Microsoft Azure + Bot Framework による
FAQ ボット アプリケーションの構築

概要

ユーザーとの対話を可能にするチャットボット (Chatbot、以下”ボット”) が近年注目を浴びています。その理由として、複雑でない定型的なくり返し業務を 24 時間 365 日実行することはコンピューターにとって苦にならないからです。例えば、宅配ピザの受付のように決まった手順でオーダーを取る、または、これまでにお問い合わせの多かった内容をまとめた FAQ リストから回答を返答する、このような業務をボットに任せることで、私たちはより時間を掛けるべき業務に集中することが可能になります。

Microsoft Bot Framework は 対話型メッセージングを行う機能をまとめたフレームワークで、Web アプリケーションとして稼働させるテンプレートが用意されています。(.NET C#版 および Node.JS 版 があります。)

また、Microsoft Azure は Bot Framework で構築されたアプリケーションをホストする App Service (Web App)、データを保存するストレージ (Azure Storage、Azure SQL Database、Azure CosmosDB) だけでなく、高度な全文検索を行う Azure Search、“AI パーツ”として画像や言語の解析を含むインテリジェンス機能を API で利用可能な Cognitive Services といった機能をすぐに利用開始できます。

このラボでは、Bot Framework を使用して、ユーザーと対話を行うボットアプリケーションを構築します。ボットのインテリジェンス部分として、自然言語で入力された文章の分類とエンティティ抽出を行う Cognitive Services Language Intelligent Service (LUIS)、FAQ のリストを保存するフレキシブルな DB として Azure CosmosDB、FAQ リストの全文検索を行う Azure Search を導入し、社内 IT サポートチケット受付機能社および FAQ 検索をボットで提供するアプリケーションを実装します。

目的

このハンズオン ラボでは、以下の方法について学習します。

- Bot Framework (.NET C#版) を使用して、ボットアプリケーションを作成する
- Cognitive Services LUIS を利用して、ボットに自然言語処理を実装する
- Azure CosmosDB および Azure Search を利用して、ボットにリスト検索機能を実装する

前提条件

このハンズオン ラボを実行するには、以下が必要です。

- マイクロソフトアカウント
- アクティブな Microsoft Azure サブスクリプション。これがない場合は、無料試用版にサインアップしてください
- Visual Studio 2017 Community エディションまたはこれ以降

目次

このラボの推定所要時間: 60 分

演習 1: Bot Builder SDK for .NET による初めての "おうむ返し" ボットの作成

この演習では、Bot Builder SDK for .NET を使用してボットを構築し、それを Bot Framework Emulator でテストする方法を示します。

Bot Builder SDK for .NET は、Visual Studio と Windows を使用してチャットボット (Chatbot) を開発するためのフレームワークです。SDK では C# を活用し、.NET 開発者にとってなじみのある方法で強力なボットを作成する手段を提供します。

このフォルダーの中には、Visual Studio ソリューションと、この演習のステップで作成するコードが入っています。このソリューションは、演習を進めるにあたってさらにヒントが必要な場合に、ガイダンスとして使用できます。

Bot Builder SDK for .NET は現在 C# 版となります。 Visual Studio for Mac はサポートされていません。

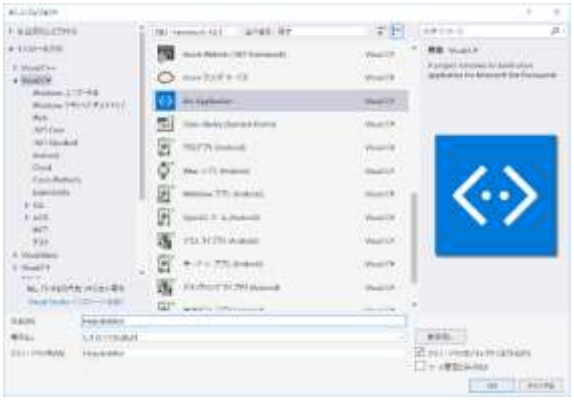
タスク 1: Bot Application テンプレートを使用して新規ボットを作成する

1. Bot Application テンプレートをダウンロードし、zip ファイルを Visual Studio 2017 プロジェクト テンプレート ディレクトリに保存してテンプレートをインストールします。Visual Studio 2017 プロジェクト テンプレート ディレクトリは通常、以下の場所にあります。

%USERPROFILE%¥Documents¥Visual Studio 2017¥ Templates¥ ProjectTemplates¥Visual C#¥

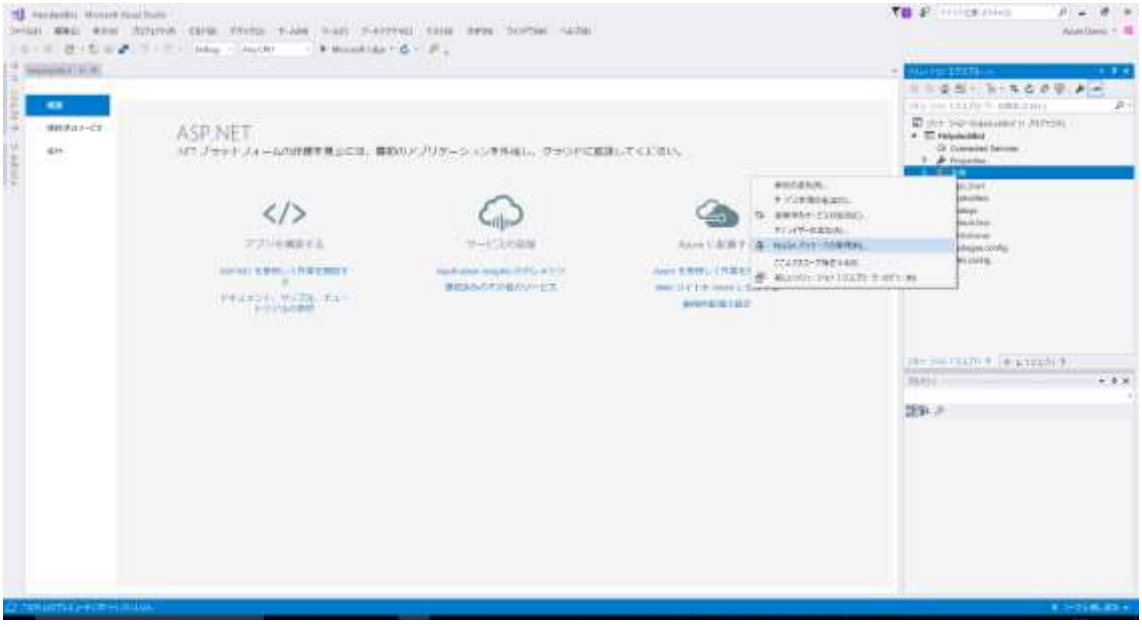
2. Visual Studio を開き、ツールバーの [ファイル] > [新規作成] > [プロジェクト] をクリックして、新規プロジェクトを作成します。

3. C#の Bot Application テンプレートを選択します。次以降の演習で名前空間の問題が発生するのを回避するために、プロジェクト名には HelpDeskBot を使用します。

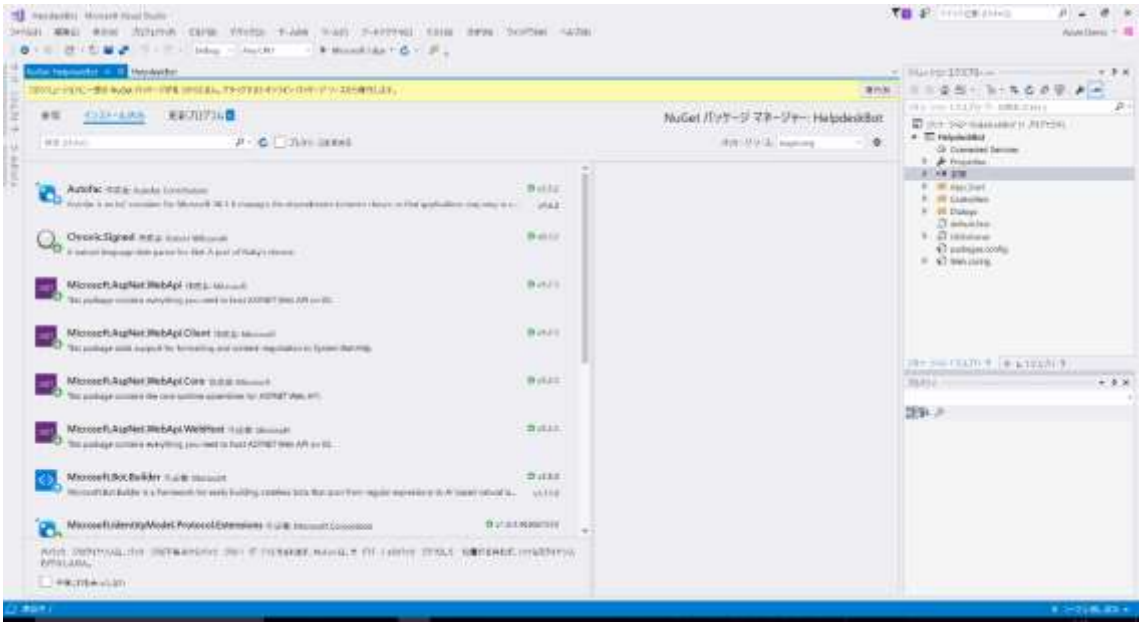


Bot Application テンプレートを使用することで、単純なボットの作成に必要なすべてのコンポーネントが既に含まれているプロジェクトを作成することになります。これには、Microsoft.Bot.Builder NuGet パッケージに含まれる Bot Builder SDK for .NET への参照も含まれます。

4. ソリューション エクスプローラーでプロジェクトの [参照] フォルダーを右クリックして、[NuGet パッケージの管理] をクリックします。



5. 「このソリューションに一部の NuGet パッケージが見つかりません...」と表示される場合は、その横にある [復元] をクリックして、パッケージを復元します。



タスク 2: ボットのコードを調べる

Bot Application テンプレートを使用することで、プロジェクトには、このチュートリアルでボットを作成するために必要なコードがすべて含まれます。追加のコードを記述する必要はありません。しかし、ボットのテストに移る前に、ボット アプリケーション テンプレートのコードの一部を少し見てみましょう。

6. 最初に、Controllers¥MessagesController.cs 内の Post メソッドがユーザーからメッセージを受信し、RootDialog.cs を呼び出します。

7. ダイアログは、会話のモデル化と会話のフローの管理に使用されます。各ダイアログは、IDialog を実装する C# クラスで自らの状態をカプセル化するアブストラクションです。ダイアログは、別のダイアログで構成して再利用を最大化することができ、ダイアログのコンテキストは、任意の時点の会話でアクティブなダイアログのスタックを維持します。また、ダイアログで構成される会話はコンピューター間で移植可能なため、ボットの実装の規模の調整が可能になります。

MessageControllers.cs
[BotAuthentication] public class MessagesController : ApiController { 　　/// <summary>

```
        /// POST: api/Messages
        /// Receive a message from a user and reply to it
        /// </summary>

        public async Task<HttpResponseMessage> Post([FromBody]Activity activity)
        {

            if (activity.Type == ActivityTypes.Message)
            {
                await Conversation.SendAsync(activity, ()
                    => new Dialogs.RootDialog());
            }
            else
            {
                HandleSystemMessage(activity);
            }

            var response = Request.CreateResponse(HttpStatusCode.OK);
            return response;
        }
        :
       (中略)
        :
}
```

8. RootDialog はメッセージを処理し、応答を生成します。Dialogs¥RootDialog.cs 内の MessageReceivedAsync メソッドは、ユーザーのメッセージの先頭に “You sent”、末尾に “which was ## characters” (## はユーザーのメッセージの文字数) というテキストを付けて、おうむ返しに返信を送ります。

9. ダイアログは、ボットのロジックを整理し、会話のフローを管理するのに役立ちます。ダイアログはスタック化して配置され、スタックの最上位のダイアログは、ダイアログが閉じるか別のダイアログが呼び出されるまで、すべての受信メッセージを処理します。

RootDialog.cs

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

```
            return Task.CompletedTask;
        }

        private async Task MessageReceivedAsync
            (IDialogContext context, IAwaitable<object> result)
        {
            var activity = await result as Activity;

            // calculate something for us to return
            int length = (activity.Text ?? string.Empty).Length;

            // return our reply to the user
            await context.PostAsync
                ($"You sent {activity.Text} which was {length} characters");

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

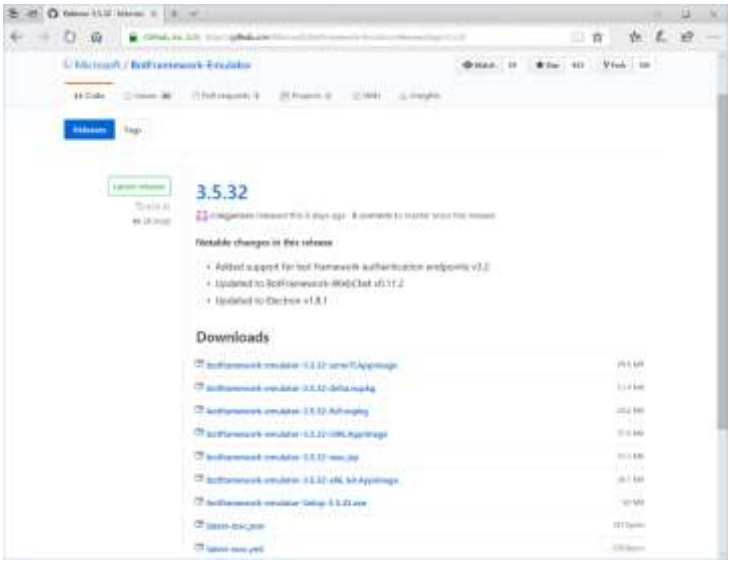
- ボットの会話処理を行うためのダイアログは、IDialogContext インターフェイスが提供する Conversation Flow Control メソッド (ここでは RootDialog)、そして IDialogContext メソッドを使用して会話フローを管理する PromptDialog ヘルパー メソッドを使用して、連結する一連のメソッドで構成されています。
- 最初に会話が始まったときは、ダイアログに状態が含まれないので、Conversation.SendAsync は RootDialog を構成し、StartAsync メソッドを呼び出します。StartAsync メソッドは継続デリゲートで IDialogContext.Wait を呼び出し、そのメソッド (この場合は MessageReceivedAsync メソッド) を新しいメッセージの受信時に呼び出すように指定します。
- Bot Framework SDK では、ユーザーからの入力の収集を簡単にするためのビルトイン プロンプトのセットを提供しています。MessageReceivedAsync メソッドはメッセージを待機し、受信すると、ユーザーにあいさつを返し、PromptDialog.Text() を呼び出して、問題について説明するよう求めます。
- また、応答はフレームワークによりダイアログ インスタンスに保持されます。これは [Serializable] としてマークされることに注意してください。これは、ダイ

アログのステップ間で一時情報を保存するために重要です。

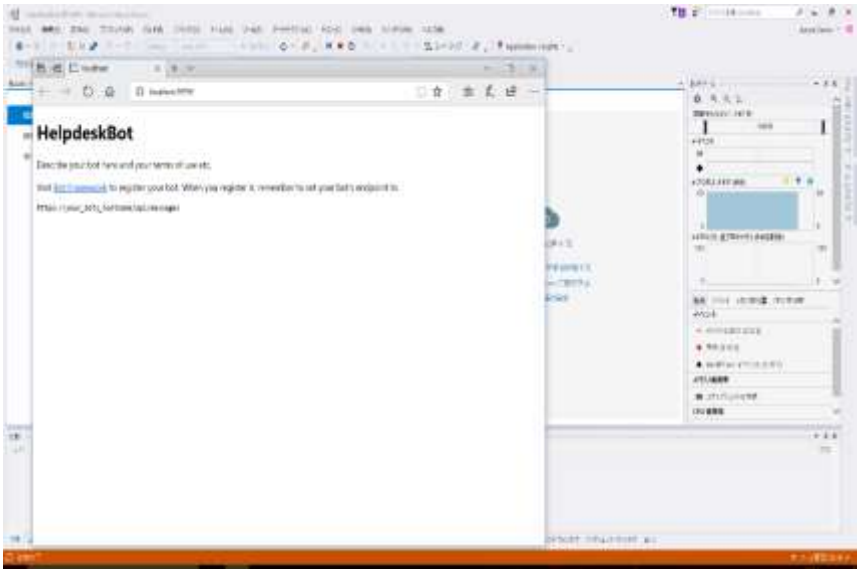
タスク 3: ボットをテストする

次に、Bot Framework Emulator を使用してボットをテストし、動作の様子を見てみましょう。このエミュレーターは、localhost 上のボット、またはトンネルを通じてリモートで実行しているボットをテストおよびデバッグできる、デスクトップ アプリケーションです。エミュレーターは、Web チャットの UI に表示されるとおりにメッセージを表示し、JSON 要求をログに記録し、ユーザーがボットとメッセージのやり取りを行うの同様に応答します。

10. まず、Bot Framework Emulator の Web サイトから botframwork-emurator-setup-xxx.exe というインストーラーをダウンロードし、ローカルで起動してインストールを行います。

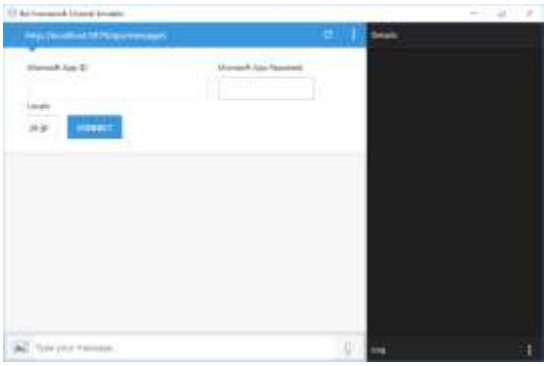


11. エミュレーターのインストール後、IIS Express をアプリケーション ホストとして使用して、Visual Studio でこれまでのタスクで作成したボットを起動します。Visual Studio の [実行] ボタンをクリックすると、Visual Studio でアプリケーションが構築されて localhost に展開、Web ブラウザーが起動してアプリケーションの default.htm ページを表示します。



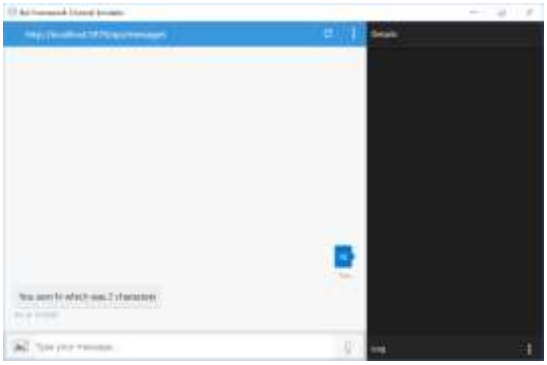
Windows ファイアウォールの警告が表示される場合は、[アクセスを許可] をクリックします。

12. 次に、エミュレーターを起動し、ボットに接続します。アドレス バーに http://localhost:3979/api/messages と入力します。これは、ボットがローカルにホストされたときにリッスンする既定のエンドポイントです。[ロケール] を ja-jp に設定し、[Connect] をクリックします。



ボットをローカルで実行しているので、[Microsoft App ID] と [Microsoft App Password] を指定する必要はありません。これらのフィールドは、今のところ空白のままにしてかまいません。この情報は、Bot Framework Portal にボットを登録する際に取得します。

13. 何かメッセージを送信します。送信したメッセージに対して、先頭に "You sent"、末尾に "which was ## characters" (## はユーザーのメッセージの文字数) というテキストを付けて、”おうむ返し” にボットが応答するのを確認できます。



14. ブラウザーを終了し、Visual Studio の [停止] ボタンをクリックして、アプリケーションを停止します。

演習 2: ボットによるヘルプ デスク チケットの送信

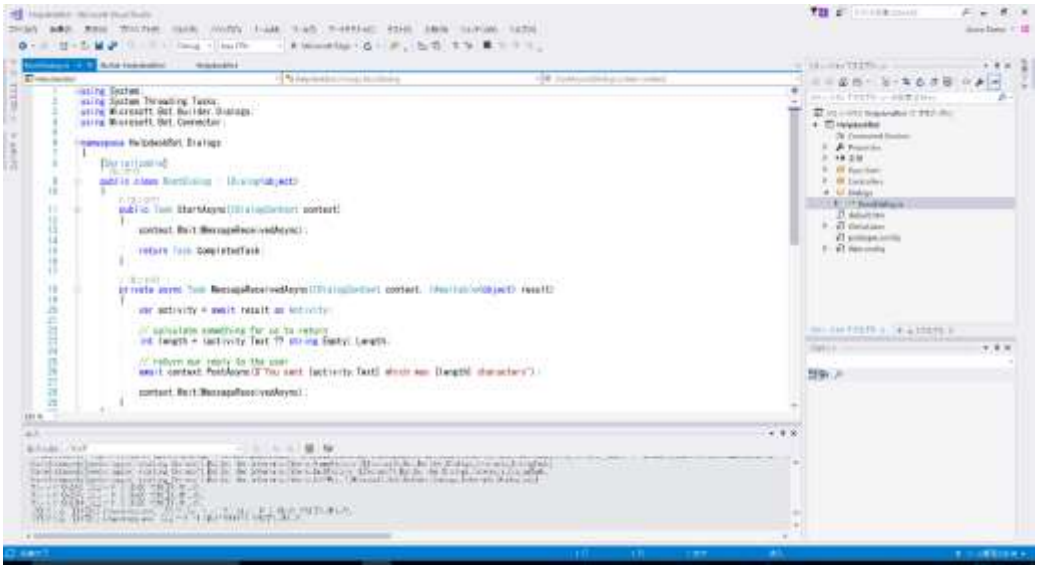
この演習では、ボットに会話機能を追加して、ヘルプ デスク チケットの作成をユーザーに案内する方法を学習します。

このフォルダーの中には、Visual Studio ソリューションと、この演習のステップで作成するコードが入っています。このソリューションは、演習を進めるにあたってさらにヒントが必要な場合に、ガイダンスとして使用できます。

タスク 1: ボットへの会話の追加

このタスクでは、演習 1 で作成したアプリケーションのコードを変更し、ユーザーに一連の質問を行い、アクションを実行するように変更します。

1. 前の演習で作成したソリューションを開き、ソリューション エクスプローラーから Dialogs¥RootDialog.cs ファイルを開きます。



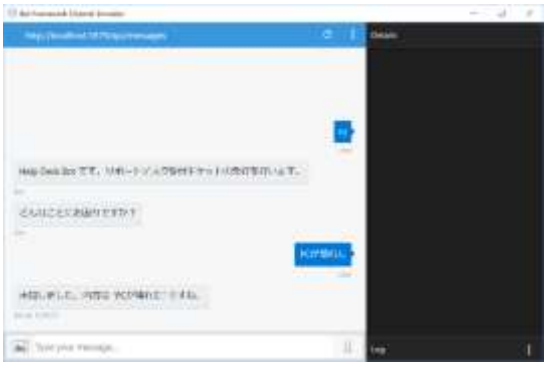
2. 以下の変数を、RootDialog クラスの先頭に追加します。この変数は、のちほどユーザーの回答を保存するために使用します。

```
private string category;
private string severity;
private string description;
```

3. MessageReceivedAsync メソッドを次のコードで置き換えます。

```
public async Task MessageReceivedAsync
        (IDialogContext context, IAwaitable<IMessageActivity> argument)
{
        var message = await argument;
        await context.PostAsync
                ("Help Desk Bot です。サポートデスク受付チケットの発行を行います。");
        PromptDialog.Text(context, this.DescriptionMessageReceivedAsync,
                "どんなことにお困りですか？");
}

public async Task DescriptionMessageReceivedAsync
        (IDialogContext context, IAwaitable<string> argument)
{
        this.description = await argument;
        await context.PostAsync
                ($"承知しました。内容は ¥"{this.description}¥" ですね。");
        context.Done<object>(null);
}
```

4. Visual Studio でソリューションを実行し ([実行] ボタンをクリック)、エミュレーターを開きます。演習 1 と同様にボットの URL を入力し (http://localhost:3979/api/messages)、ボットをテストします。URL 入力欄の横にある [Start New Conversation] をクリックすると、会話をリセットできます。



タスク 2: チケット詳細のプロンプト

このタスクでは、さらに多くのメッセージ ハンドラーをボット コードに追加して、サポートチケットを発行するための詳細について尋ねます。

5. アプリの Visual Studio での実行を停止し、Dialogs¥RootDialog.cs ファイルを開きます。

6. DescriptionMessageReceivedAsync を更新して、ユーザーが入力した説明を保存し、チケットの重要度について尋ねるコードを以下のように追加します。今回は、ユーザーに選択肢を示す PromptDialog.Choice メソッドを使用します。

```
public async Task DescriptionMessageReceivedAsync
        (IDialogContext context, IAwaitable<string> argument)
{
        this.description = await argument;
        var severities = new string[] { "high", "normal", "low" };
        PromptDialog.Choice(context, this.SeverityMessageReceivedAsync,
        severities, "この問題の重要度を入力してください");
}
```

7. 次に、SeverityMessageReceivedAsync メソッドを追加します。このメソッドは重要度を受信し PromptDialog.Text メソッドを使用して、ユーザーにカテゴリーの入力を求めます。

```
public async Task SeverityMessageReceivedAsync
        (IDialogContext context, IAwaitable<string> argument)
{
        this.severity = await argument;
        PromptDialog.Text(context, this.CategoryMessageReceivedAsync,
                "この問題のカテゴリーを以下から選んで入力してください ¥n¥n" +
                "software, hardware, networking, security, other");
}
```

8. 今度は、CategoryMessageReceivedAsync メソッドを追加します。このメソッドは、カテゴリーを保存し、PromptDialog.Confirm メソッドを使用して、ユーザーにチケットの作成の確認を求めます。

```
public async Task CategoryMessageReceivedAsync
      (IDialogContext context, IAwaitable<string> argument)
{
      this.category = await argument;
      var text = "承知しました。¥n¥n" +
            $"重要度: ¥"{this.severity}¥"、カテゴリー: ¥"{this.category}¥" "+
            "でサポートチケットを発行します。¥n¥n"+
            $"詳細: ¥"{this.description}¥" ¥n¥n"+
            "以上の内容で宜しいでしょうか？";

      PromptDialog.Confirm(context,
            this.IssueConfirmedMessageReceivedAsync, text);
}
```

Markdown 構文を使用すると、よりリッチなテキスト メッセージを作成できます。ただし、すべてのチャネルで Markdown がサポートされるわけではありません。

9. IssueConfirmedMessageReceivedAsync　メソッドを追加して、チケット作成確認メッセージの応答を処理します。

```
public async Task IssueConfirmedMessageReceivedAsync
      (IDialogContext context, IAwaitable<bool> argument)
{
      var confirmed = await argument;

      if (confirmed)
      {
            await context.PostAsync("サポートチケットを発行しました。");
      }
      else
      {
            await context.PostAsync("サポートチケットの発行を中止しました。"+
                  "最初からやり直してください。");
      }

      context.Done<object>(null);
}
```

10. アプリを再実行して、エミュレーターの [Start new conversation] ボタン を使用して接続し、新しい会話をテストします。

再度ボットに話しかけると、ダイアログが最初から開始されます。

タスク 3: 外部 API を呼び出してチケットを保存

これで、チケットのすべての情報が取得されましたが、この情報はダイアログが終了すると破棄されます。今度は、外部 API を使用して、チケットを作成するコードを追加します。わかりやすくするため、チケットをインメモリ アレイに保存する単純なエンドポイントを使用します。実稼働環境では、ボットのコードからアクセスできる外部 API を使用することになります。

11. アプリの Visual Studio での実行を停止します。Controllers フォルダーに、assets フォルダー の TicketsController.cs をコピーします。これは、/api/tickets エンドポイントへの POST 要求を処理し、チケットをアレイに追加して、作成されたチケット ID を使用して応答します。

12. 新しい Util フォルダーをプロジェクトに追加します。Util フォルダーに、assets フォルダー の TicketAPIClient.cs ファイルをコピーします。これはボットからチケット API を呼び出します。

13. Web.Config ファイルを開き、appSettings セクションで TicketsAPIBaseUrl キーを追加して、更新します。このキーには、チケット API を実行するベース URL が含

まれます。この演習では、ボットを実行する URL と同じになりますが、実稼働環境のシナリオでは別の URL になることがあります。

```
<add key="TicketsAPIBaseUrl" value="http://localhost:3979/" />
```

14. Dialogs¥RootDialog.cs ファイルを開きます。冒頭部分に HelpDeskBot.Util への参照を追加します。

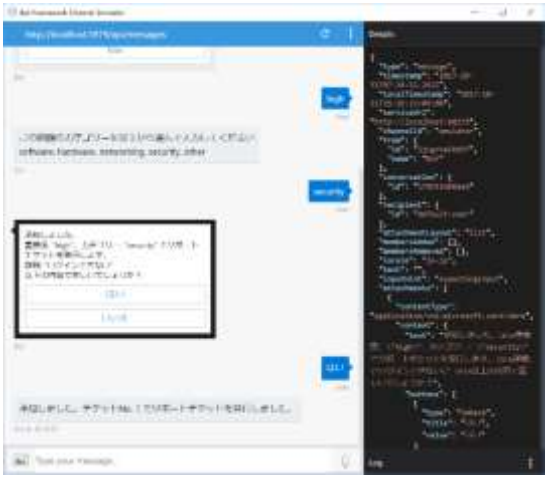
```
using HelpDeskBot.Util;
```

15. IssueConfirmedMessageReceivedAsync メソッドのコンテンツを以下のように置き換えて、TicketAPIClient を使用して呼び出しを行います。

```
public async Task IssueConfirmedMessageReceivedAsync
    (IDialogContext context, IAwaitable<bool> argument)
{
    var confirmed = await argument;

    if (confirmed)
    {
        var api = new TicketAPIClient();
        var ticketId = await api.PostTicketAsync
            (this.category, this.severity, this.description);

        if (ticketId != -1)
        {
            await context.PostAsync($"承知しました。チケット No. {ticketId} " +
                "でサポートチケットを発行しました。");
        }
        else
        {
            await context.PostAsync("サポートチケットの発行中に" +
                "エラーが発生しました。" +
                "恐れ入りますが、後ほど再度お試しください");
        }
    }
    else
    {
        await context.PostAsync("サポートチケットの発行を中止しました。" +
```

```
                "サポートチケット発行が必要な場合は再度やり直してください。");
        }

        context.Done<object>(null);
}
```

16. アプリを再実行して、エミュレーターの [Start new conversation] ボタン をクリックします。すべての会話を再度テストして、API からチケット ID が返されることを確認します。

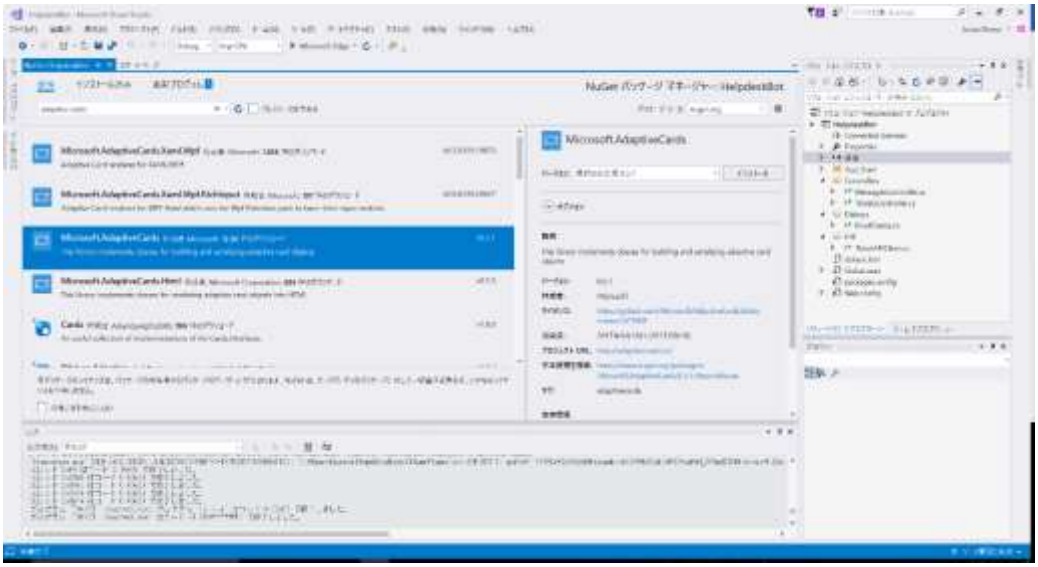


タスク 4: Adaptive Card による通知メッセージ

このタスクでは、チケットに Adaptive Card を適用し、ユーザーに表示される確認メッセージを向上させます。Adaptive Card とは、開発者が一定の共通方式で UI コンテンツをやり取りできるようにするための、オープン ソースのカード交換フォーマットです。Adaptive Card のコンテンツは、JSON オブジェクトとして指定できます。コンテンツはホスト アプリケーション (Bot Framework チャネル) 内でネイティブにレンダリングでき、ホストの外観に自動的に適合します。

17. Microsoft.AdaptiveCards NuGet パッケージを追加する必要があります。[ソリューション エクスプローラー] でプロジェクトの [参照] フォルダーを右クリックして、[NuGet パッケージの管理] をクリックします。[参照]タブの検索欄に「Adaptive

Cards」と入力して [Microsoft.AdaptiveCards を探し、[インストール] ボタンをクリックします。または、[パッケージ マネージャー コンソール] で Install-Package Microsoft.AdaptiveCards と入力します。



18. Dialogs¥RootDialog.cs ファイルを開きます。冒頭部分に System.Collections.Generic と AdaptiveCards への参照を追加します。

```
using System.Collections.Generic;
using AdaptiveCards;
```

19. ファイルの末尾 (RootDialog クラス内) に、Adaptive Card を作成する以下のコードを追加します。

このコードでは以下の内容をカードの内容として構成します

- ヘッダー セクションには ticketID を含むタイトルが入ります。
- 中央のセクションには、ColumnSet と 2 つの列が含まれます。1 列は重要度とカテゴリーを含む FactSet、もう 1 列はアイコンが含まれます。
- 最後のセクションには、チケットについて説明する説明ブロックが含まれます。

```
private AdaptiveCard CreateCard(int ticketId, string category, string severity, string description)
```

```
{
    AdaptiveCard card = new AdaptiveCard();

    var headerBlock = new TextBlock()
    {
        Text = $"Ticket #{ticketId}",
        Weight = TextWeight.Bolder,
        Size = TextSize.Large,
        Speak = $"承知しました。チケット No. {ticketId} でサポートチケットを" +
                "発行しました。担当者からの連絡をお待ちください。"
    };

    var columnsBlock = new ColumnSet()
    {
        Separation = SeparationStyle.Strong,
        Columns = new List<Column>
        {
            new Column
            {
                Size = "1",
                Items = new List<CardElement>
                {
                    new FactSet
                    {
                        Facts = new List<AdaptiveCards.Fact>
                        {
                            new AdaptiveCards.Fact("Severity:", severity),
                            new AdaptiveCards.Fact("Category:", category),
                        }
                    }
                }
            },
            new Column
            {
                Size = "auto",
                Items = new List<CardElement>
                {
                    new Image
                    {
                        Url =
                        "https://raw.githubusercontent.com/GeekTrainer/
                        help-desk-bot-lab/master/assets/botimages/head-
                        smiling-medium.png",
                        Size = ImageSize.Small,
```

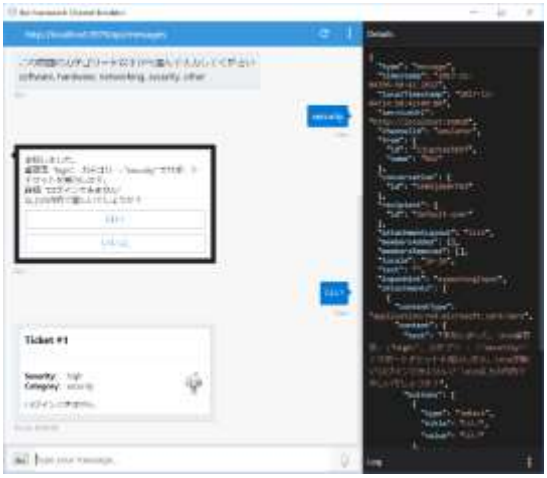
```
                                HorizontalAlignment = HorizontalAlignment.Right
                            }
                        }
                    }
                }
            };

            var descriptionBlock = new TextBlock
            {
                Text = description,
                Wrap = true
            };

            card.Body.Add(headerBlock);
            card.Body.Add(columnsBlock);
            card.Body.Add(descriptionBlock);

            return card;
        }
```

20. IssueConfirmedMessageReceivedAsync メソッドを以下のように更新して、チケットが作成されたら、CreateCard メソッドを呼び出します。

```
public async Task IssueConfirmedMessageReceivedAsync
        (IDialogContext context, IAwaitable<bool> argument)
{
        var confirmed = await argument;

        if (confirmed)
        {
                var api = new TicketAPIClient();
                var ticketId = await api.PostTicketAsync(this.category,
                                        this.severity, this.description);

                if (ticketId != -1)
                {
                        var message = context.MakeMessage();
                        message.Attachments = new List<Attachment>
                        {
                                new Attachment
                                {
                                        ContentType
                                                = "application/vnd.microsoft.card.adaptive",
```

```
                        Content = CreateCard(ticketId, this.category,
                            this.severity, this.description)
                    }
                };
                await context.PostAsync(message);
            }
            else
            {
                await context.PostAsync("サポートチケットの発行中に" +
                    "エラーが発生しました。" +
                    "恐れ入りますが、後ほど再度お試しください");
            }
        }
        else
        {
            await context.PostAsync("サポートチケットの発行を中止しました。" +
                "サポートチケット発行が必要な場合は再度やり直してください。");

        }
        context.Done<object>(null);
}
```

21. アプリを再実行して、エミュレーターの [Start new conversation] ボタン を使用します。新しい会話をテストし、Adaptive Card 形式でチケットが発行されるのを確認します。



演習 3: 言語理解の機能によるボットのスマート化

人間が何を欲しているかをコンピューターが理解することは、人間とコンピューターとの対話式操作における大きな問題の 1 つになっています。Cognitive Services LUIS (Language Understanding Intelligent Service) は、人間の言語を理解することにより、ユーザーの要求に応じることができるスマート アプリケーションを開発者が構築できるように設計されています。

この演習では、ヘルプ デスク ボットに自然言語理解機能を追加して、ユーザーがチケットを容易に作成できるようにする方法を学習します。そのためには、LUIS を使用します。これはボットがコマンドを理解して行動できるようにさせるための言語モデルを、開発者が構築できるようにします。たとえば、前の演習ではユーザーが重大度とカテゴリーを入力する必要がありました。今回は、ユーザーのメッセージから両方の "エンティティ" が認識されるようにします。

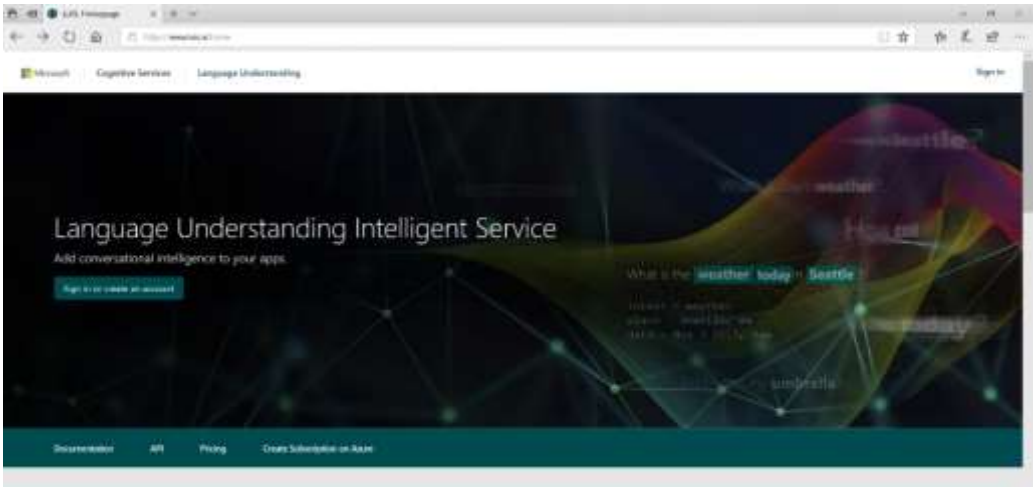
このフォルダーの中には、Visual Studio ソリューションと、この演習のステップで作成するコードが入っています。

タスク 1: LUIS アプリを作成する

このタスクでは、LUIS ポータルでアプリを作成します。

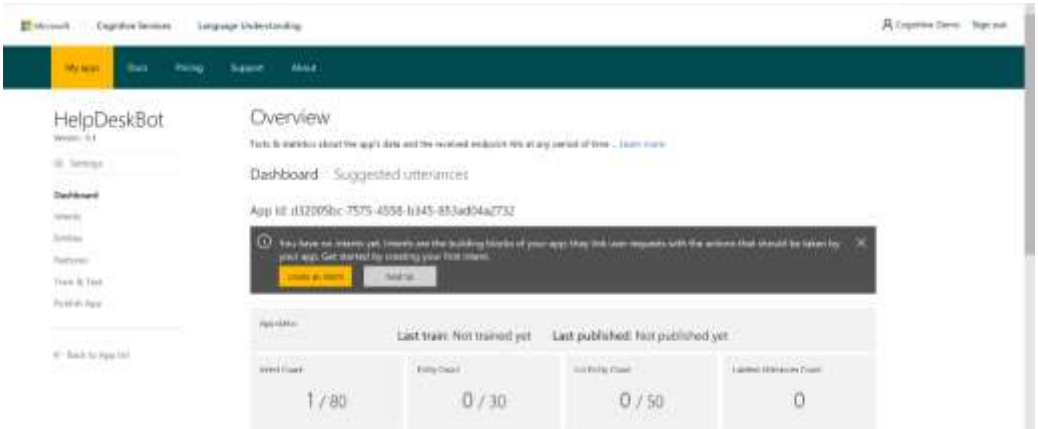
> assets フォルダー にある TSFAQBot.json を各自のアカウントにインポートして、モデルをトレーニングおよび発行し、タスク 4 に進むことができます。初めて LUIS に触れる方は、下記の演習を通して理解を深めることをおすすめします。

1. LUIS ポータル (https://www.luis.ai) をブラウザーで開き、[Sign in or create an account] をクリックし、Microsoft アカウントでサインインし、初めての場合は初期設定を行います。



2. [My apps] タブが表示されたら、[New App] をクリックします。ダイアログ ボックスにアプリケーションの名前 (たとえば、「HelpDeskBot」) を入力します。[Culture] で [Japanese] を選択します。[Create] をクリックして LUIS アプリを作成します。

3. 空の LUIS アプリ ダッシュボードが表示されます。後で使用できるように App ID を保存します。

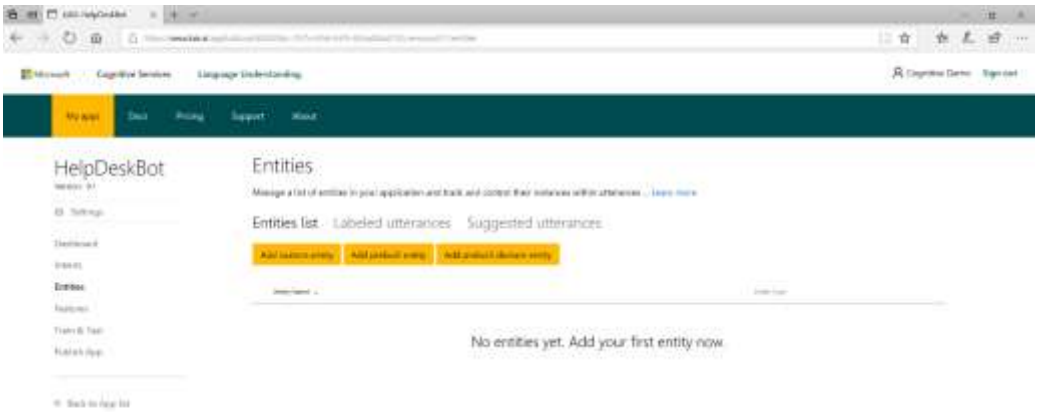


タスク 2: LUIS に新しいエンティティを追加する

このタスクでは、LUIS アプリにエンティティ (Entity) を追加します。これによって、ボットはチケットのカテゴリーと重大度を、ユーザーが入力した問題の説明から理解できるようになります。エンティティは、類似のオブジェクト (場所、もの、人間、イベントまたは概念) の集合を含むクラスを表します。

4. LUIS ポータルの左パネルで [Entities] をクリックします。

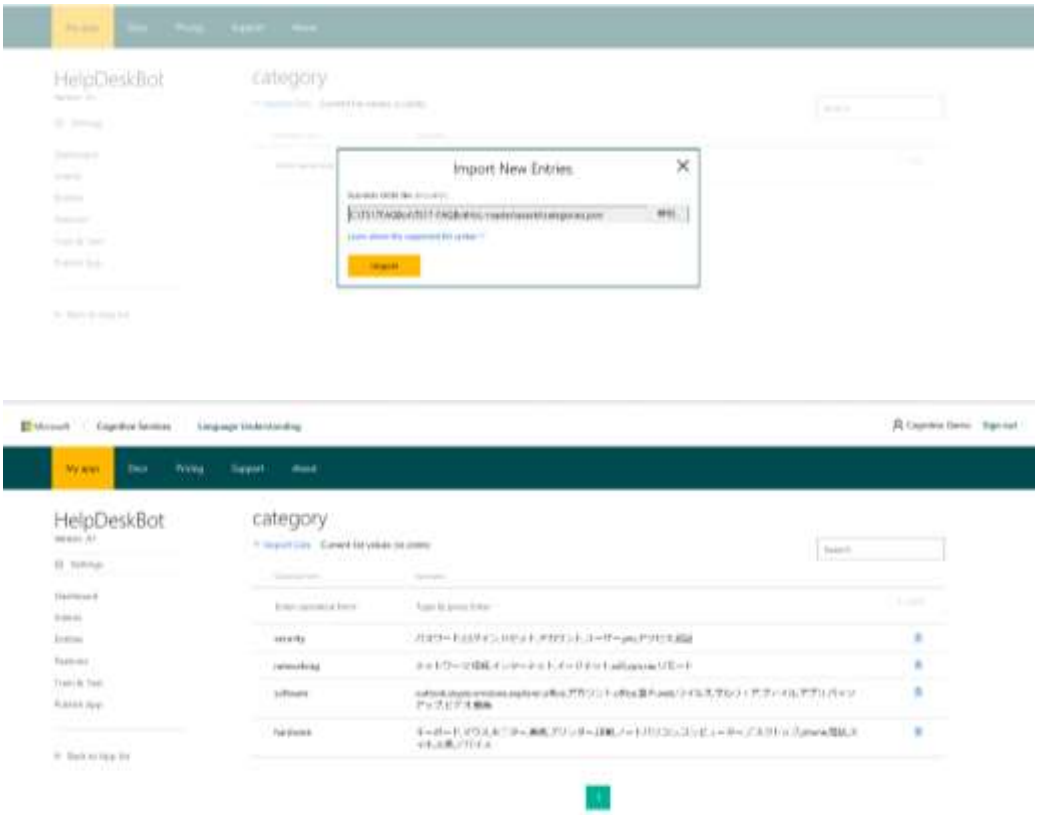
5. [Add custom entity] をクリックします。



6. 表示されたダイアログで [Entity name] に「category」と入力します。[Entity type] には "List" を選択します。[Save] をクリックします。

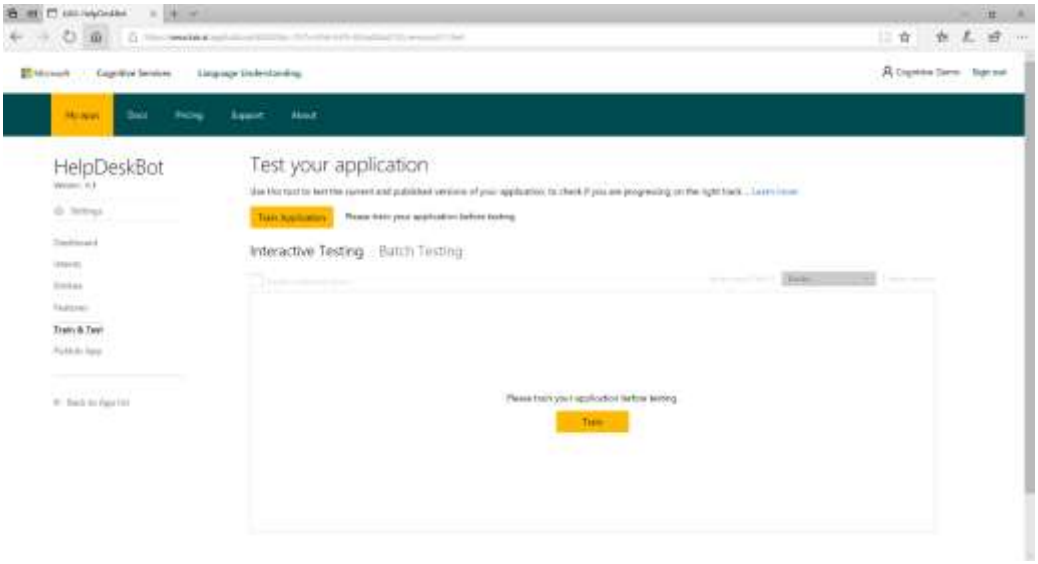
> このラボでは、Entity Type の List を使用します。これにより、一般に "クローズド リスト" と呼ばれるものを作成できます。これは、用語に機械学習を適用せず、直接一致を使用することを意味します。用語の正規化の試行時、または一定のキーワードを常にエンティティとして取得する場合に非常に有益です。

7. 新しいページが表示され、そのページで使用可能な値を追加できます。今回はファイルからインポートします。assets フォルダー にある categories.json ファイルをダウンロードします。[+ Imports Lists] リンクをクリックして categories.json を指定、[Import] をクリックします。



8. 左パネルの [Entities] をクリックし、"severity" という名前の新しいエンティティを同様に作成します。assets フォルダー にある "severities.json" という名前のファイルを使用して読み込みます。

9. 次に、左パネルの [Train & Test] をクリックします。



10. [Train Application] をクリックして、完了するまで待ちます。

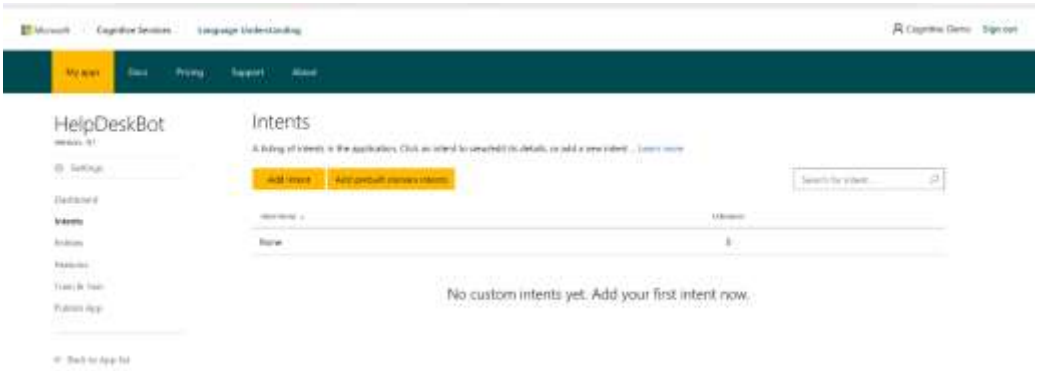
> LUIS アプリのモデルを更新する場合、必ずトレーニング、テスト/発行を行う必要があります。

タスク 3: インテントおよび発話を追加する

インテント (Intent) は発話 (文) を通じて伝達される意図または望まれるアクションです。インテントは、ボットにアクションを実行させることでユーザーの要求に応じます。このため、ボットがユーザーの要求を理解し、適切に対応できるようにインテントを追加する必要があります。

発話は、ボットに受信/解釈させるためのユーザー クエリやコマンドのサンプルを表す文です。アプリ内の各インテントにサンプルの発話を追加する必要があります。LUIS がこれらの発話から学習すると、アプリは同様のコンテキストの一般化および理解が可能になります。発話を継続的に追加して、ラベル付けすることで、ボットの言語学習エクスペリエンスが向上します。

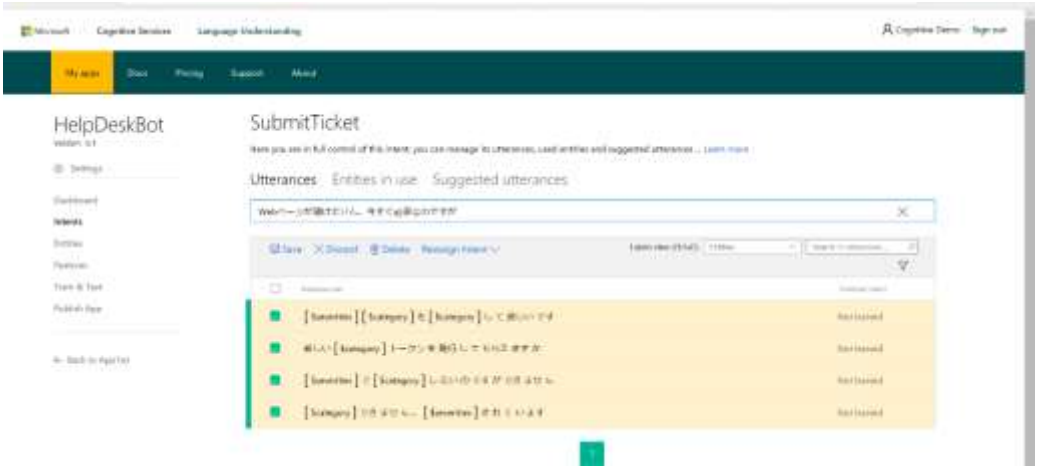
11. LUIS ポータルの左パネルで [Intents] をクリックします。既に "None" インテントがデフォルトで存在しています。



12. [Add Intent] をクリックすると、ポップアップが表示されます。[Intent name] に「SubmitTicket」と入力して、[Save] をクリックします。

13. 次は、テキスト ボックスに次の発話を追加します。1 つ入力するごとに Enter キーを押します。ユーザーがこれらの文または類似の文を入力すると、LUIS アプリはユーザーがチケットを送信しようとしていると想定します。Bot Framework 言語ではこれを "インテント (Intent) " と呼びます。

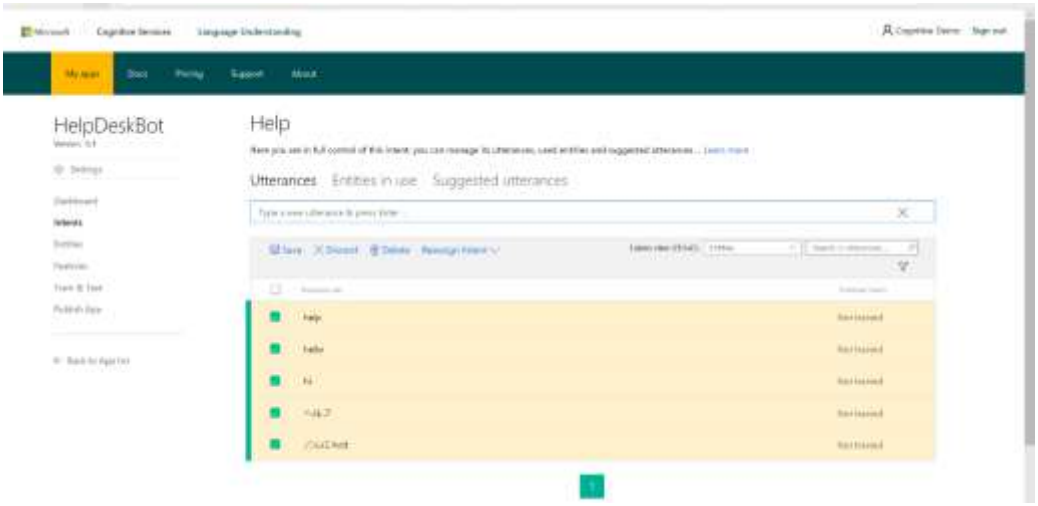
- ログインできません。ブロックされています
- 急ぎで印刷したいのですができません
- 新しい RAS トークンを発行してもらえますか
- 至急パスワードをリセットして欲しいです
- Web ページが開けません。今すぐ必要なのですが



発話は必要なだけ追加できます。追加する発話が多いほど、アプリがユーザーのインテントを認識する能力が高まります。今回の使用例では、多様な発話 (ハードウェアの問題もあれば、ソフトウェアの問題もあります) が SubmitTicket を起動する可能性があるため、ボットを実稼働に向けてリリースする前に大量の発話でボットをトレーニングすることが理想的です。

14. [Save] をクリックして保存します。

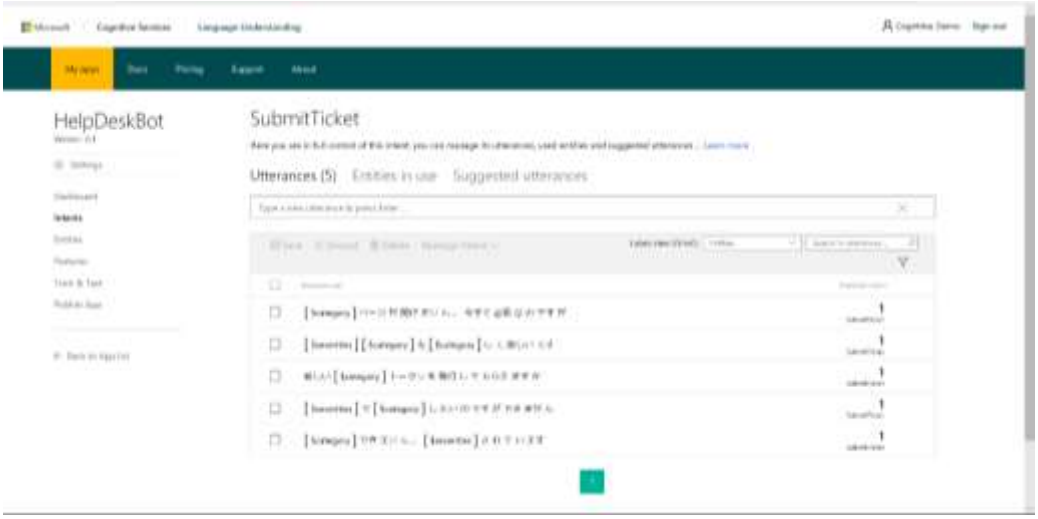
15. 前述の手順と同様に、新しい Help インテントを追加し、発話に "こんにちは"、"ヘルプ"、"hi"、"hello"、"help" を追加し、[Save]をクリックして保存します。



"None" インテントに発話をいくつか追加すると、他のインテントの起動精度の向上に役立ちます。

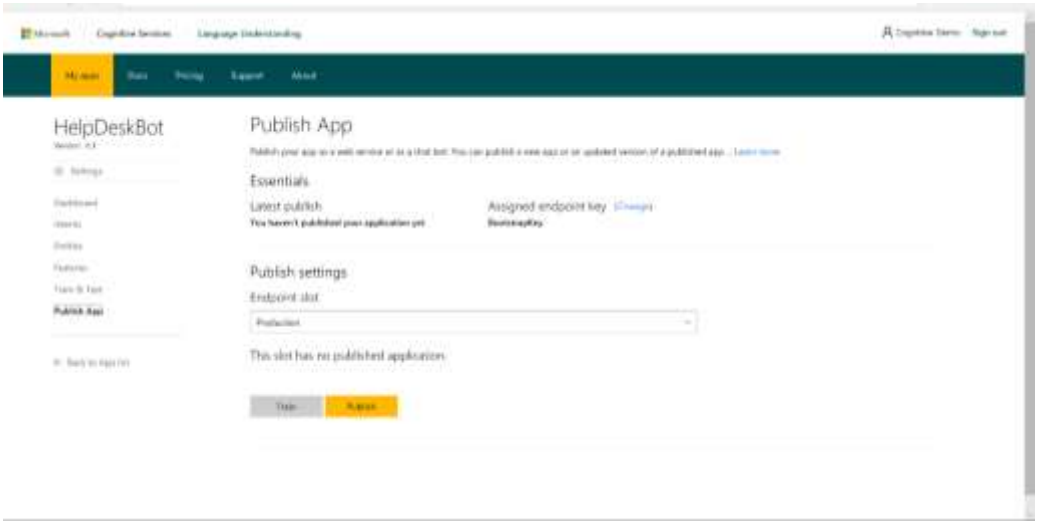
16. 左パネルの [Train & Test] をクリックして表示し、[Train Application] をクリックして、アプリを再度トレーニングします。

17. [Intents] メニューを開き、[SubmitTicket] インテントをクリックします。発話に含まれるエンティティが認識されていることを確認します。



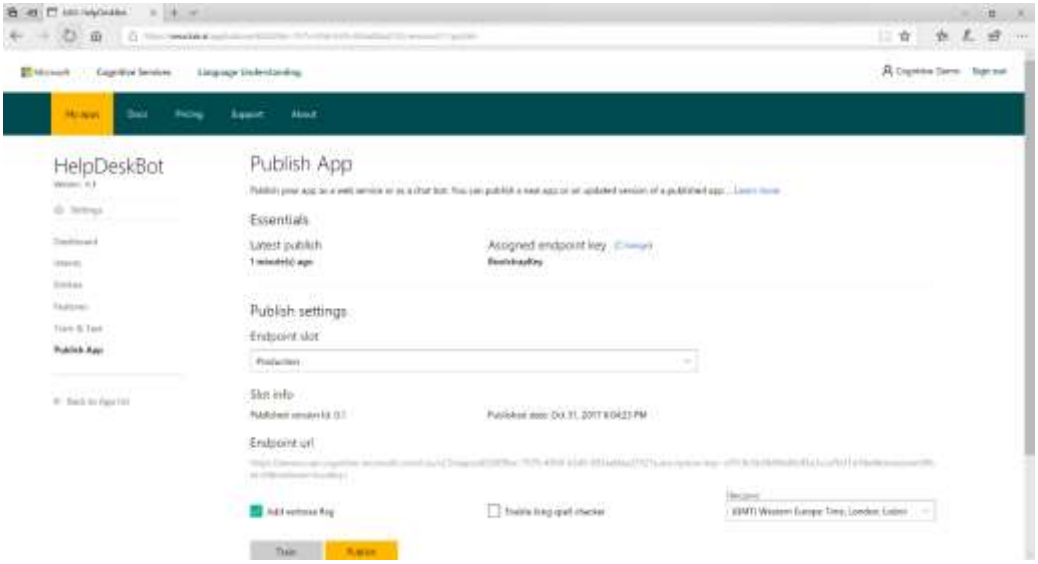
18. 次に、LUIS アプリを発行してボットから使用できるようにします。左側のメニューの [Publish App] をクリックします。

19. [Endpoint key] には BootstrapKey が選択されていることを確認します。既定の [Production] スロットはそのままにします。



BoostrapKey は 1 か月あたり 1000 トランザクションを上限として試用可能なキーです。

20. [Publish] をクリックします。LUIS アプリが発行され、Endpoint URL が表示されます。この Endpoint URL に含まれる “…?subscription-key=” と “&timezone…” の間に表示されている文字列 (=サブスクリプションキー) を保管しておきます。



> 表示される URL は LUIS アプリの HTTP エンドポイント (自然言語の理解を追加する際にボットから参照する) が設定された Web サービスです。この URL からモデルを構築した LUIS アプリを利用可能です。

タスク 4: LUIS を使用するようにボットを更新する

このタスクでは、ボット コードを更新して、前の手順で作成した LUIS アプリを使用するようにします。

21. 演習 2 で作成したソリューションを開きます。

22. Dialogs¥RootDialog.cs ファイルを開きます。冒頭に Microsoft.Bot.Builder.Luis と Microsoft.Bot.Builder.Luis.Models への参照を追加します。

```
using Microsoft.Bot.Builder.Luis;
using Microsoft.Bot.Builder.Luis.Models;
```

23. 次のようにして、LuisModel 属性をクラスに追加します (RootDialog クラスの直前に挿入します。) LUISAppID を LUIS ポータルから保存したアプリ ID に置き換え、LUISSubKey を Publish App ページから保存したサブスクリプションキーに置き換えます。

```
[LuisModel("LUISAppID", "LUISSubKey")]
```

24. RootDialog のインターフェイス IDialog の実装を、LuisDialog<object> から導出するように置き換えます。メソッド StartAsync、MessageReceivedAsync、および DescriptionMessageReceivedAsync は、呼び出されなくなるため、削除します。

```
//public class RootDialog : IDialog<object>
public class RootDialog : LuisDialog<object>
```

25. LUIS モデルがインテントを検出したときに実行される None メソッドを作成します。LuisIntent 属性を使用し、インテント名をパラメーターとして渡し、インテントによる対応を記述します。

```
[LuisIntent("")]
[LuisIntent("None")]
public async Task None(IDialogContext context, LuisResult result)
{
    await context.PostAsync("申し訳ありません。" +
        $"「{result.Query})」を理解できませんでした。¥n" +
        " 'ヘルプ' または 'help' と入力すると、ヘルプメニューを表示します。");
    context.Done<object>(null);
}
```

26. 次のコードを追加して、Help インテントに対応します。

```
[LuisIntent("Help")]
public async Task Help(IDialogContext context, LuisResult result)
{
    await context.PostAsync("私は Help Desk Bot です。¥n" +
        "「パスワードをリセットしたい」「印刷できない」といった文章を" +
        "入力してください。");
    context.Done<object>(null);
}
```

27. 次のコードを追加して、インテント SubmitTicket を処理するメソッドを追加します。TryFindEntity メソッドを使用して、発話にエンティティが存在するかどうかを判定して、それを抽出します。

```
[LuisIntent("SubmitTicket")]
public async Task SubmitTicket(IDialogContext context, LuisResult result)
{
    EntityRecommendation categoryEntityRecommendation, severityEntityRecommendation;

    result.TryFindEntity("category", out categoryEntityRecommendation);
    result.TryFindEntity("severity", out severityEntityRecommendation);

    this.category
    = ((Newtonsoft.Json.Linq.JArray)categoryEntityRecommendation?
        .Resolution["values"])?[0]?.ToString();
    this.severity
    =((Newtonsoft.Json.Linq.JArray)severityEntityRecommendation?
        .Resolution["values"])?[0]?.ToString();
    this.description = result.Query;

    await this.EnsureTicket(context);
}
```

28. 次に、EnsureTicket メソッドを作成します。これは、エンティティが識別されたかどうかを検証し、識別されていない場合はユーザーに不足のエンティティの入力を求めるものです。

```
private async Task EnsureTicket(IDialogContext context)
{
    if (this.severity == null)
    {
        var severities = new string[] { "high", "normal", "low" };
        PromptDialog.Choice(context, this.SeverityMessageReceivedAsync,
        severities, "この問題の重要度を選択してください。");
    }
    else if (this.category == null)
    {
        PromptDialog.Text(context, this.CategoryMessageReceivedAsync,
            "この問題は以下のどのカテゴリーになりますか？¥n¥n" +
            "software, hardware, networking, security, other  のいずれかを" +
```

```
                "入力してください。");
        }
        else
        {
            var text = "承知しました。¥n¥n" +
                $"重要度: {this.severity}、カテゴリー: {this.category}¥n¥n" +
                $"詳細: {this.description} ¥n¥n" +
                "以上の情報でチケットを発行します。よろしいでしょうか？";

            PromptDialog.Confirm(context,
                this.IssueConfirmedMessageReceivedAsync, text);
        }
}
```

29. 次のように、SeverityMessageReceivedAsync および CategoryMessageReceivedAsync を更新して、EnsureTicket メソッドをコール バックするようにします。

```
private async Task SeverityMessageReceivedAsync(IDialogContext context, IAwaitable<string> argument)
{
    this.severity = await argument;
    await this.EnsureTicket(context);
}
```

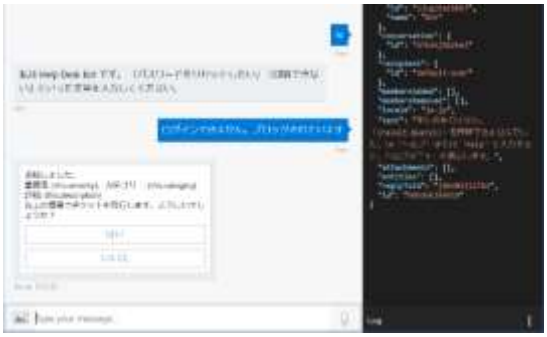
```
private async Task CategoryMessageReceivedAsync(IDialogContext context, IAwaitable<string> argument)
{
    this.category = await argument;
    await this.EnsureTicket(context);
}
```

30. Visual Studio からアプリを実行して、エミュレーターにボットの URL (http://localhost:3979/api/messages) を入力してアクセスします。

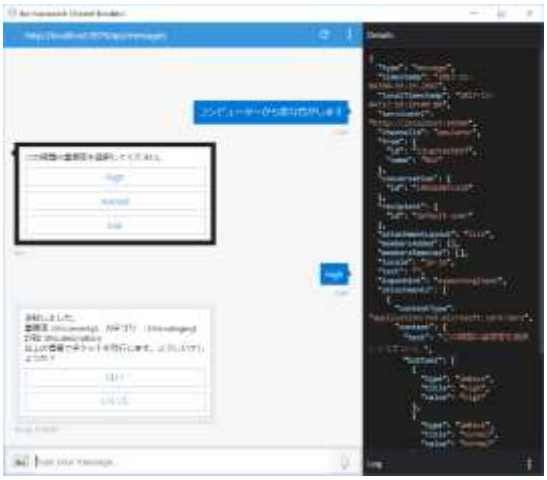
31. 「hi」と入力します。Help インテントが認識されるのを確認してください。

32. ボットのトレーニングに使用した発話のいずれかを入力します。たとえば「ログインできません。ブロックされています」と入力します。ユーザーのメッセージから、チケッ

トのカテゴリーおよび重大度が自動的に把握されます。「はい」と入力して、チケットを保存します。



33. 次に、ボットのトレーニングに使用されていない発話を入力してみます。(例:コンピューターから変な音がします) 重大度は把握されていませんが、エンティティ computer が存在するためカテゴリーは把握されます。



34. LUIS が認識できない発話を入力すると、LUIS は “None” インテントを返し、ボットは None タスクで指定したメッセージを返答します。

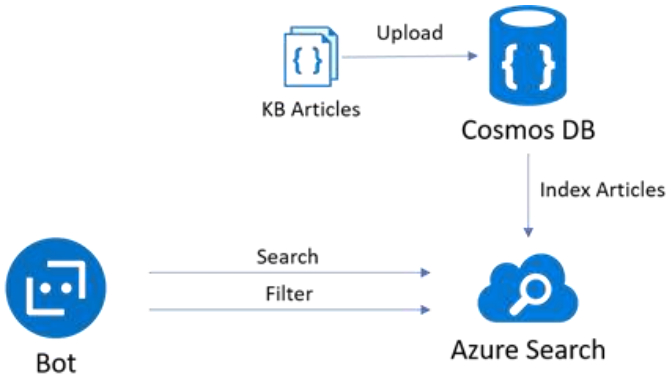
アプリケーションを展開してユーザーの入力を開始すると、LUIS による判定を改善する必要があります。LUIS ポータルのインテント内には [Suggested Utterances] セクションが存在し、LUIS があまり確信できない発話に対してインテントまたはエンティティを行います。もし、その判定が間違っている場合はそのセクション内で正しい “ラベル付け” を実行できます。

演習 4: Azure Search と Cosmos DB によるヘルプ デスク ナレッジ ベースの実装

ボットは、ユーザーが大量のコンテンツをナビゲートする支援を行い、ユーザーのためにデータ駆動型の検索エクスペリエンスを実現することもできます。この演習では、検索機能をボットに追加し、ユーザーがナレッジ ベースを検索する支援を行う方法について学習します。これを行うには、Azure Cosmos DB に保管されている ナレッジ ベース (KB) の記事のインデックスを作成する Azure Search サービスにボットを接続します。

Azure Cosmos DB は、マイクロソフトが提供する、ミッション クリティカルなアプリケーション向けのグローバル分散型マルチ モデル データベース サービスです。Azure Cosmos DB は、さまざまなデータ モデルをサポートします。この演習では、Azure Cosmos DB の DocumentDB API を使用します。これを使用することで、KB の記事を JSON ドキュメントとして保管できます。

Azure Search は、カスタム アプリケーションで充実した検索エクスペリエンスを実現する完全管理型のクラウド検索サービスです。Azure Search は、さまざまなソース (Azure SQL DB、Cosmos DB、BLOB ストレージ、テーブル ストレージ) のコンテンツのインデックスを作成でき、その他のデータ ソースに対応する「プッシュ型」のインデックス作成をサポートします。また、PDF、Office ドキュメント、および非構造化データが含まれるその他の形式のドキュメントを開くことができます。コンテンツ カタログが Azure Search インデックスに取り込まれることで、ボットのダイアログからクエリを行えるようになります。

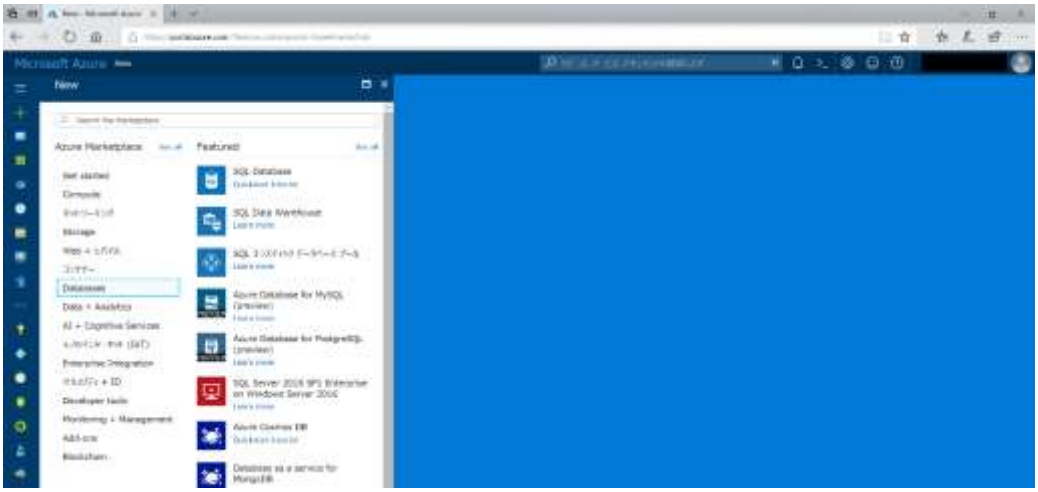


このフォルダーの中には、Visual Studio ソリューションと、この演習のステップで作成するコードが入っています。

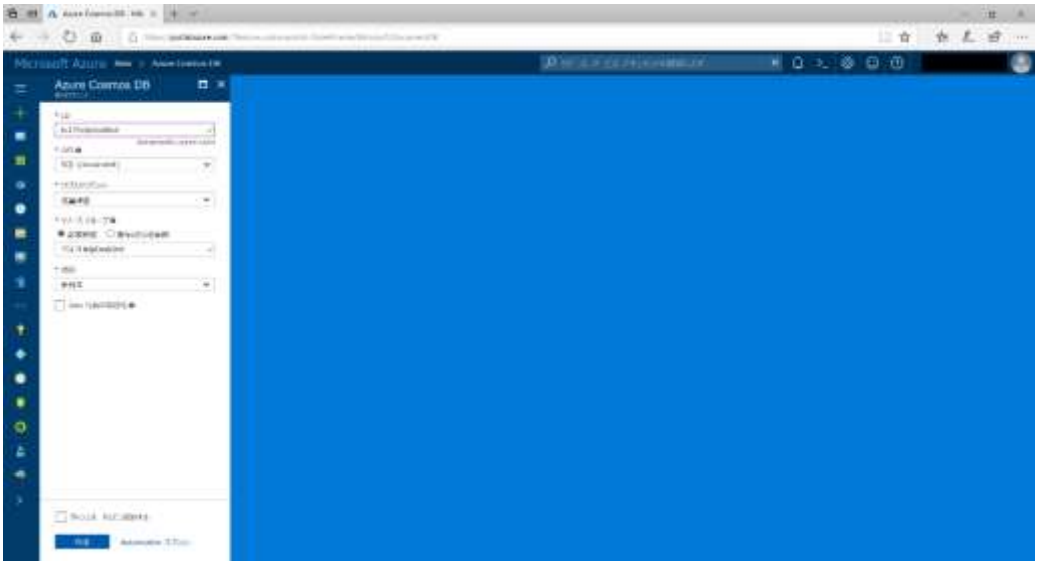
タスク 1: Cosmos DB サービスを作成し、ナレッジ ベースをアップロードする

このタスクでは、Cosmos DB データベースを作成し、ボットによって使用されるいくつかのドキュメントをアップロードします。

1. Azure ポータル (https://portal.azure.com/) にアクセスしてサインインします。左側のバーにある [新規](+) をクリックし、次に [データベース] をクリックして、Azure Cosmos DB を選択します。

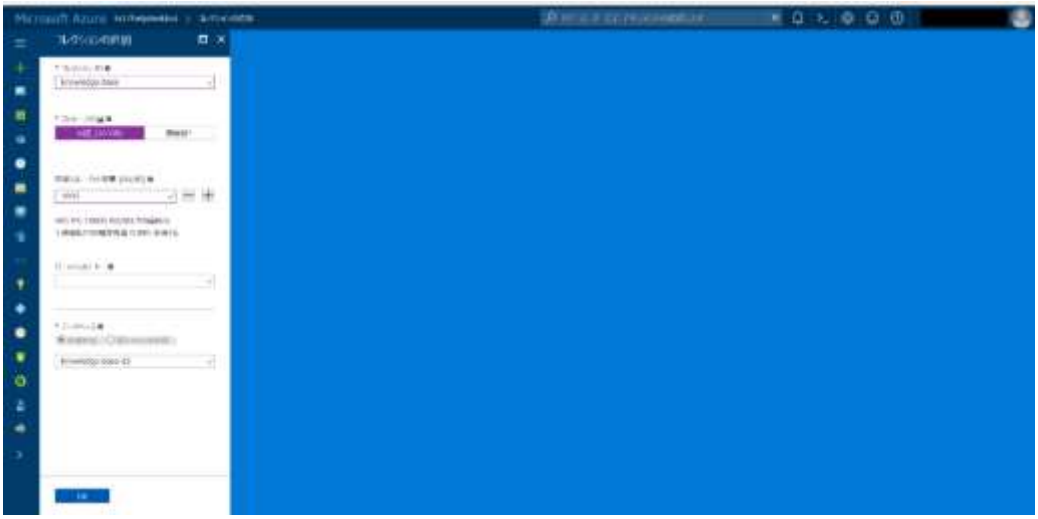


2. ダイアログ ボックスで一意のアカウント ID (例: help-desk-bot) を入力し、[API] で SQL (DocumentDB) を選択します。新しいリソース グループ名を入力し、[作成] をクリックします。展開が完了するまで待ちます。

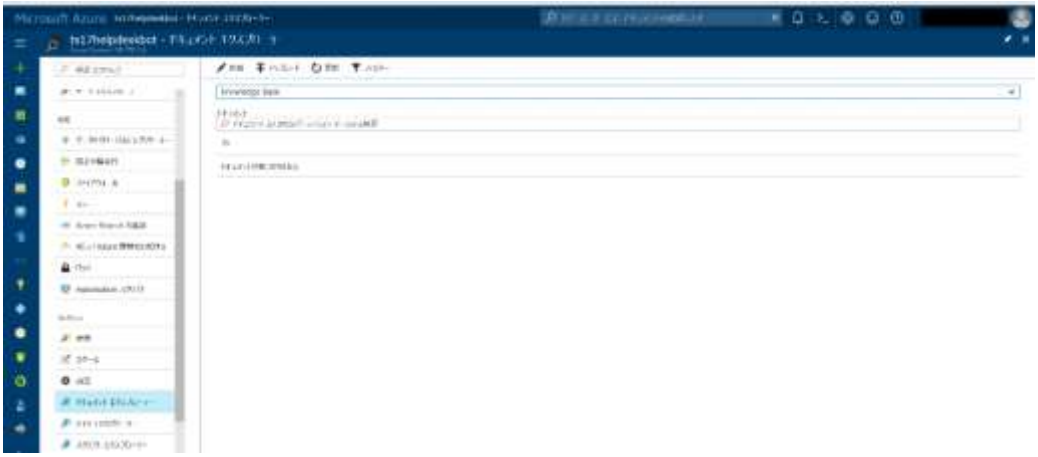


3. 作成した Cosmos DB アカウントを開き、[概要] セクションに移動します。[コレクションの追加] ボタンをクリックします。

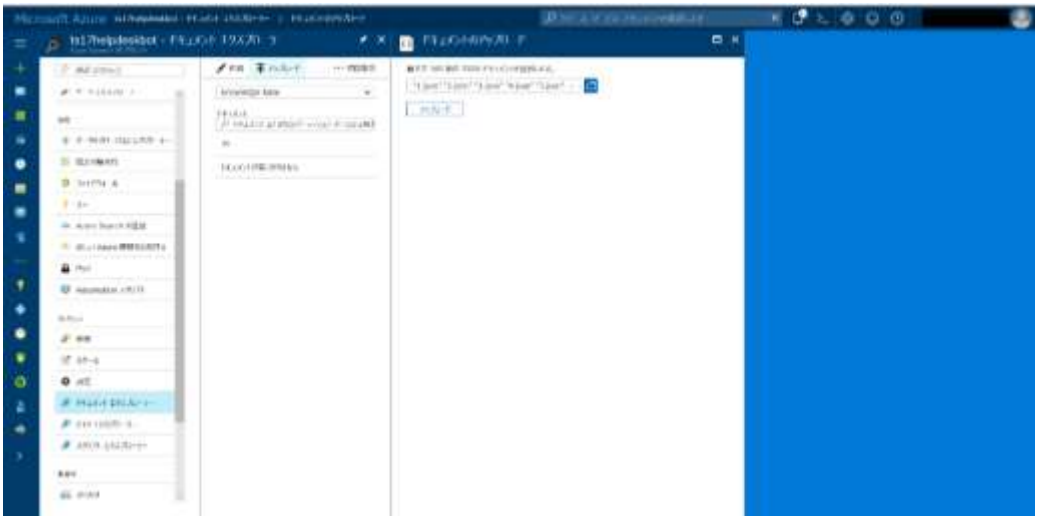
4. ダイアログ ボックスの [コレクション ID] で「knowledge-base」と入力し、[ストレージ容量] で [固定] を選択します。[データベース] は新規作成を選択し、名前として「knowledge-base-db」と入力します。[OK] をクリックします。



5. 左側のパネルで [ドキュメント エクスプローラー] を選択し、次に [アップロード] ボタンをクリックします。



6. ファイルの選択 で、assets/kb フォルダー のファイルをすべて選択します。各ファイルは、ナレッジ ベースの 1 つの記事に相当します。[アップロード] をクリックします。アップロードが完了するまでブラウザーを閉じないでください。

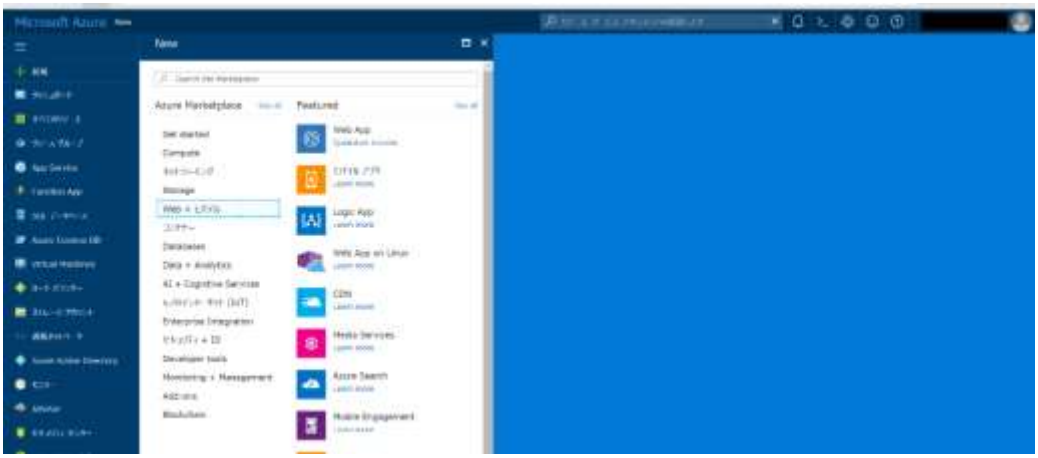


> 記事の「ドキュメント」にはそれぞれ 3 つのフィールド (タイトル、カテゴリー、およびテキスト) が含まれています。

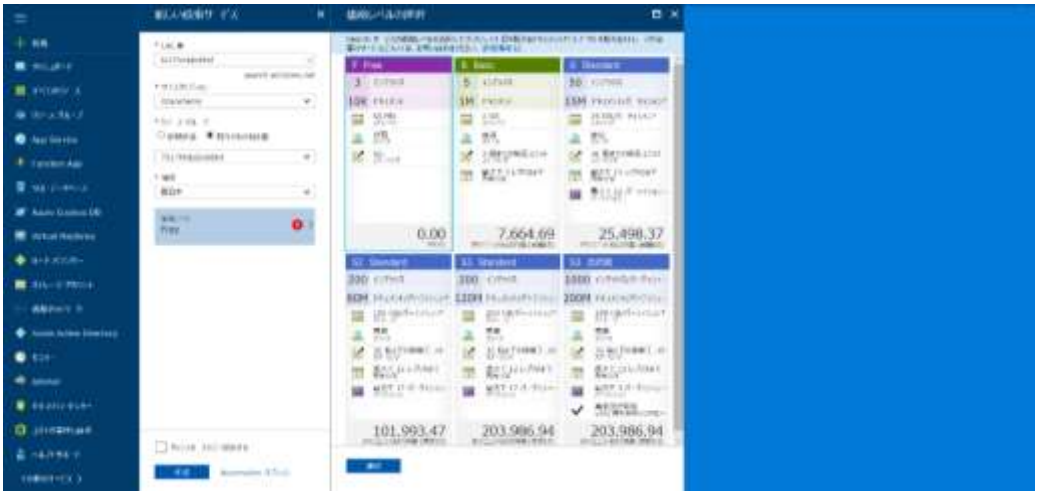
タスク 2: Azure Search サービスを作成する

このタスクでは、Cosmos DB にアップロードされたコンテンツのインデックスを作成するための Azure Search サービスを作成します。

7. Azure ポータルの左側のバーにある [新規] (+) をクリックし、次に [Web + モバイル] をクリックして、Azure Search を選択し、[作成] ボタンをクリックします。

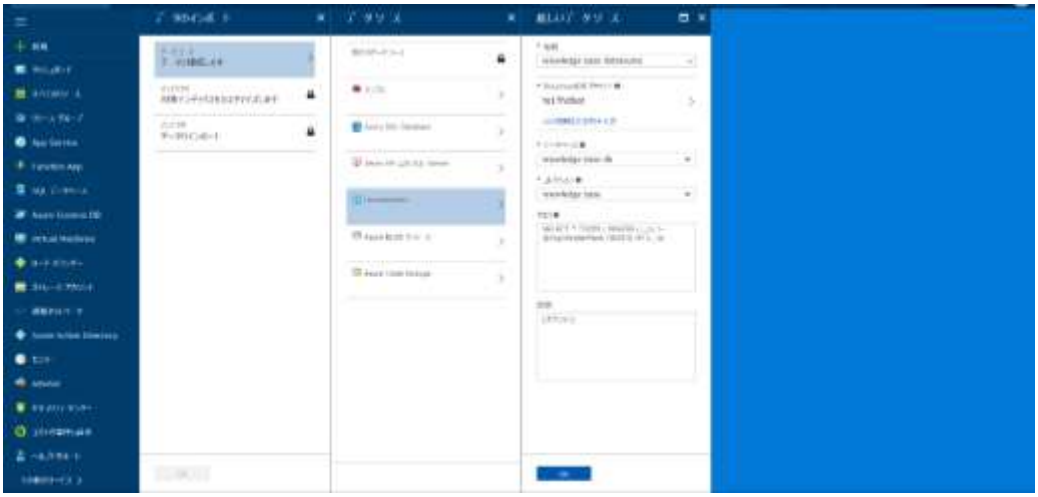


8. 一意の URL (例: help-desk-bot-search) を入力し、後で使用するために保存します。Cosmos DB で使用したものと同じリソース グループを選択します。[Price Tier] を [無料] に変更し、[作成] をクリックします。

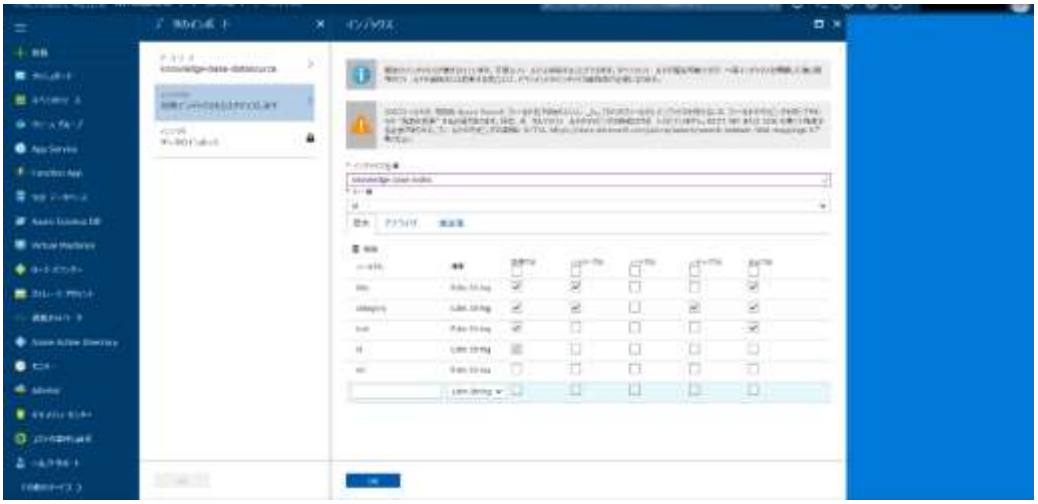


9. サービスのプロビジョニング後、[概要] に移動してから [データのインポート] ボタン をクリックします。

10. [データに接続します] ボタン、[DocumentDB] の順にクリックします。データ ソース名として「knowledge-base-datasource」と入力します。先ほど作成した Cosmos DB のアカウント、データベース、およびコレクションを選択します。[OK] をクリックします。



11. [Index – 対象インデックスをカスタマイズします] ボタンをクリックします。[インデックス名] で「knowledge-base-index」と入力します。インデックスの定義が以下の図と一致するように各列のチェックボックスを更新します。[OK] をクリックします。



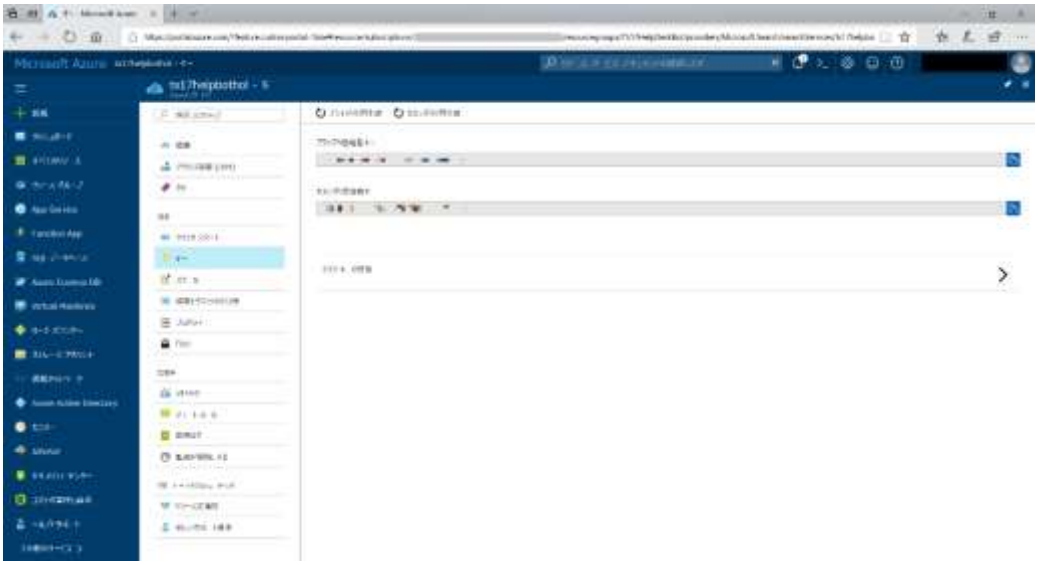
カテゴリー フィールドの [フィルター可能] と [ファセット可能] にチェックマークが付いていることを確認します。これにより、カテゴリーが一致するすべての記事を取得できると共に、各カテゴリーの記事の数も取得できるようになります。これは、Azure Search の専門用語で「ファセット ナビゲーション」と呼ばれます。

インデックスの詳細については、こちらの記事を参照してください。
https://docs.microsoft.com/ja-jp/azure/search/search-what-is-an-index

12. 最後に、[インデクサー -データのインポート] をクリックします。[名前] で「knowledge-base-indexer」と入力します。[スケジュール] で [1 度] が選択されていることを確認します。[OK] をクリックします。

13. 再び [OK] をクリックし、[データのインポート] ダイアログを閉じます。

14. 左側で [キー] をクリックし、次に [クエリ キーの管理] をクリックします。次のタスクで使用するために、既定の Azure Search キー (<空> という名前で示されています) を保存します。



> クエリ キーは、Search インデックスの読み取り専用の操作 (例: ID によるドキュメントのクエリと検索) でしか使用できません。管理キー (プライマリ管理キーとセカンダリ管理キー) は、すべての操作 (例: サービスの管理、インデックス、インデクサー、データ ソースの作成と削除) を行える完全な権限を付与します。

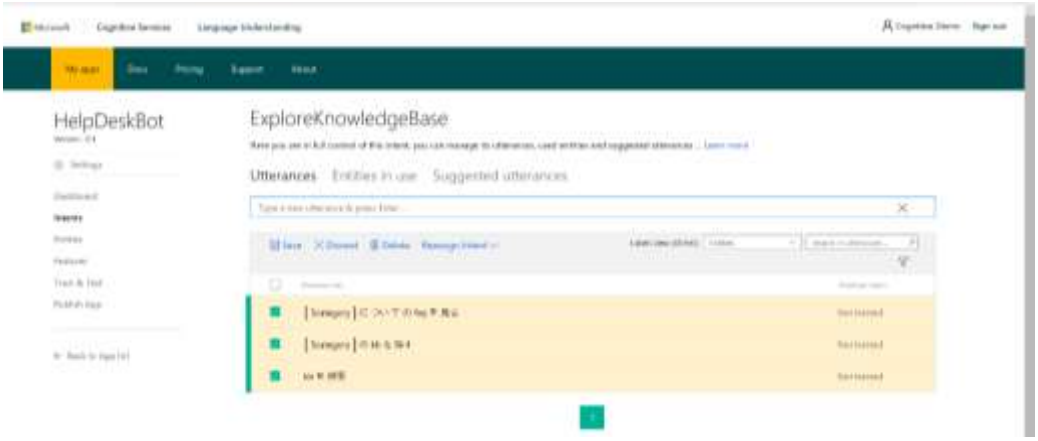
タスク 3: ExploreKnowledgeBase インテントが含まれるように LUIS モデルを更新する

このタスクでは、ナレッジ ベースを検索するために LUIS に新しいインテントを追加します。

15. LUIS ポータル にサインインします。演習 3 で作成した LUIS アプリを編集します。

16. 左側のメニューで [Intents] をクリックし、次に [Add Intent] ボタンをクリックします。インテント名として「ExploreKnowledgeBase」と入力してから、以下の発話を追加します。

- 「KB を検索」
- 「hardware の KB を探す」
- 「hardware についての FAQ を見る」



17. [Save] をクリックします。

18. 左側にある [Publish App] リンクをクリックします。[Train] ボタンをクリックし、完了したら [Publish] ボタンをクリックします。

タスク 4: Azure Search API を呼び出せるようにボットを更新する

このタスクでは、先ほど作成したインテントに対応し、Azure Search サービスを呼び出すためのダイアログを追加します。

19. 前の演習で作成したソリューションを開きます。

20. 前のタスクで作成した Azure Search サービスを使用できるように、Web.config に含まれる appSettings セクションの以下のキーを追加します。AzureSearchAccount と AzureSearchKey は タスク 2 で作成した値を指定してください。

```
...
<add key="AzureSearchAccount" value="YourAzureSearchAccount" />
<add key="AzureSearchIndex" value="knowledge-base-index" />
<add key="AzureSearchKey" value="YourAzureSearchKey" />
...
```

21. プロジェクトに Model フォルダーを追加します。assets フォルダー から Model フォルダーに SearchResult.cs と SearchResultHit.cs をコピーします。これらのクラスは、Azure からの記事の検索に対応します。

22. プロジェクトで Services フォルダーを作成し、新たに AzureSearchService.cs という名前でクラスファイルを追加します。

23. AzureSearchService.cs を下記のコードで書き換えます。

```
namespace HelpDeskBot.Services
{
    using Model;
    using Newtonsoft.Json;
    using System;
    using System.Net.Http;
    using System.Threading.Tasks;
    using System.Web.Configuration;

    [Serializable]
    public class AzureSearchService
    {
        private readonly string QueryString =
            $"https://{WebConfigurationManager.AppSettings["AzureSearch
            Account"]}.search.windows.net/indexes/{WebConfigurationMana
            ger.AppSettings["AzureSearchIndex"]}/docs?api-
            key={WebConfigurationManager.AppSettings["AzureSearchKey"]}
            &api-version=2016-09-01&";

        public async Task<SearchResult> SearchByCategory(string category)
        {
            using (var httpClient = new HttpClient())
            {
                string nameQuery
                    = $"{QueryString}$filter=category eq '{category}'";
                string response
                    = await httpClient.GetStringAsync(nameQuery);
                return JsonConvert.DeserializeObject<SearchResult>(response);
            }
        }
    }
}
```

この演習では、単純な Search のクエリしか行わないため、Azure Search REST API を直接使用します。より複雑な操作を行う場合は、Azure Search .NET SDK を使用できます。この SDK は、管理操作 (例: Search サービスの作成と拡張、API キーの管理) をサポートしません。

24. RootDialog クラスに、Services フォルダーおよび Model フォルダーにモジュールへの参照と、以下の ExploreCategory メソッドを追加します。新しい ExploreKnowledgeBase インテントに対応し、ユーザーが入力したカテゴリーに属する記事の一覧を Azure Search から取得します。

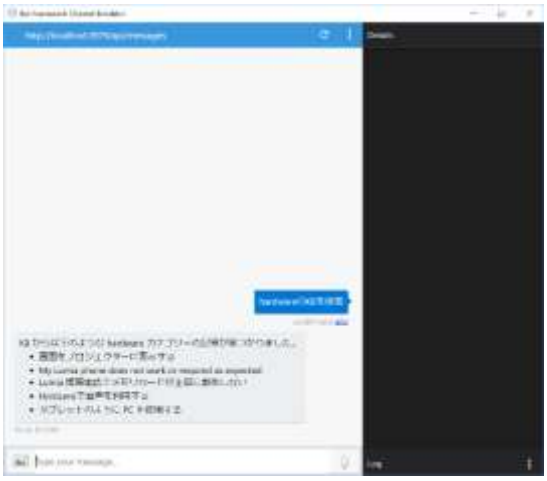
```
using HelpDeskBot.Services;
using HelpDeskBot.Model;
```

```
[LuisIntent("ExploreKnowledgeBase")]
public async Task ExploreCategory(IDialogContext context, LuisResult result)
{
    EntityRecommendation categoryEntityRecommendation;
    result.TryFindEntity("category", out categoryEntityRecommendation);
    var category =
        ((Newtonsoft.Json.Linq.JArray)categoryEntityRecommendation?
        .Resolution["values"])?[0]?.ToString();

    AzureSearchService searchService = new AzureSearchService();

    if (string.IsNullOrWhiteSpace(category))
    {
        await context.PostAsync($"例えば「hardware の KB を検索」"+
            "といった文章で入力してください");
        context.Done<object>(null);
    }
    else
    {
        SearchResult searchResult
            = await searchService.SearchByCategory(category);
        string message;
        if (searchResult.Value.Length != 0)
        {
            message = $"KB から以下のような {category} カテゴリー"+
                "の記事が見つかりました。";
            foreach (var item in searchResult.Value)
            {
                message += $"¥n * {item.Title}";
            }
        }
        else
        {
            message = $"KB から {category} カテゴリーの記事は"+
```

```
                    "見つかりませんでした。 ";
            }
            await context.PostAsync(message);
            context.Done<object>(null);
        }
}
```

25. この時点でボットをテストします。[実行] ボタンをクリックしてアプリを実行します。エミュレーターを起動し、ボットの URL (http://localhost:3979/api/messages) を入力します。

26. 「hardware の KB を検索」と入力します。そのカテゴリーに属する記事がボットにより一覧表示されることを確認します。他のカテゴリー値 (例: networking、software) で試してもかまいません。テストが終了したら、アプリを停止します。



タスク 5: カテゴリーと記事を表示できるようにボットを更新する

このタスクでは、KB をカテゴリー別に取得し、特定のテーマに関する情報を取得できるようにボットのコードを更新します。

27. assets フォルダー からプロジェクトの Model フォルダーにファイル FacetResult.cs、SearchFacets.cs、および Category.cs をコピーします。これらのク

ラスは、Azure Search サービスのクエリに必要となります。

28. AzureSearchService クラスにメソッドを追加します。まずはカテゴリーを取得して、それらを一覧表示するための FetchFacets メソッドを追加します。

```
public async Task<FacetResult> FetchFacets()
{
    using (var httpClient = new HttpClient())
    {
        string facetQuery = $"{QueryString}facet=category";
        string response = await httpClient.GetStringAsync(facetQuery);
        return JsonConvert.DeserializeObject<FacetResult>(response);
    }
}
```

ファセット (facet (=category クエリ)) は、すべての記事の「Category」フィルターをインデックスから取得します (この場合は、software, hardware, networking など)。また、Azure Search は、各ファセットの記事の数を返します。

29. 記事を取得するための SearchByTitle メソッドを追加します。

```
public async Task<SearchResult> SearchByTitle(string title)
{
    using (var httpClient = new HttpClient())
    {
        string nameQuery = $"{QueryString}$filter=title eq '{title}'";
        string response = await httpClient.GetStringAsync(nameQuery);
        return JsonConvert.DeserializeObject<SearchResult>(response);
    }
}
```

このコードでは記事のコンテンツは Azure Search から直接取得されます。しかしながら、本番のシナリオでは、Azure Search はインデックスとしてのみ機能し、記事の全文は Cosmos DB から取得する方が良いでしょう。

30. 全体の検索を行うための Search メソッドを追加します。

```
public async Task<SearchResult> Search(string text)
{
	using (var httpClient = new HttpClient())
	{
		string nameQuery = $"{QueryString}search={text}";
		string response = await httpClient.GetStringAsync(nameQuery);
		return JsonConvert.DeserializeObject<SearchResult>(response);
	}
}
```

> Azure Search では、search=... クエリは、インデックス内のすべての検索可能フィールドの用語を 1 つ以上検索し、Google や Bing などの検索エンジンと同様に機能します。filter=... クエリは、インデックス内のすべてのフィルター可能フィールドでブール式を評価します。検索クエリとは異なり、フィルター クエリは、フィールドの正確なコンテンツのマッチングを行います。つまり、文字列フィールドの大文字と小文字が区別されます。

31. assets フォルダー からプロジェクトの Util フォルダーに CardUtil.cs ファイルをコピーします。このクラスは、Azure Search からの記事の一覧で ThumbnailCard のカルーセルを作成するのに使用されます。

> ユーザーに対してリッチなカードを示す方法の詳細については、こちらの記事を参照してください。
>
> https://docs.microsoft.com/ja-jp/bot-framework/nodejs/bot-builder-nodejs-send-rich-cards

32. assets フォルダー からプロジェクトの Dialogs フォルダーに SearchScorable.cs と ShowArticleDetailsScorable.cs をコピーします。これらのクラスは、ボットに送信されるすべてのメッセージから検索サービスを起動し、スコア付けを行う ”Scoreable” です。

- SearchScorable は、メッセージが「を検索」で終わる場合に起動され、AzureSearchService の Search メソッドを呼び出します。
- ShowArticleDetailsScorable は、メッセージが「KB で検索」で終わる場合に起動され、AzureSearchService の SearchByTitle メソッドを呼び出します。

> Scorable は、ユーザーから送信されるメッセージを分析し、定義したロジックに基づいてメッセージにスコアを付けます。最もスコアの高い Scorable がメッセージを処理する機会を「獲得」します。ボットで実装するグローバル コマンドごとに Scorable を作成することで、グローバル メッセージ ハンドラーを実装できます。Scorable の詳細については、こちらのサンプルを参照してください。
>
> https://github.com/Microsoft/BotBuilder-Samples/tree/master/CSharp/core-GlobalMessageHandlers

33. Global.asax.cs を開き、以下の using ステートメントを追加します。

```
using Autofac;
using HelpDeskBot.Dialogs;
using Microsoft.Bot.Builder.Dialogs;
using Microsoft.Bot.Builder.Scorables;
using Microsoft.Bot.Connector;
```

34. Application_Start メソッドを以下のコードに置き換え、会話コンテナーに Scorable を登録します。

```
protected void Application_Start()
{
    GlobalConfiguration.Configure(WebApiConfig.Register);

    var builder = new ContainerBuilder();

    builder.RegisterType<SearchScorable>()
        .As<IScorable<IActivity, double>>()
        .InstancePerLifetimeScope();

    builder.RegisterType<ShowArticleDetailsScorable>()
        .As<IScorable<IActivity, double>>()
```

```
            .InstancePerLifetimeScope();

        builder.Update(Conversation.Container);
}
```

35. RootDialig.cs の ExplorerCategory を以下のコードに置き換えます。このコードでは、ExplorerCategory に変数 originalText を追加し、元のメッセージを保持します。また、ボットが元のメッセージでカテゴリーを検出しなかった場合はカテゴリーの一覧を取得するように変更します。

```
[LuisIntent("ExploreKnowledgeBase")]
public async Task ExploreCategory(IDialogContext context, LuisResult result)
{
        EntityRecommendation categoryEntityRecommendation;
        result.TryFindEntity("category", out categoryEntityRecommendation);
        var category =
                ((Newtonsoft.Json.Linq.JArray)categoryEntityRecommendation?
                .Resolution["values"])?[0]?.ToString();
        var originalText = result.Query;

        AzureSearchService searchService = new AzureSearchService();
        if (string.IsNullOrWhiteSpace(category))
        {
                FacetResult facetResult = await searchService.FetchFacets();
                if (facetResult.Facets.Category.Length != 0)
                {
                        List<string> categories = new List<string>();
                        foreach (Category searchedCategory in facetResult.Facets.Category)
                        {
                                categories.Add($"{searchedCategory.Value}({searchedCatego
                                ry.Count})");
                        }

                        PromptDialog.Choice(context, this.AfterMenuSelection, categories,
                           "お探しの答えが KB の中にあるか確認しましょう。"+
                           "どのカテゴリーをご覧になりますか？");
                }
        }
        else
        {
                SearchResult searchResult
```

```
                = await searchService.SearchByCategory(category);

            if (searchResult.Value.Length != 0)
            {
                await context.PostAsync($"{category}  には以下のような KB が" +
                    "見つかりました。" +
                    "**More details**  をクリックすると詳細が表示されます。");
            }

            await CardUtil.ShowSearchResults(context, searchResult,
                $"KB から{category}  カテゴリーの記事は見つかりませんでした。");

            context.Done<object>(null);
        }
}
```

36. AfterMenuSelection メソッドを追加します。このメソッドは、ユーザーが検索するカテゴリーを選択する際に呼び出されます。

```
public virtual async Task AfterMenuSelection(IDialogContext context,
IAwaitable<string> result)
{
        this.category = await result;
        this.category = System.Text.RegularExpressions.Regex.Replace
            (this.category, @"¥([^)]*¥)", string.Empty);
        AzureSearchService searchService = new AzureSearchService();

        SearchResult searchResult
            = await searchService.SearchByCategory(this.category);
        await context.PostAsync($"{this.category}には以下のような KB が" +
            "見つかりました。" +
            "**More details**  をクリックすると詳細が表示されます。");
        await CardUtil.ShowSearchResults(context, searchResult, $"KB  から" +
            "{this.category}  カテゴリーの記事は見つかりませんでした。");

        context.Done<object>(null);
}
```

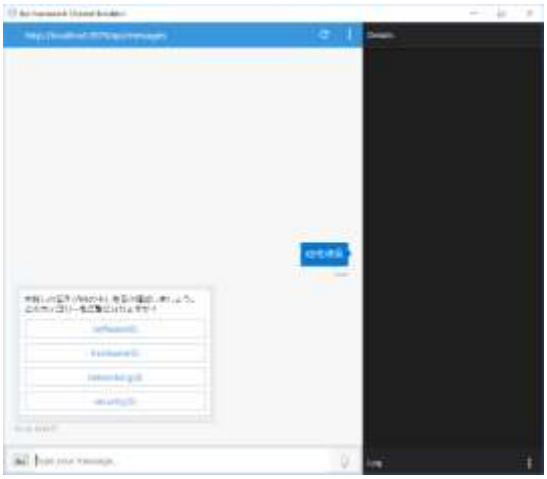
37. 最後に、KB 検索機能が含まれるように Help メソッドのテキストを更新します。

```
[LuisIntent("Help")]
public async Task Help(IDialogContext context, LuisResult result)
{
        await context.PostAsync("Help Desk Bot  です。" +
                "サポートデスク受付チケットの発行、KB 検索ができます。¥n¥n" +
                "どんなことにお困りですか？例えば「パスワードをリセットしたい」" +
                "「印刷できない」といった文章で入力してください。");
        context.Done<object>(null);
}
```

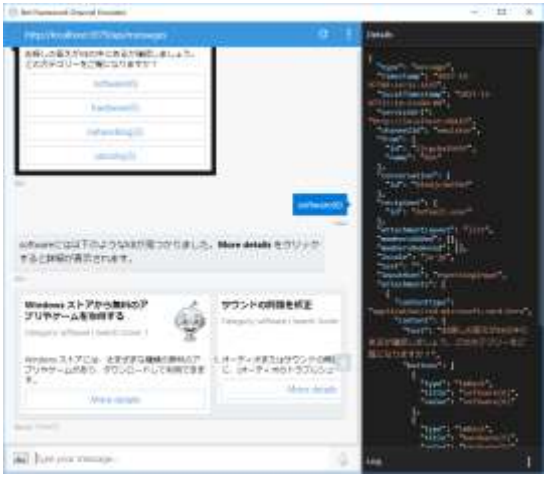
タスク 6: エミュレーターからボットをテストする

38. [実行] ボタンをクリックしてアプリを実行し、エミュレーターを開きます。ボットの URL (http://localhost:3979/api/messages) を入力します。

39. 「KB を検索」と入力します。Cosmos DB にアップロードした記事カテゴリーの一覧、および各カテゴリーの記事の数が表示されます。



40. 一覧表示されたカテゴリーのいずれかをクリックすると、そのカテゴリーの記事が表示されます。



41. 記事の [More Details] ボタンをクリックすると、記事の全文が表示されます。

42. 特定のカテゴリーを検索してみてもかまいません。「software を検索」と入力すると、そのカテゴリーに属する記事がいくつか表示されます。

43. 同様に、特定のトピックに関する記事を検索してみてもかまいません。たとえば、「OneDrive を検索」と入力します。

検索によって返されるドキュメントごとにスコア（関連度）が返されることを確認してください。

おつかれさまでした。以上で、Azure を利用したインテリジェンスを活用した、ユーザーと対話を行うボットアプリケーションの完成です。

まとめ

このハンズオン ラボでは、以下の方法について学習しました。

- Bot Framework (.NET C#版) を使用して、ボットアプリケーションを作成する
- Cognitive Services LUIS を利用して、ボットに自然言語処理を実装する
- Azure CosmosDB および Azure Search を利用して、ボットにリスト検索機能を実装する
