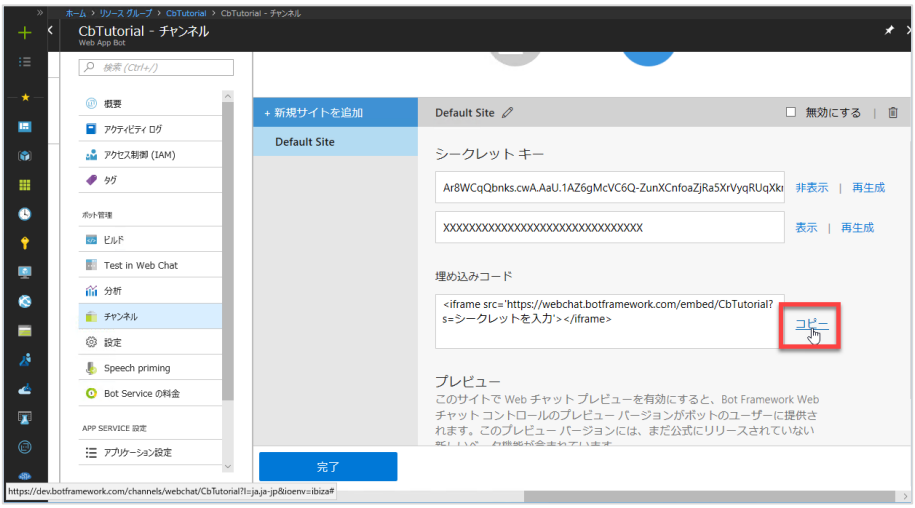


5. シークレットキーの [表示] をクリックします。マスクが外れて実際のシークレットキーが表示されるので、メモ帳などにコピーします。

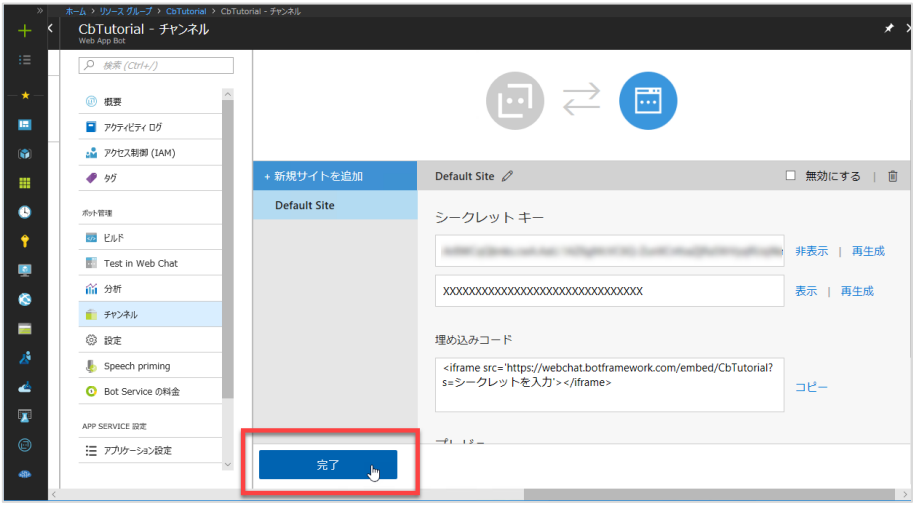




6. シークレットキーのすぐ下に、埋め込みコードが表示されています。[コピー] をクリックしてから、メモ帳などに貼り付けます。

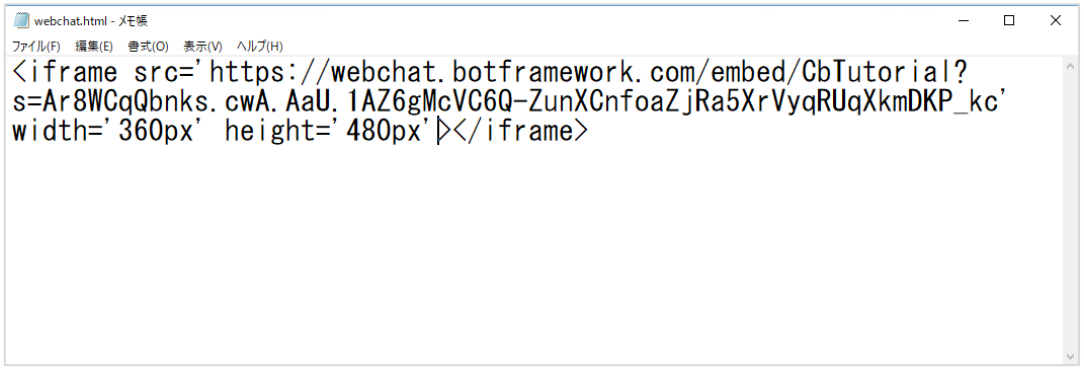


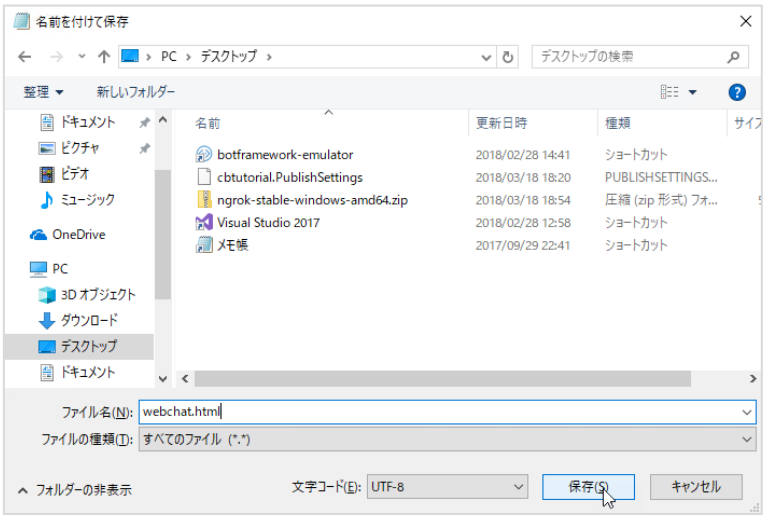
7. 最後に [完了] ボタンをクリックします。



8. 埋め込みコードを貼り付けたメモ帳を保存します。例えば “webchat.html” などの名前にします。

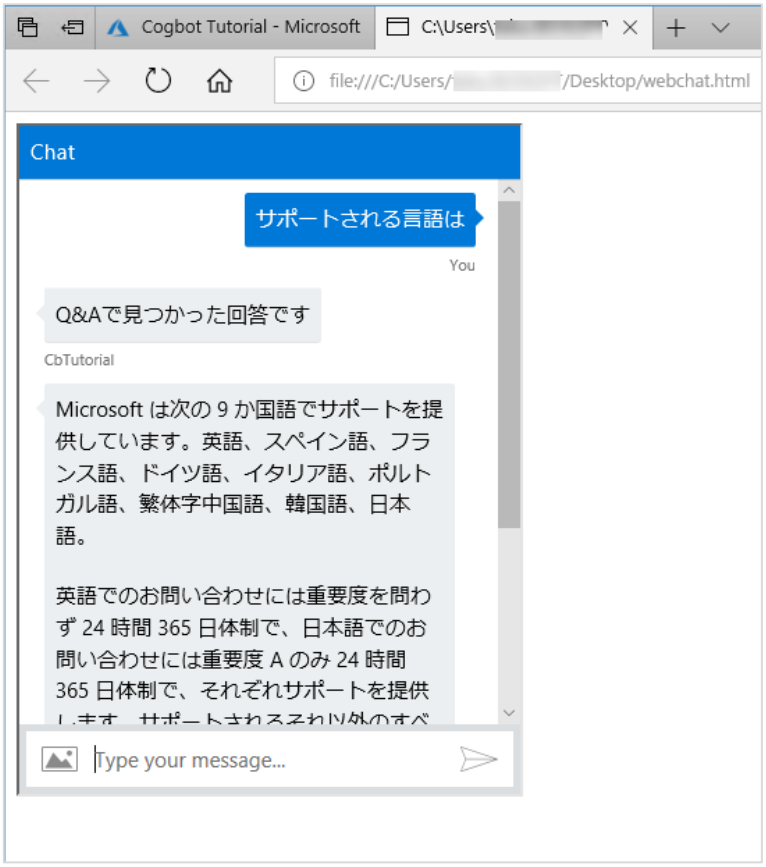
ただし、埋め込みコードの中の “シークレットキーを入力” の部分には、先ほど取得したシークレットキーを貼り付けます。
なおチャットクライアントのサイズが小さいので、width や height などを追加するのがいいでしょう。





この自習書の範囲を超えて Web Chat を利用する際には、貼り付ける先のページのスタイルに合わせるなどの対応が必要です。

9. 保存した HTML ファイルを Web ブラウザーで開いて、メッセージを入力します。Bot アプリケーションが応答を返してくれます。



デフォルトの埋め込みコードは認証が行われていません。
このため、ユーザー情報（名前）を永続化する機能や、タスクの保存、閲覧機能は利用できません。
Web ブラウザーを閉じると、入力した名前などの情報は失われます。

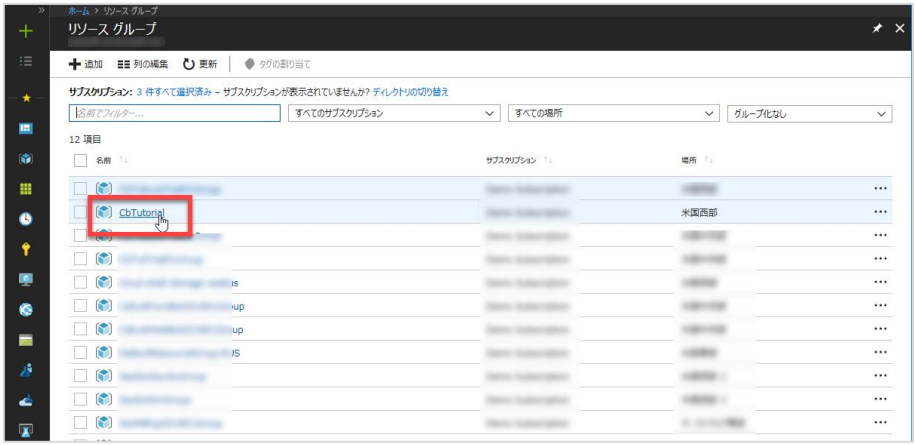
ワンポイント
デフォルトの埋め込みコードはシークレットキーが平文で記述されています。 利用範囲によっては、シークレットキーを HTML ヘッダーで渡すなどの工夫が必要です。

35. Skype

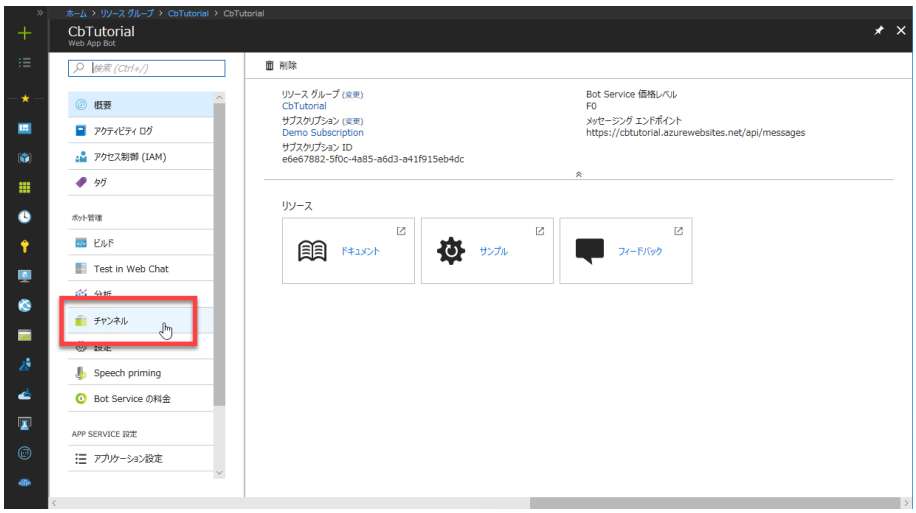
クライアントとして Skype を利用してみます。

1. Microsoft Azure 管理ポータル (https://portal.azure.com/) を開いて、サインインします。

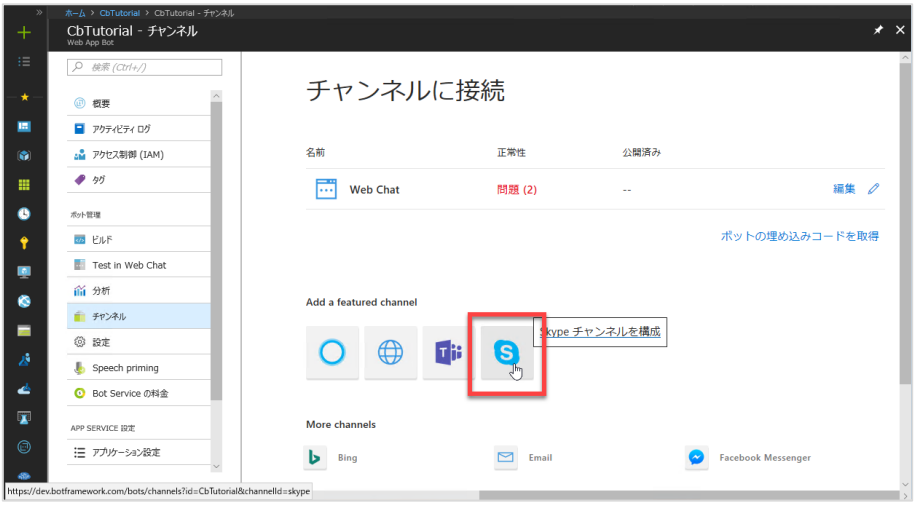
2. この自習書で使っているリソースグループを開きます。



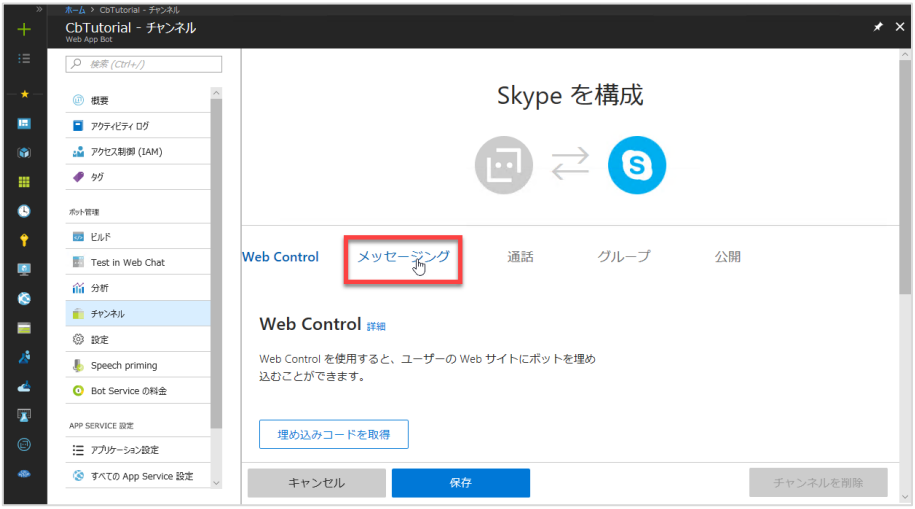
3. [チャンネル] をクリックします。



4. [Add a featured channel] の [Skype チャンネルを構成] をクリックします。

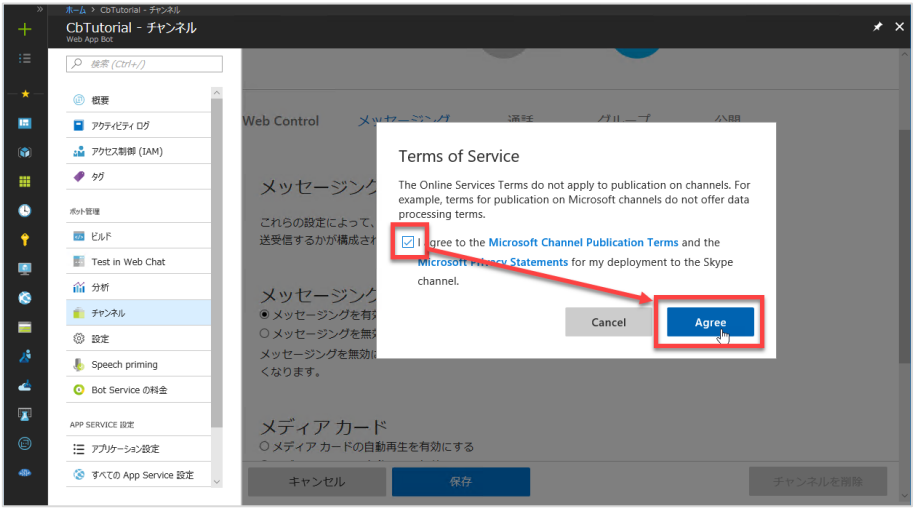


5. [Skype を構成] ページで [メッセージング] をクリックします。

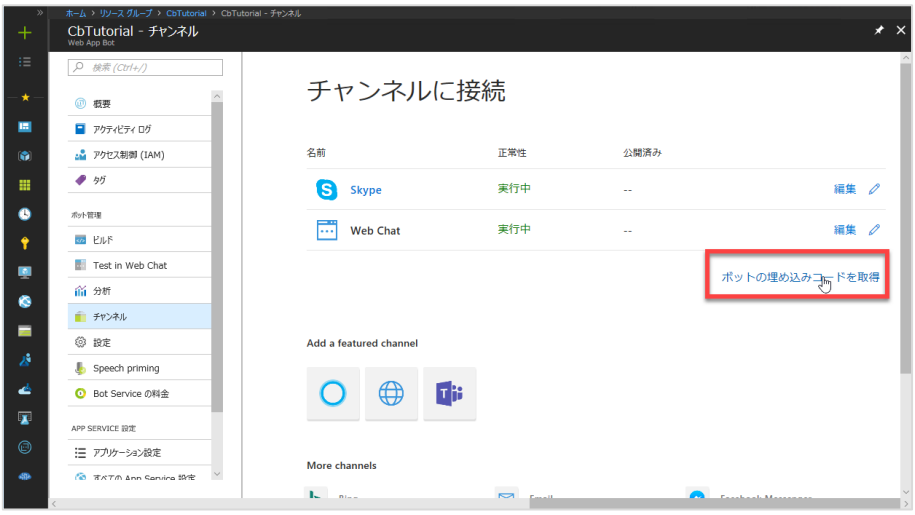


6. [メッセージングを有効にする] が選択されていることを確認して、[保存] をクリックします。

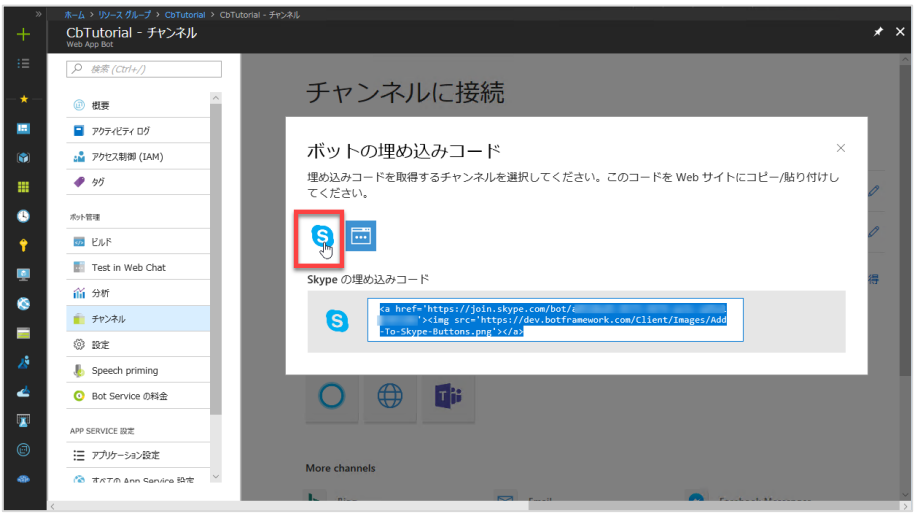
7. 利用許諾を確認して [Agree] をクリックします。



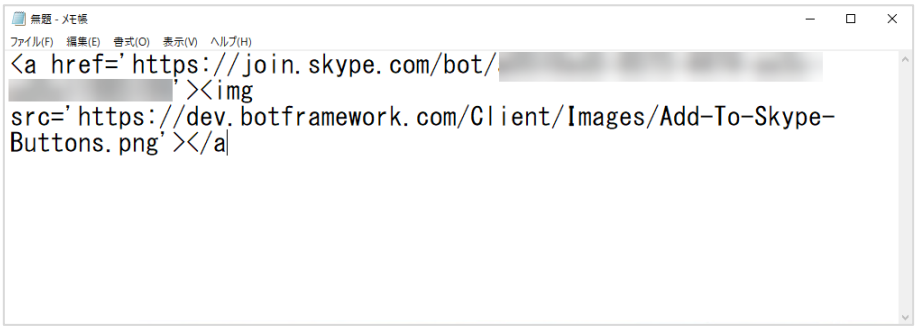
8. [ボットの埋め込みコードを取得] をクリックします。



9. [ボットの埋め込みコード] が表示されるので、[Skype] をクリックして、埋め込みコードをメモ帳などにコピーします。



10. 埋め込みコードを貼り付けたメモ帳を保存します。例えば “skype.html” などの名前にします。

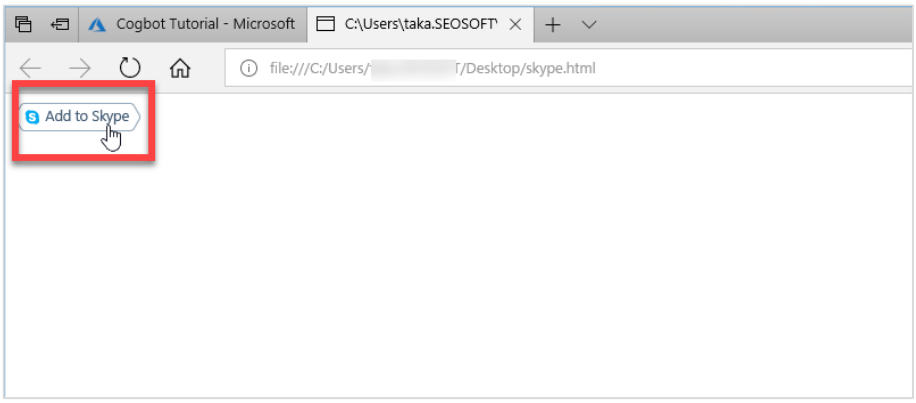


利用者には、

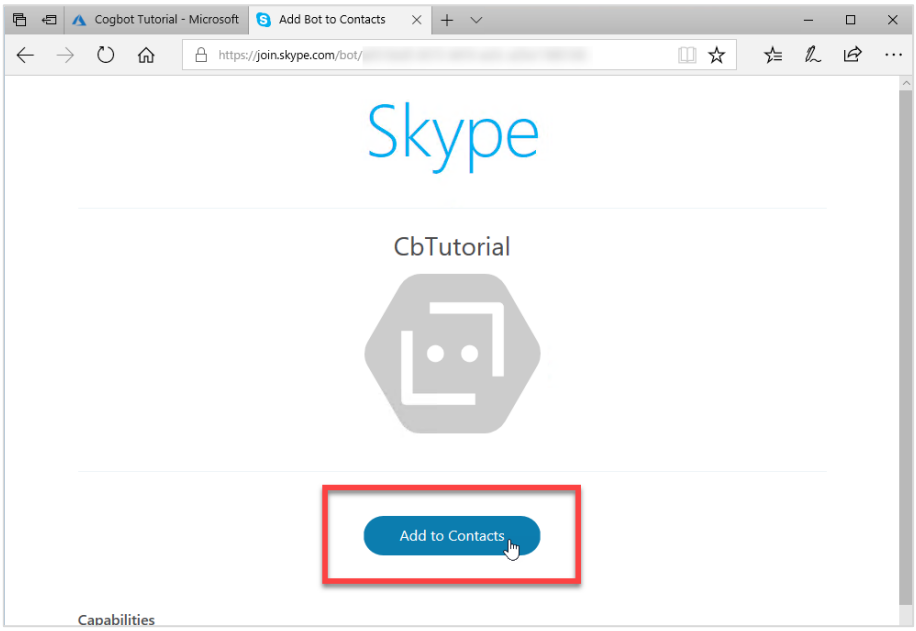
- ポータルサイトに埋め込む
- メール本文に埋め込む

などの方法で通知します。

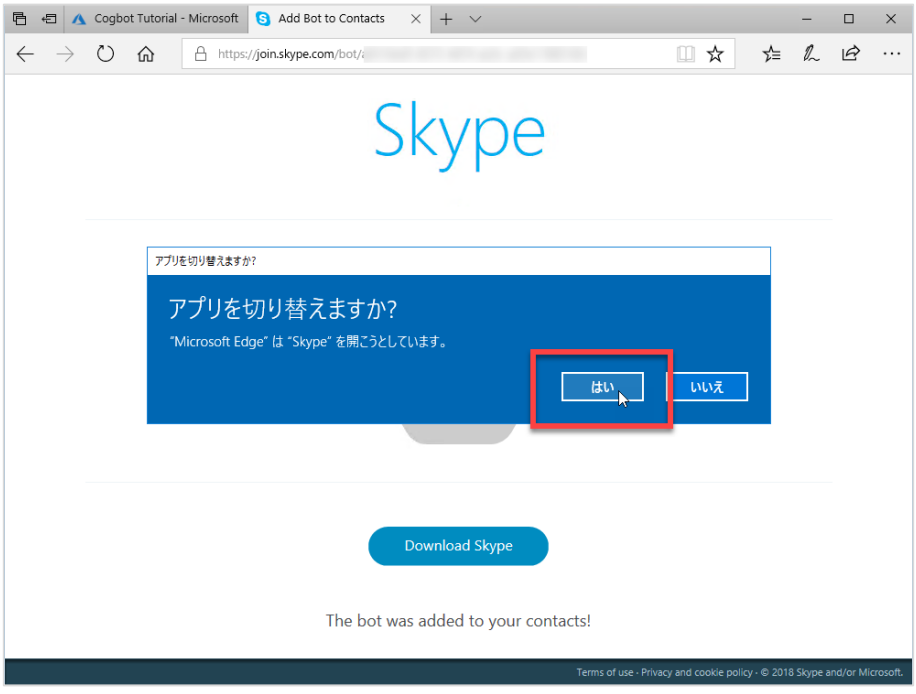
11. 保存した “skype.html” を Web ブラウザーで開き、[Add to Skype] をクリックします。



12. [Add to Contancts] をクリックします。

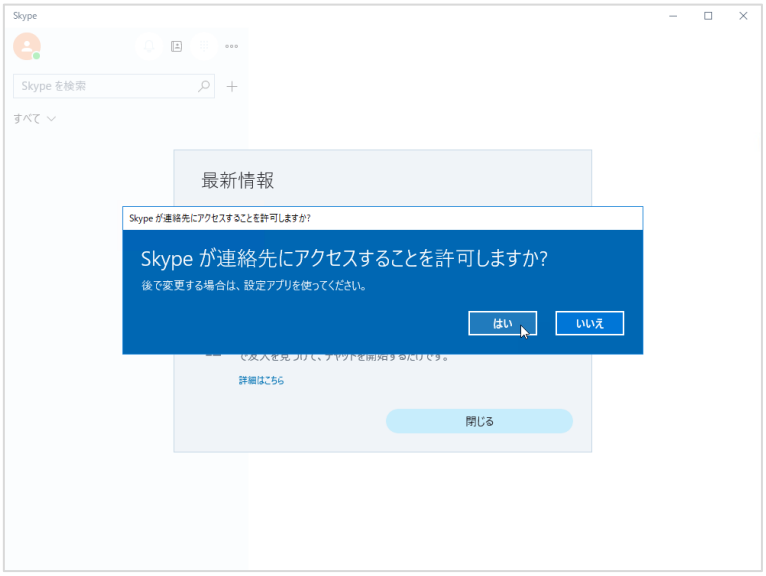


13. [アプリを切り替えますか？] に対して、[はい] を選択します。

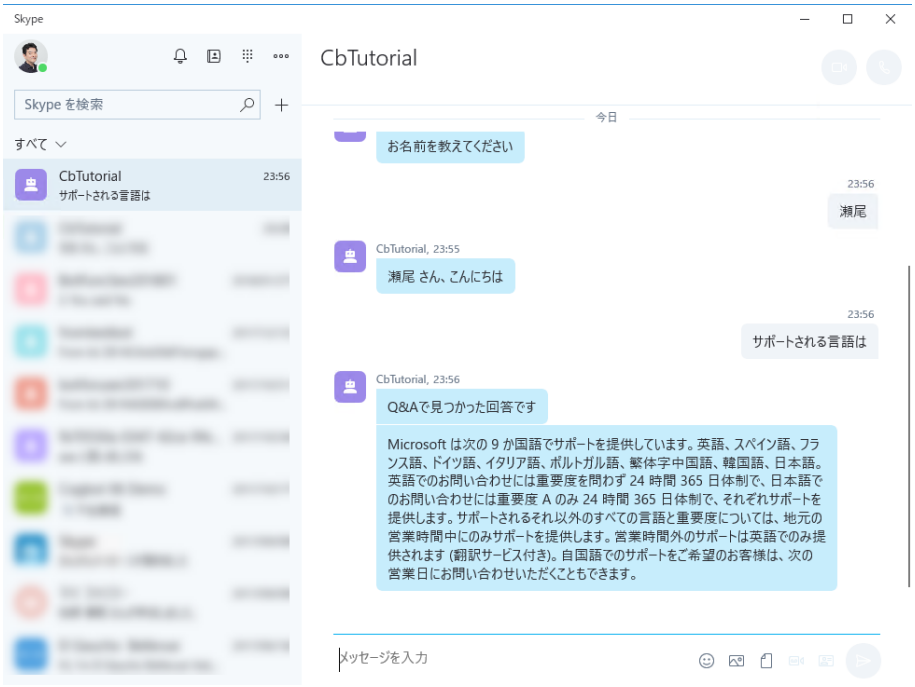


14. （まだ Skype にサインインしていない場合）Skype にサインインします。

15. 作業 PC での Skype の利用状況によっては、Skype が連絡帳、マイク、カメラへのアクセスを求めてくるかもしれません。Bot アプリケーション利用には必須ではありませんので、適宜選択して進めます。



16. Bot アプリケーションとのチャット画面が開きます。



Skype を Bot アプリケーションに接続するのは非常に簡単です。
ただしビジネスユースには利用が難しいこともあるかもしれません。続いて Microsoft Teams で Bot アプリケーションと対話する方法を紹介します。

36. Microsoft Teams

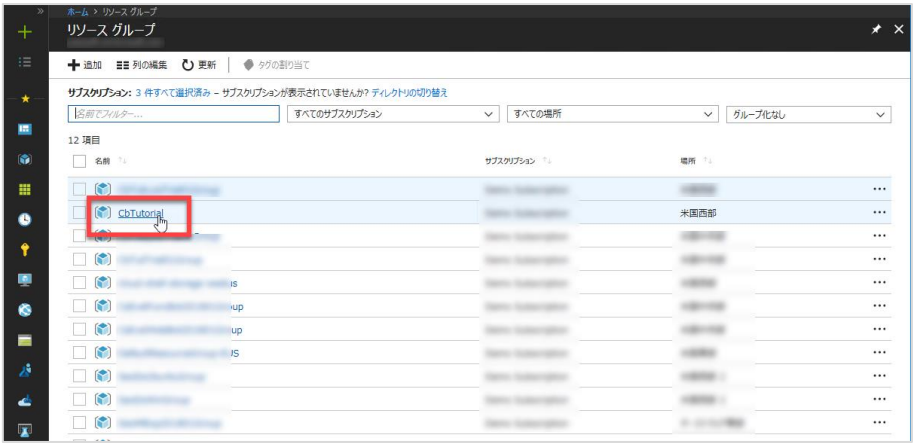
Skype は非常に手軽ですが、業務ではなかなか使いづらいこともあります。
Office 365 サブスクリプションをお持ちであれば、Microsoft Teams から Bot アプリケーションに接続することができます。

実際に Microsoft Teams をチャンネル設定してみます。
手順は Skype とほとんど同じです。こちらも簡単に利用できます。

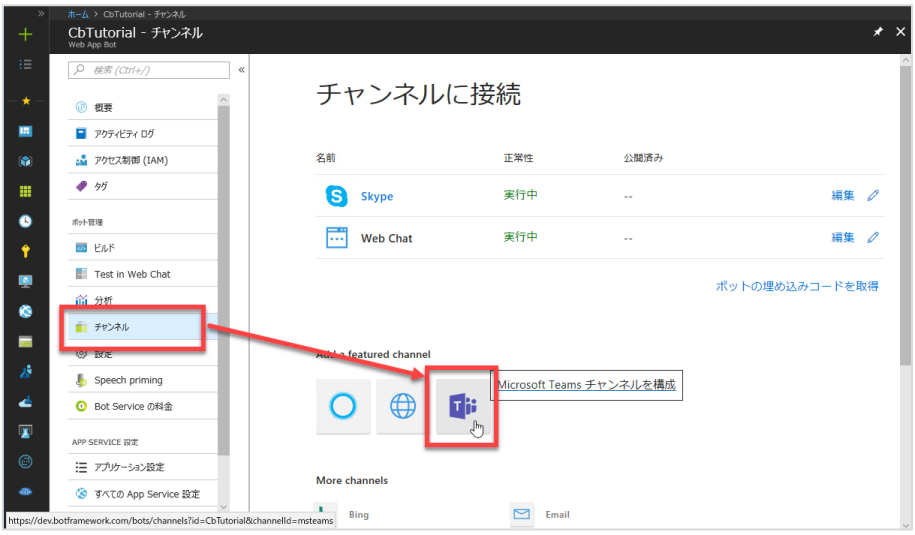
ただし以下の手順は、Office 365 サブスクリプションをお持ちの方のみ実施できます。

1. Microsoft Azure 管理ポータル (https://portal.azure.com/) を開いて、サインインします。

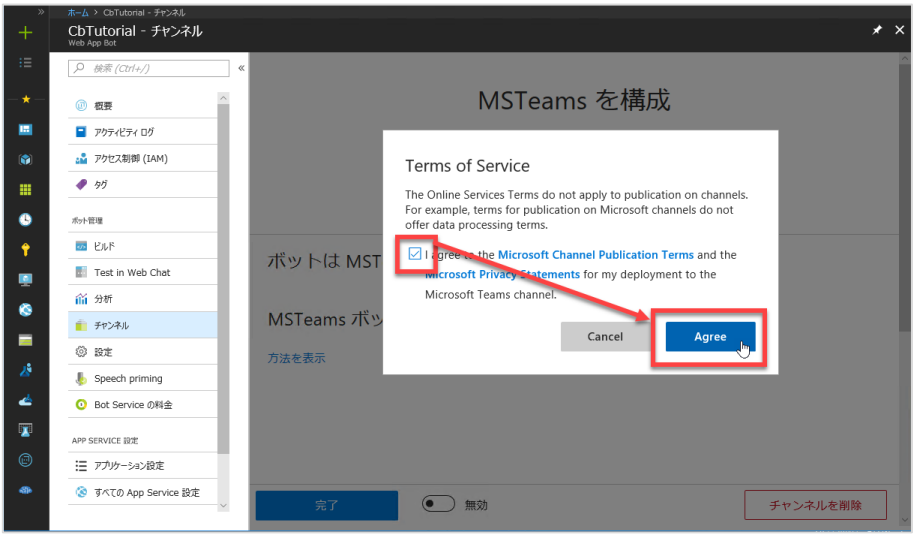
2. この自習書で使っているリソースグループを開きます。



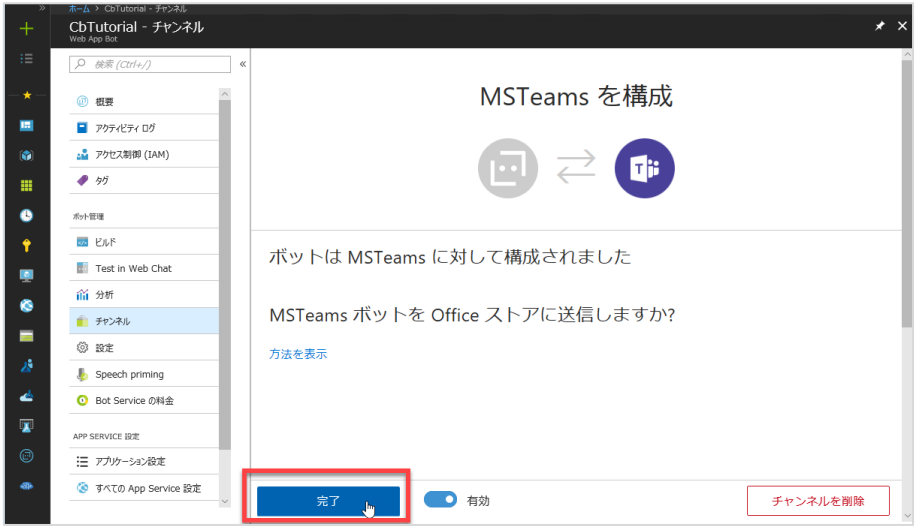
3. [チャンネル] をクリックして、"Add a featured channel" の [Microsoft Teams] をクリックします。



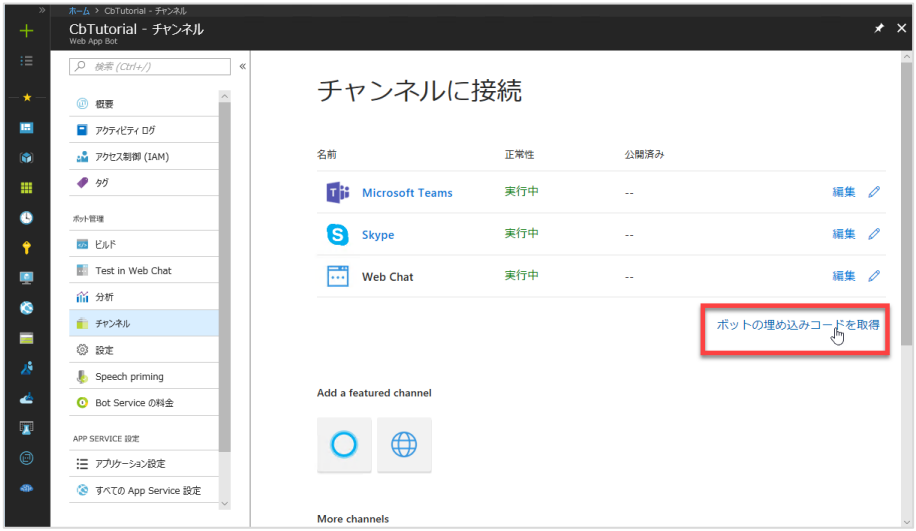
4. 利用許諾を確認して [Agree] をクリックします。



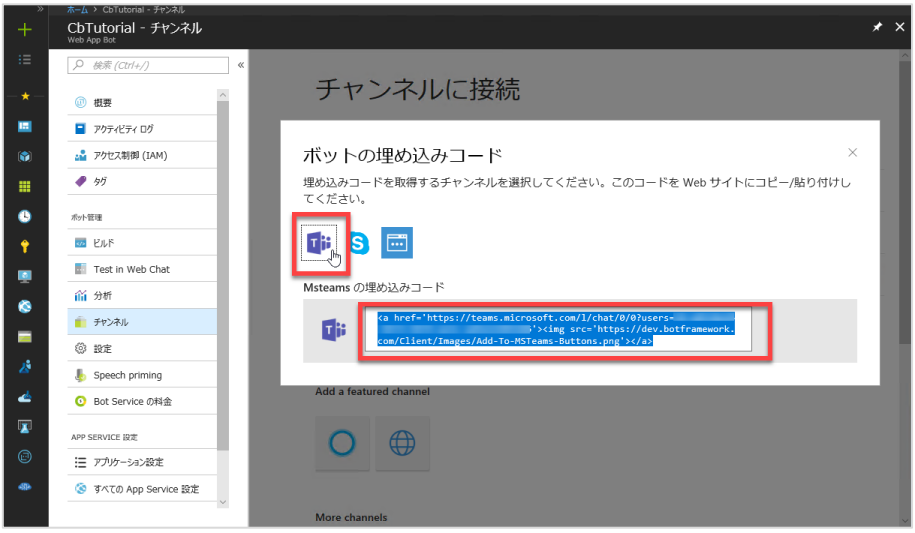
5. “MSTeams を構成” で [完了] をクリックします。



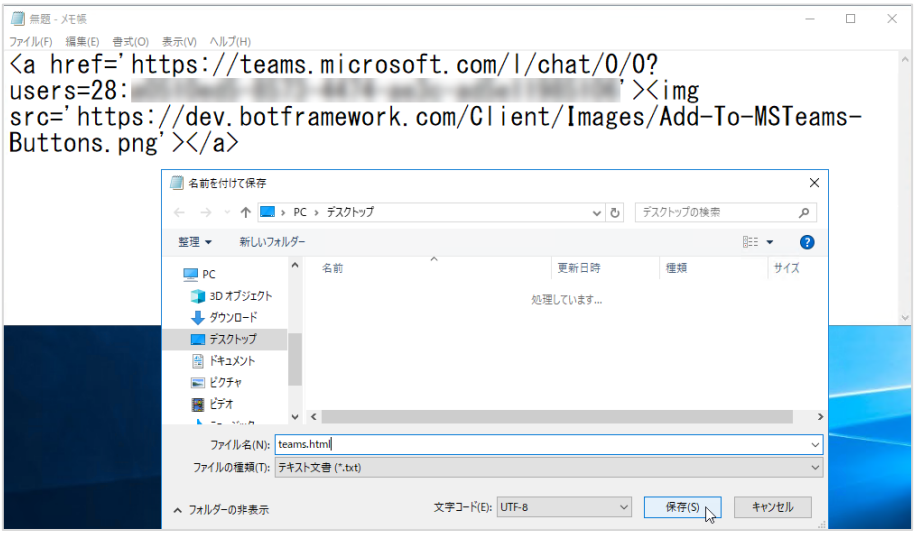
6. “チャンネルに接続” 画面で、[ボットの埋め込みコードを取得] をクリックします。



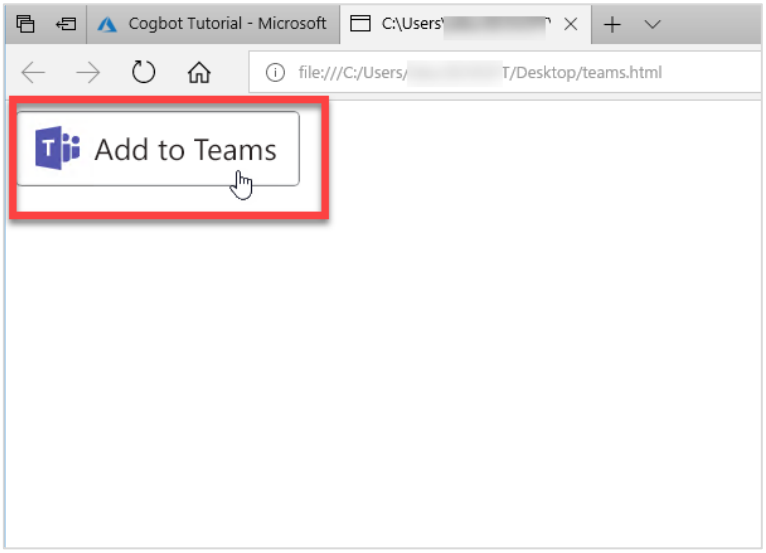
7. [ボットの埋め込みコード] が表示されるので、[Msteams] をクリックして、埋め込みコードをメモ帳などにコピーします。



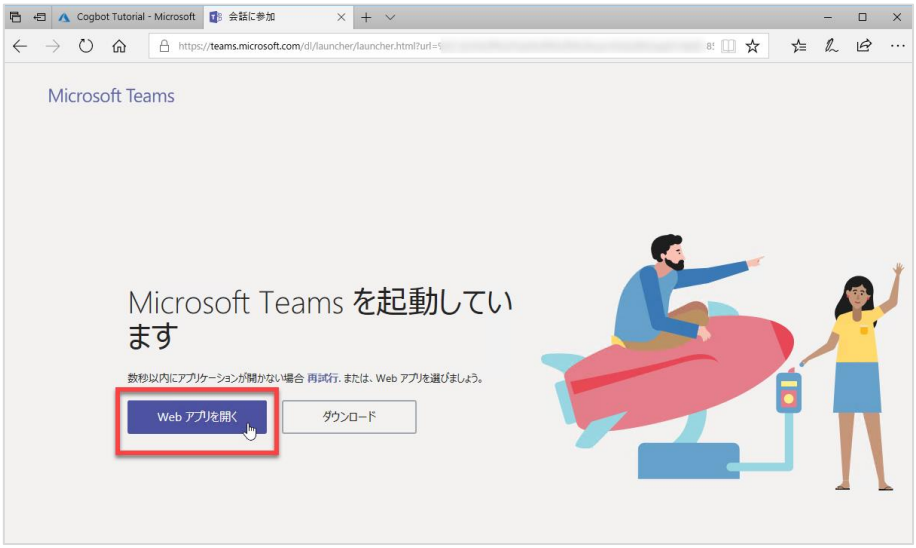
8. 埋め込みコードを貼り付けたメモ帳を保存します。例えば “teams.html” などの名前にします。



9. 保存した “teams.html” を Web ブラウザーで開き、[Add to Teams] をクリックします。



10. Microsoft Teams を起動します。[Web アプリを開く]、デスクトップ版を [ダウンロード] するのどちらかを選択します。すでにデスクトップ版の Teams をダウンロード、インストール済みの場合は、自動的にそれが開きます。



11. Bot アプリケーションとのチャット画面が開きます。

Microsoft Teams は、ビジネス用途にマッチしたチャットクライアントとして利用可能です。
Office 365 サブスクリプションをお持ちの場合は、積極的に利用したいツールです。

以上で、Bot アプリケーション開発の最初のステップから、チャットクライアントのチャンネル設定までの一連の作業手順を紹介しました。
他の資料ではあまり触れられない、複数機能を持つ Bot アプリケーションの開発についても紹介しています。
この自習書が、Azure Bot Service を使ったアプリケーション開発を始めるきっかけになれば幸いです。

参考文献

この自習書で利用したサービス等のサイトをまとめます。自習書の復習と今後の学習の際に参考にしてください。

37. 各サービスの情報

1. Microsoft Azure
   - Microsoft Azure
     https://azure.microsoft.com/ja-jp/
     Azure の全てのサービスの紹介、各サービスのドキュメントへのリンクなどが掲載されています。

2. Cognitive Services
   - Cognitive Services
     https://azure.microsoft.com/ja-jp/services/cognitive-services/
     Cognitive Services 全体の公式サイトです。各サービスの詳細やデモ、ドキュメントへのリンクなどが掲載されています。
   - Language Understanding (LUIS)
     https://azure.microsoft.com/ja-jp/services/cognitive-services/
     自然言語を理解する機能です。「学習」を行うのもこのサイトで行います。
   - QnA Maker
     https://qnamaker.ai/
     アプリケーションに、自然言語対応の FAQ 機能を付加するものです。ナレッジの強化もこのサイトで行います。

3. Bot Service
   - Bot Service
     https://azure.microsoft.com/ja-jp/services/bot-service/
     Bot アプリケーションの構築、開発、管理を行うサービスです。この自習書の中心となった技術です。
   - Bot Framework
     https://dev.botframework.com/
     GetStarted を含むドキュメントや開発チームのブログが掲載されています。一歩進んだ Bot アプリケーション開発をする際に参照してください。
   - Bot Framework Emulator
     https://docs.microsoft.com/en-us/azure/bot-service/bot-service-debug-emulator
     Bot アプリケーション開発で必須ともいえるチャットクライアントです。

4. Visual Studio 2017
   - Visual Studio
     https://www.visualstudio.com/ja/
     Visual Studio の公式サイトです。Visual Studio、Visual Studio Code のインストーラーをダウンロードしたり、サブスクリプションの特典を利用できたりします。

5. その他
   - ngrok
     https://ngrok.com/
     Bot Framework Emulator から Microsoft Azure にデプロイした Bot アプリケーションに接続する際に必要です。