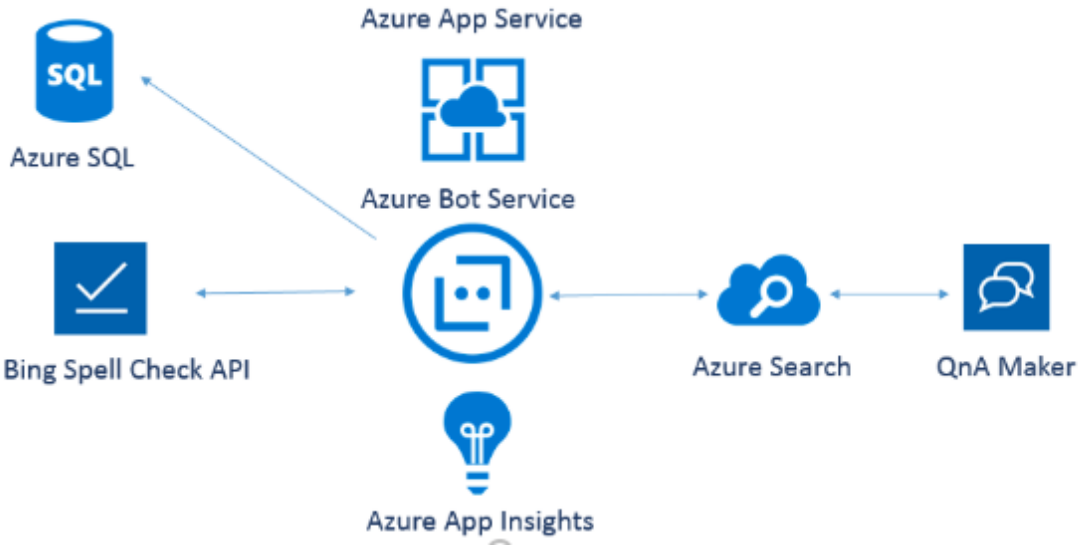
Developing a ChatBot using Microsoft Bot Framework

This Blog demonstrates the development of a ChatBot for an organization. The ChatBot helps customers to get answers to their questions.

Standards Bot Logic flow and components



Logic Flow:

1. Customer asks a question to the Chat Bot.
2. Bot queries the Bing Spell Check API with question asked by the Customer.
3. After Spell Check, Bot queries QnA Maker with the updated Query (if there was a spelling mistake, it is updated with correct spelling in step 2) which leverages Azure Search to get the answer to the Customer's question from QnA Knowledge Base.
4. Bot responds to the Customer with the answer.
5. Conversation history is logged in Azure SQL table.
6. Customer satisfaction scores are logged and also errors are logged and visualized using into Azure App Insights.
7. If Bot is unable to answer Customer's question, a Form Builder is used to capture Customer contact information, which is emailed to company's IT department so that they can contact the Customer later with an answer.

This ChatBot uses Microsoft Bot Framework.

Bot code Flow:

When a User sends a message to ChatBot, it is received by a Controller, since Bot is an underlying .net web application.

```
namespace EnkayTech.ChatBot.QnABot
{
    [BotAuthentication]
    public class MessagesController : ApiController
    {
        private static readonly bool IsSpellCorrectionEnabled = bool.Parse(ConfigurationManager.AppSettings["IsSpellCorrectionEnabled"]);
        private BingSpellCheckService spellService = new BingSpellCheckService();
        /// <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)
            {
                try
                {
                    var text = await this.spellService.GetCorrectedTextAsync(activity.Text);
                    if (text != activity.Text)
                    {
                        activity.Text = text;
                    }
                }

                catch (Exception ex)
                {

                }
                await Conversation.SendAsync(activity, () => new Dialogs.QnaDialog());
                //await Conversation.SendAsync(activity, () => new Dialogs.RootDialog());
            }
```

See the above code snippet from MessagesController. It receives the User Activity.

Activity-

The Connector uses an Activity object to pass information back and forth between bot and channel (user). The most common type of activity is message, but there are other activity types that can be used to communicate various types of information to a bot or channel.

If the Activity is of type Message, it is passed into Bing spell check API to check and correct spelling mistakes (typos etc.) and corrected text is sent to QnA dialogue to check the User Query in the QnA Knowledge Base. QnA Dialogue calls QnA Maker Service by passing QnAHost, Knowledge Base ID and End point Key which are available when a QnA Service is created and Knowledge Base is

```
{
    [Serializable]
    public class QnaDialog : IDialog<object>
    {
        public QnAMakerService _QnAService = new QnAMakerService(
                ConfigurationManager.AppSettings["QnAHost"],
                ConfigurationManager.AppSettings["QnAKnowledgebaseId"],
                ConfigurationManager.AppSettings["QnAEndPointKey"]);
        [NonSerialized()]
        private TelemetryClient telemetry = new TelemetryClient();


        private async Task QuestionFeedback(IDialogContext context, IAwaitable<bool> result)...
```

Within, QnAService Get Answer () method is called by passing the context object and Corrected text from the activity.

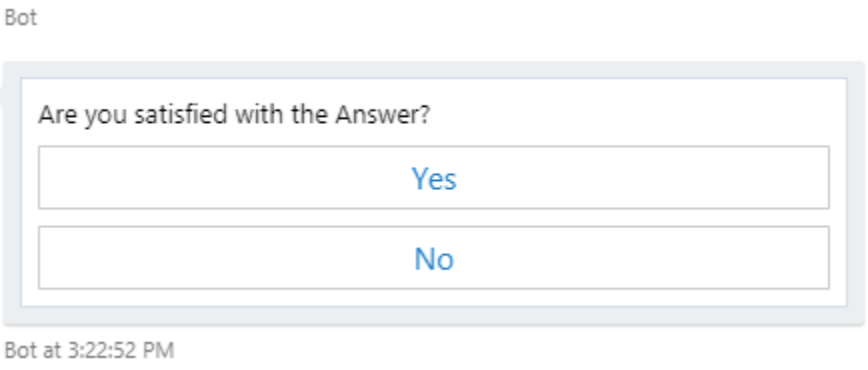
```
public async Task MessageReceivedASync(IDialogContext context, IAwaitable<IMessageActivity> result)
{
    //telemetry = new TelemetryClient();
    string safeText = HttpUtility.UrlEncode(((IMessageActivity)context.Activity).Text);
    string answer = string.Empty;
    //string answer = await _QnAService.GetAnswer(safeText, context);
    if (((IMessageActivity)context.Activity).Text.Contains("fire extinguisher requirements"))...
    else if (((IMessageActivity)context.Activity).Text.Contains("follow up appointments"))...
    else if (((IMessageActivity)context.Activity).Text.Contains("vials are considered to be"))...
    else
    {
        IList<Answer> answerList = await _QnAService.GetAnswer(safeText, context);

        if (answerList.Count > 0)
        {
            answer = answerList[0].answer;
        }
        if (string.IsNullOrEmpty(answer))
        {
            //await context.PostAsync(MakeRootDialog());
            //await context.PostAsync("No good match found in KB.");
        }
        if (answerList.Count > 0)
        {
            if (answerList[0].Score > 70 || answerList[0].Source == "qna_chitchat_the_professional.tsv")
            {
                await context.PostAsync(answer);
                //telemetry.TrackEvent("UserSatisfied");
            }
            else
            {
                await context.PostAsync(answer);
                PromptDialog.Confirm(context, QuestionFeedback, "Are you satisfied with the Answer?");
            }
        }
    }

}
```

When a user query is matched against a knowledge base, QnA Maker returns relevant answers, along with a confidence score. This score indicates the confidence that the answer is the right match for the given user query.

The confidence score is a number between 0 and 100. A score of 100 is likely an exact match, while a score of 0 means, that no matching answer was found. The higher the score- the greater the confidence in the answer.

If the confidence score is less than 70, after the answer is returned to user Bot asks user if he/she is satisfied with the answer. PromptDialog.confirm() method is used to ask the user to confirm an action with a **yes/no** response. The prompt returns the user's response as an IPromptConfirmResult.

```
if (answerList.Count > 0)
{
    answer = answerList[0].answer;
}
if (string.IsNullOrEmpty(answer))
{
    //await context.PostAsync(MakeRootDialog());
    //await context.PostAsync("No good match found in KB.");
}
if (answerList.Count > 0)
{
    if (answerList[0].Score > 70 || answerList[0].Source == "qna_chitchat_the_professional.tsv")
    {
        await context.PostAsync(answer);
        //telemetry.TrackEvent("UserSatisfied");
    }
    else
    {
        await context.PostAsync(answer);
        PromptDialog.Confirm(context, QuestionFeedback, "Are you satisfied with the Answer?");
    }
}
```



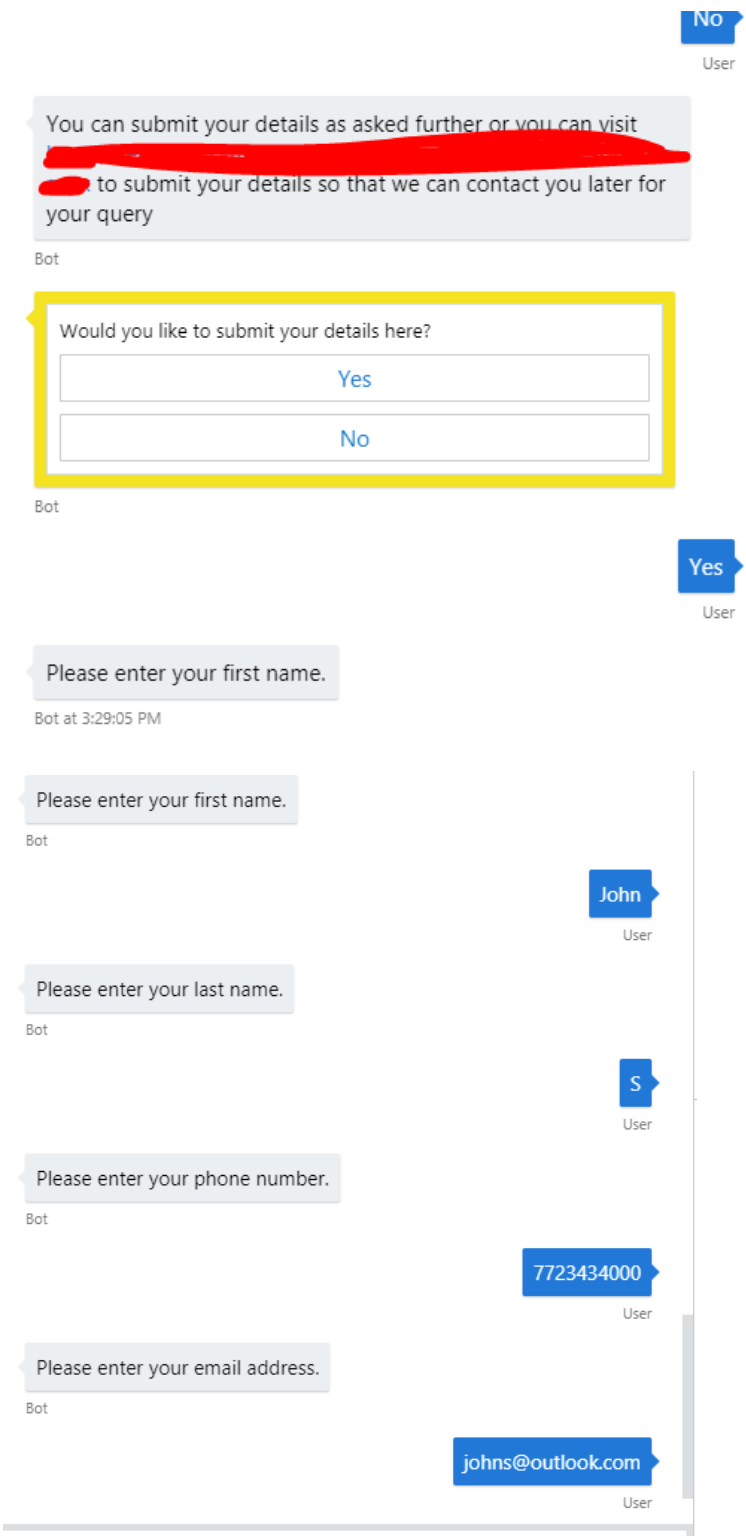
If the user selects No, Bot asks user to submit user information with the query so that user can be contacted later regarding the query.

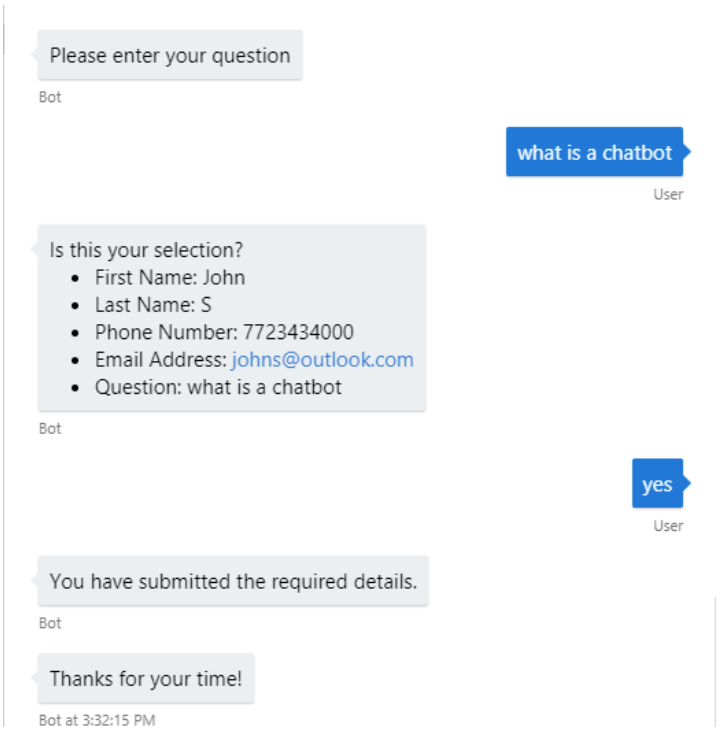
User’s feedback is logged into App insights using TelemetryClient class.

```
private async Task QuestionFeedback(IDialogContext context, IAwaitable<bool> result)
{
    //var config = new TelemetryConfiguration();
    //config.InstrumentationKey = ConfigurationManager.AppSettings["ikey"];
    //config.TelemetryChannel = new Microsoft.ApplicationInsights.WindowsServer.TelemetryChannel.ServerTelemetryChannel();
    //config.TelemetryChannel = new Microsoft.ApplicationInsights.Channel.InMemoryChannel(); // Default channel
    //config.TelemetryChannel.DeveloperMode = true;
    telemetry = new TelemetryClient();
    string convID = context.MakeMessage().Conversation.Id;
    bool UserResp = await result;
    if (!UserResp)
    {

        await context.PostAsync("You can submit your details as asked further or you can visit [redacted]
        PromptDialog.Confirm(context, AfterQuestionFeedback, "Would you like to submit your details here?");
        //telemetry.TrackEvent("UserUnsatisfied");

        var properties = new Dictionary<string, string>
        { {"Conversation ID", convID} };
        var metrics = new Dictionary<string, double>
        { {"UserFeedback", 0} };
        // Send the event:
        telemetry.TrackEvent("UserUnsatisfied", properties, metrics);
    }
    else
    {

        //await context.PostAsync("Thanks! Do you have any other question?");
        PromptDialog.Confirm(context, AfterAskingOtherQuestion, "Thanks! Do you have any other question?");
        //telemetry.TrackEvent("UserSatisfied");
        var properties = new Dictionary<string, string>
        { {"Conversation ID", convID} };
        var metrics = new Dictionary<string, double>
        { {"UserFeedback", 1} };
        // Send the event:
        telemetry.TrackEvent("UserSatisfied", properties, metrics);
```

User is prompted to go to company's website or to enter the information in the chat window itself. These details are captured via form builder.

```
namespace EnkayTech.ChatBot.QnABot
{
    [Serializable]
    public class UserDetails
    {
        [Prompt("Please enter your {&}.")]
        public string FirstName;

        [Prompt("Please enter your {&}.")]
        public string LastName;

        [Prompt("Please enter your {&}.")]
        public string PhoneNumber;

        [Prompt("Please enter your {&}.")]
        [Pattern(@"[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Za-z]{2,}")]
        public string EmailAddress;

        [Prompt("Please enter your question")]
        public string Question;

        public static IForm<UserDetails> BuildForm()
        {
            //return new FormBuilder<UserDetails>().Build();
            return new FormBuilder<UserDetails>().OnCompletion(async (context, UserDetails) =>
                    {
                        // Tell the user that the form is complete
                        await context.PostAsync("You have submitted the required details.");
                    })
                    .Build();
        }
    }
}
```


No
User
You can submit your details as asked further or you can visit
to submit your details so that we can contact you later for your query
Bot
Would you like to submit your details here?
Yes
No
Bot
Yes
User
Please enter your first name.
Bot at 3:29:05 PM
Please enter your first name.
Bot
John
User
Please enter your last name.
Bot
S
User
Please enter your phone number.
Bot
7723434000
User
Please enter your email address.
Bot
johns@outlook.com
User



Rich Card usage-

For an enhanced graphical experience, Microsoft Bot Framework allows the usage of Carousels with hero cards and adaptive cards.

```
else if (((IMessageActivity)context.Activity).Text.Contains("vials are considered to be"))
{

    IList<Answer> answerList = await _QnAService.GetAnswer(safeText, context);

    if (answerList.Count > 0)
    {
        answer = answerList[0].answer;
    }
    if (string.IsNullOrEmpty(answer))
    {
        //await context.PostAsync(MakeRootDialog());
        //await context.PostAsync("No good match found in KB.");
    }
    if (answerList.Count > 0)
    {

            await context.PostAsync(answer);
            context.Call(new ThumbnailCarouselDialog(), AfterThumbnailCarousel);
    }

}
```

```
[Serializable]
public class ThumbnailCarouselDialog : IDialog<string>
{

    public async Task StartAsync(IDialogContext context)
    {
        await context.PostAsync("You can check below additional resources:");
        var reply = context.MakeMessage();

        reply.AttachmentLayout = AttachmentLayoutTypes.Carousel;
        reply.Attachments = GetCardsAttachments();
        await context.PostAsync(reply);

        QnaDialog r1 = new QnaDialog();
        context.Wait(r1.MessageReceivedASync);
    }

    public static IList<Attachment> GetCardsAttachments()
    {
        return new List<Attachment>()
        {
            GetHeroCard(
                "One and Only Compaign",
                new CardImage(url: "[redacted]/chatbotresources/oneandonlycampaign.png"),
                new CardAction(ActionTypes.OpenUrl, "Visit this", value: "http://www.oneandonlycampaign.org/")),
            GetHeroCard(
                "Frequently Asked Questions (FAQs)\nregarding Safe Practices for Medical Injections",
                new CardImage(url: "[redacted]/chatbotresources/cdc-safe-Injection-practices.png"),
                new CardAction(ActionTypes.OpenUrl, "Visit this", value: "https://www.cdc.gov/injectionsafety/providers/provider_faqs_multi
            GetHeroCard(
                "Safe Injection Practices to Prevent\nTransmission of Infections to Patients",
                new CardImage(url: "[redacted]/chatbotresources/cdc-medical-injections.png"),
                new CardAction(ActionTypes.OpenUrl, "Visit this", value: "https://www.cdc.gov/injectionsafety/ip07_standardprecaution.html"
        };
```

Bot

You can check below additional resources:

Bot

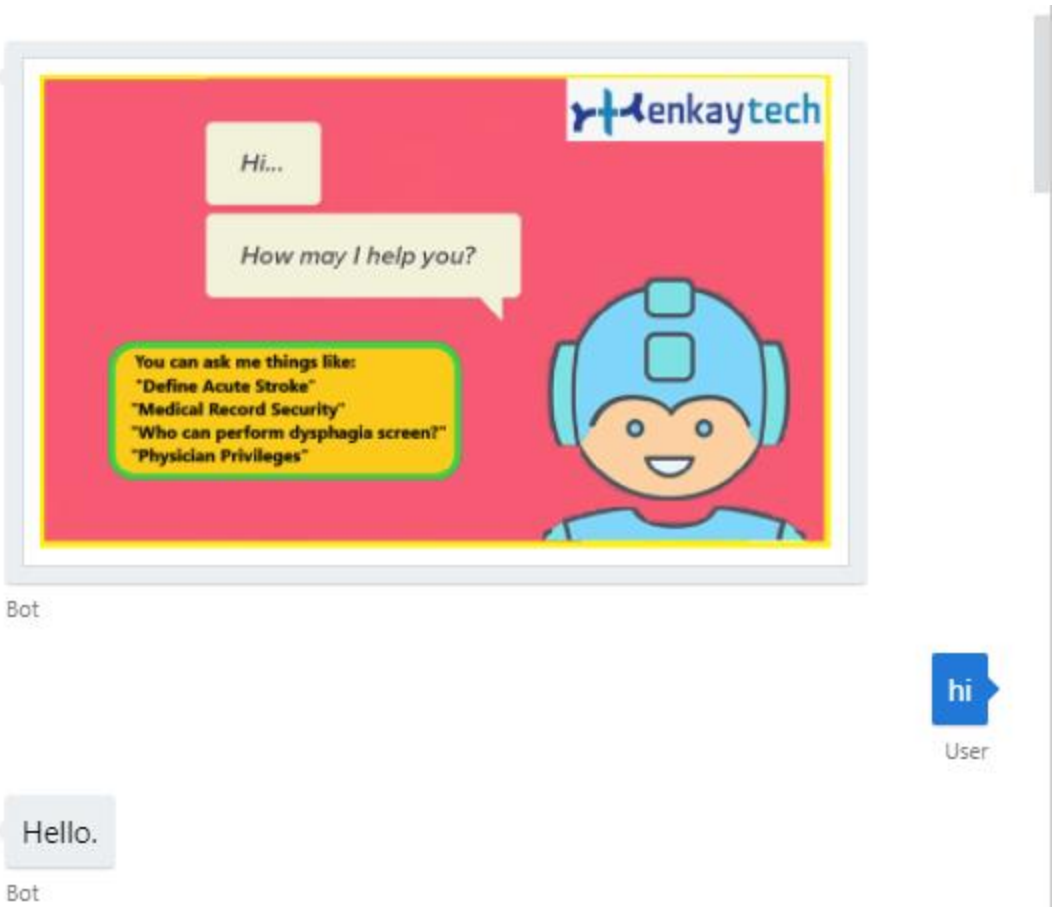
Adaptive cards- Adaptive Cards are a great fit for Bots. They let you author a card once and have it render beautifully inside the chat. Our Bot uses adaptive card to introduce itself.

```
else if (message.Type == ActivityTypes.ConversationUpdate)
{
    IConversationUpdateActivity update = message;
    using (var scope = DialogModule.BeginLifetimeScope(Conversation.Container, message))
    {
        var client = scope.Resolve<IConnectorClient>();
        if (update.MembersAdded.Any())
        {
            var reply = message.CreateReply();
            foreach (var member in update.MembersAdded ?? System.Array.Empty<ChannelAccount>())
            {
                // if the bot is added, then
                if (member.Id == update.Recipient.Id)
                {
                    // the bot is always added as a user of the conversation, since we don't
                    // want to display the adaptive card twice ignore the conversation update triggered by the bot
                    //if(newMember.Name.ToLower() != "tjc-dev-innovation-bot")
                    //{
                    try
                    {
                        // read the json in from our file
                        string json = File.ReadAllText(HttpContext.Current.Request.MapPath(@"\AdaptiveCards\WelcomeCard.json"));
                        // use Newtonsofts JsonConvert to deserialized the json into a C# AdaptiveCard object
                        AdaptiveCards.AdaptiveCard card = JsonConvert.DeserializeObject<AdaptiveCards.AdaptiveCard>(json);
                        // put the adaptive card as an attachment to the reply message
                        reply.Attachments.Add(new Attachment
                        {
                            ContentType = AdaptiveCard.ContentType,
                            Content = card
                        });
```

WelcomeCard.json

Schema: http://adaptivecards.io/schemas/adaptive-card.json

```
{
  "$schema": "http://adaptivecards.io/schemas/adaptive-card.json",
  "type": "AdaptiveCard",
  "version": "1.0",
  "body": [
    {
      "type": "ColumnSet",
      "columns": [
        {
          "type": "Column",
          "width": "stretch",
          "items": [
            {
              "type": "Image",
              "url": "[redacted]/chatbotresources/Chatbot-Welcome.png",
              "horizontalAlignment": "center",
              "size": "stretch"
            }
          ]
        }
      ]
    }
  ]
}
```



Logging the conversation-

Conversation between Bot and User can be logged into Azure Storage. One of the most common operations when working with conversational history is to intercept and log message activities between bots and users. The IActivityLogger interface contains the definition of the functionality that a class needs to implement to log message activities. The ActivityLogger implementation of the IActivityLogger interface writes message activities to the Azure Table storage. Conversation ID is also logged for each activity.

```
public class ActivityLogger : IActivityLogger
{
    public async Task LogAsync(IActivity activity)
    {
        string ConversationID = activity.Conversation.Id;
        //string UniqueID = System.Guid.NewGuid().ToString();
        string FromName = activity.From.Name;
        string FromID = activity.From.Id;
        string ToName = activity.Recipient.Name;
        string ToID = activity.Recipient.Id;
        string Message = activity.AsMessageActivity()?.Text;
        InsertRecordToTable(ConversationID, FromName, FromID, ToName, ToID, Message);
   // Debug.WriteLine($"From:{activity.From.Name} - To:{activity.Recipient.Name} - Message:{activity.AsMessageActivity()?.Text}");
    }
    public string getConversationID(IActivity activity)
    {
        string ConvID = activity.Conversation.Id;
        return ConvID;
    }
    public static void InsertRecordToTable(string ConversationID, string FromName, string FromID, string ToName, string ToID, string Message
    {
        CloudStorageAccount cloudStorageAccount = CloudStorageAccount.Parse(ConfigurationManager.AppSettings["AzureWebJobsStorage"]);
        CloudTableClient tableClient = cloudStorageAccount.CreateCloudTableClient();
        string tableName = "StandardsBotConversationLogger";
        CloudTable cloudTable = tableClient.GetTableReference(tableName);
        cloudTable.CreateIfNotExists();
        ConversationDetails convdetails = new ConversationDetails();
        convdetails.ConversationID = ConversationID;
        //convdetails.UniqueID = UniqueID;
        convdetails.FromName = FromName;
        convdetails.FromId = FromID;
        convdetails.ToName = ToName;
        convdetails.ToId = ToID;
        convdetails.Message = Message;
        convdetails.AssignPartitionKey();
        convdetails.AssignRowKey();

        TableOperation tableOperation = TableOperation.Insert(convdetails);
        cloudTable.Execute(tableOperation);
    }
```

The logging activity is an event that takes place for the lifetime of the bot application. Application_Start method is called when the application starts and lasts for the lifetime of the application.

Global.asax allows us to write event handlers that react to important life-cycle events. Global.asax events are never called directly by the user. They are called automatically in response to application events. For this lab, we must register DebugActivityLogger in Application_Start (in Global.asax) as follows.

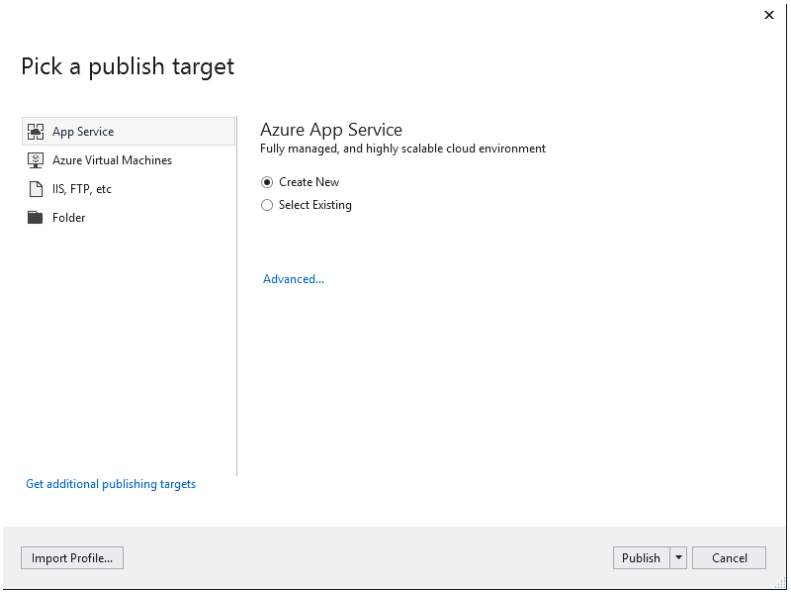
```
public class WebApiApplication : System.Web.HttpApplication
{
    protected void Application_Start()
    {

        Conversation.UpdateContainer(
        builder =>
        {
            builder.RegisterModule(new AzureModule(Assembly.GetExecutingAssembly()));

            // Using Azure Table Storage
            var store = new TableBotDataStore(ConfigurationManager.AppSettings["AzureWebJobsStorage"]); // requires Microsoft

            builder.Register(c => store)
                .Keyed<IBotDataStore<BotData>>(AzureModule.Key_DataStore)
                .AsSelf()
                .SingleInstance();
            builder.RegisterType<ActivityLogger>().AsImplementedInterfaces().InstancePerDependency();


        });
        GlobalConfiguration.Configure(WebApiConfig.Register);
    }
}
```

The Application_Start method is called only one time during the life cycle of an application. You can use this method to perform startup tasks.

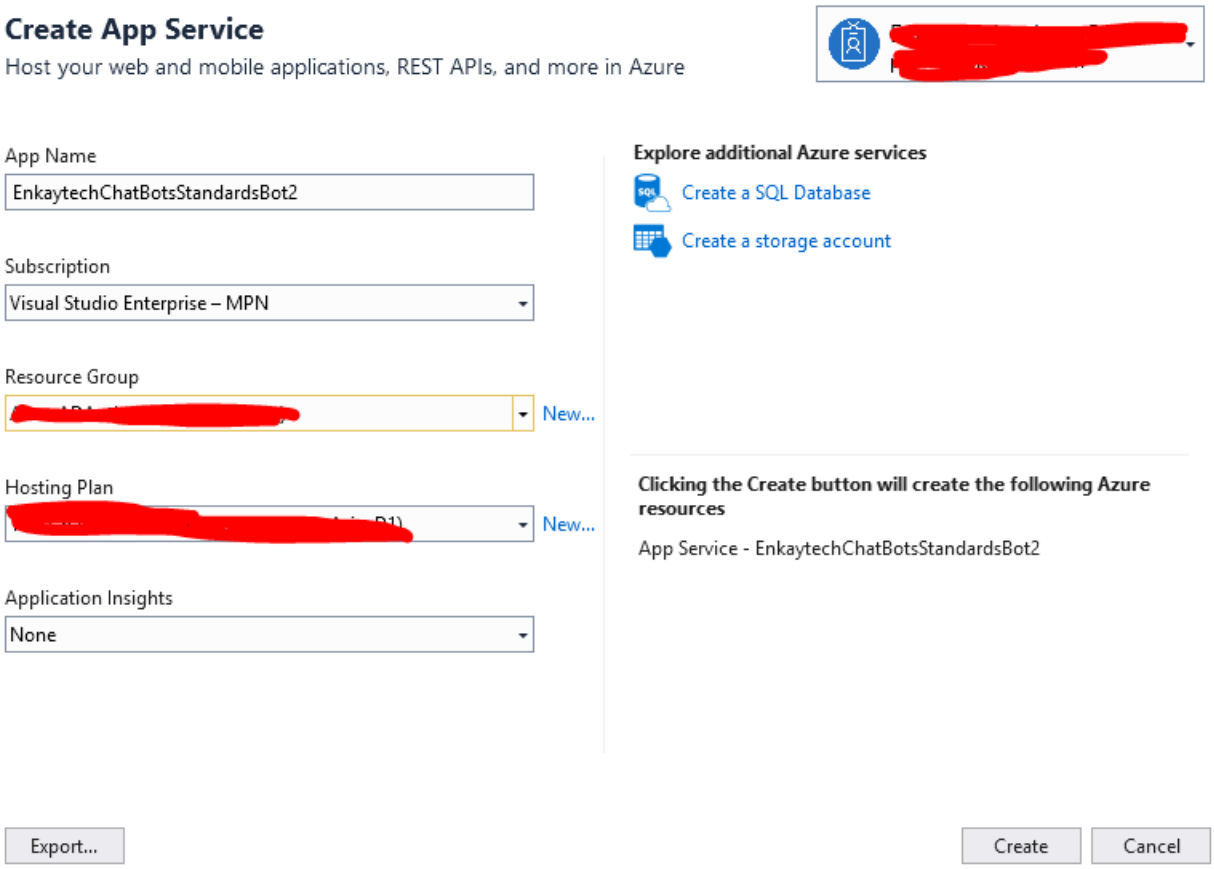
Publish bot to Bot Service

You can publish your bot into a running bot in Bot Service using Visual Studio.

1. In the Solution Explorer pane, right-click your project, and click Publish... The Publish window opens.
2. In the Publish window, click Create new profile, click Import profile, and click OK.



This will create an app service to host the Bot in Azure Portal.



Register bot with Azure Bot Service

Once you have tested your bot locally, you can use the Bot Service to connect it to other channels, you will need to register your bot with the Bot Service.

You only need to register your bot if it is not hosted in Azure. If you created a bot through the Azure portal then your bot is already registered with the Bot Service.

Create a Bot Channels Registration

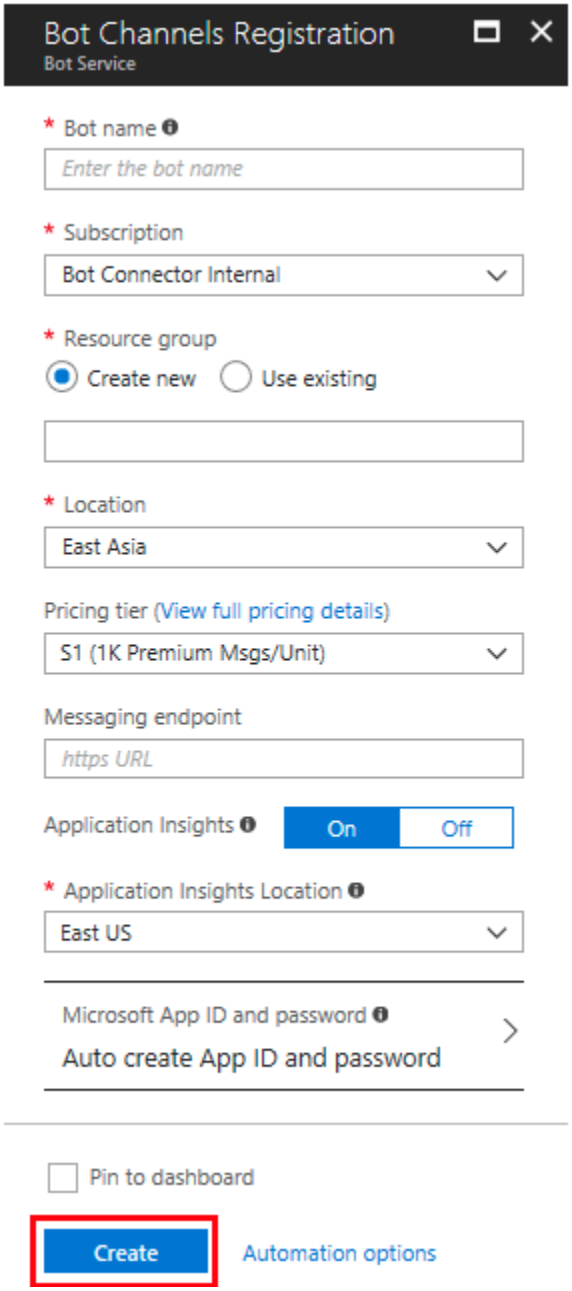
You need a Bot Channels Registration bot service to be able to use Bot Service functionality. A registration bot lets you connect your bot to channels.

To create a Bot Channels Registration, do the following:

Click the New button found on the upper left-hand corner of the Azure portal, then select AI + Cognitive Services > Bot Channels Registration.

A new blade will open with information about the Bot Channels Registration. Click the Create button to start the creation process.

In the Bot Service blade, provide the requested information about your bot.



In the Messaging endpoint, provide the URL of the published Bot.

Click the ***Auto create App ID and password*** button.

A new *blade* will open

Click the ***Auto create App ID and password*** button

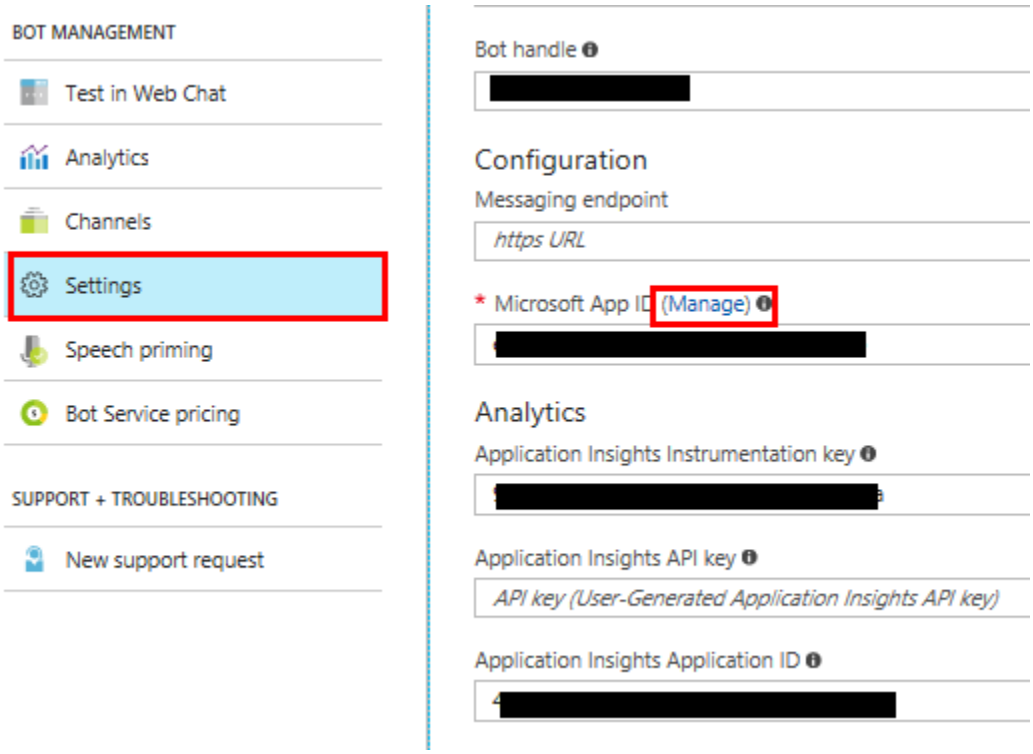
The *blade* will then close

Bot Channels Registration bot service only has a *MicrosoftAppID*. You need to generate the password manually and save it yourself. You will need this password if you want to test your bot using the emulator.

To generate a MicrosoftAppPassword, do the following:

1. From the **Settings** blade, click **Manage**. This is the link appearing by the **Microsoft App ID**. This link will open a window where you can generate a new

password.

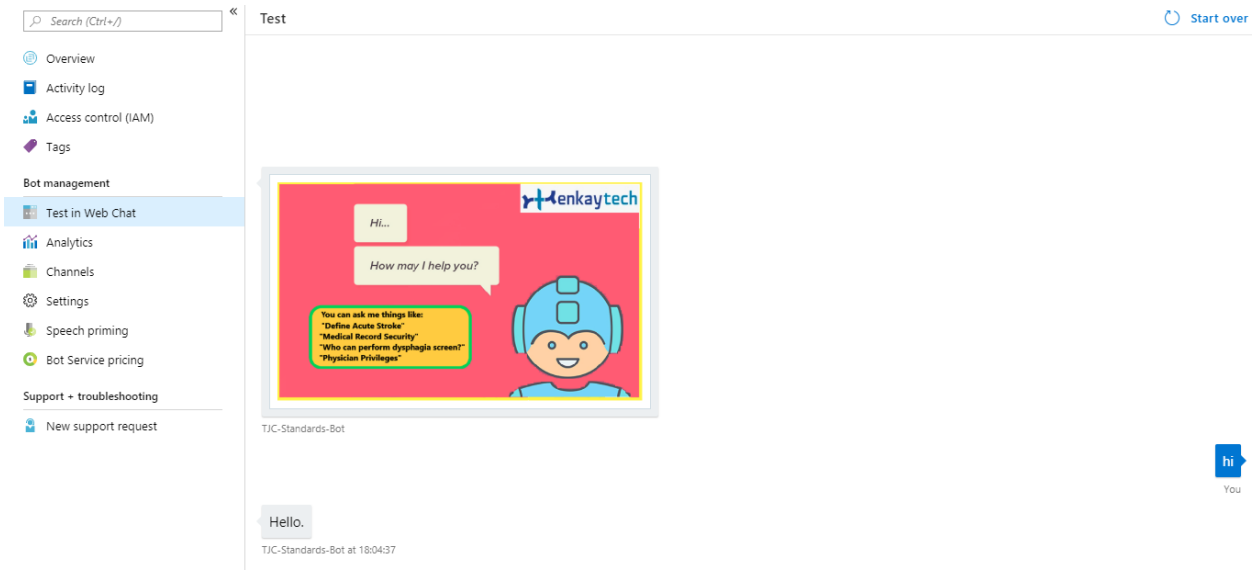


2. Click **Generate New Password**. This will generate a new password for your bot. Copy this password and save it to a file. This is the only time you will see this password. If you do not have the full password saved, you will need to repeat the process to create a new password should you need it later.

3. Now open web.config of bot application and add Microsoft App Id and password.

```
<?xml version="1.0" encoding="utf-8"?>
<!--
  For more information on how to configure your ASP.NET application, please visit
  http://go.microsoft.com/fwlink/?LinkId=301879
  -->
<configuration>
  <appSettings>
    <!-- update these with your BotId, Microsoft App Id and your Microsoft App Password-->
    <add key="BotId" value="digitalassistant01" />
    <add key="MicrosoftAppId" value="[redacted]" />
    <add key="MicrosoftAppPassword" value="[redacted]" />
  </appSettings>
  <!--
    For a description of web.config changes see http://go.microsoft.com/fwlink/?LinkId=235367.

    The following attributes can be set on the <httpRuntime> tag.
      <system.Web>
        <httpRuntime targetFramework="4.6" />
      </system.Web>
  -->
  <system.web>
    <customErrors mode="Off" />
    <compilation debug="true" targetFramework="4.6" />
    <httpRuntime targetFramework="4.6" />
```

4. Publish application again to azure.
5. Now go to bot channels registration which you registered in Azure
6. Click Test in Web Chat and test chat with Bot.



The Bot can be connected to web chat and other channels as well using the Azure portal.