isolutions

**Hands-on Lab**

# Bot in a Day

# 1 Einrichtung der Entwicklungsumgebung

Ihre Entwicklungsumgebung soll folgende Voraussetzungen erfüllen:

- Windows 10
- Visual Studio 2017
- .NET Framework 4.6+

## 1.1 Installation der Komponenten für Bot-Entwicklung

Folgende Komponenten müssen zusätzlich installiert werden:

"Bot Application" Projektvorlage für Visual Studio:
https://marketplace.visualstudio.com/items?itemName=BotBuilder.BotBuilderV3

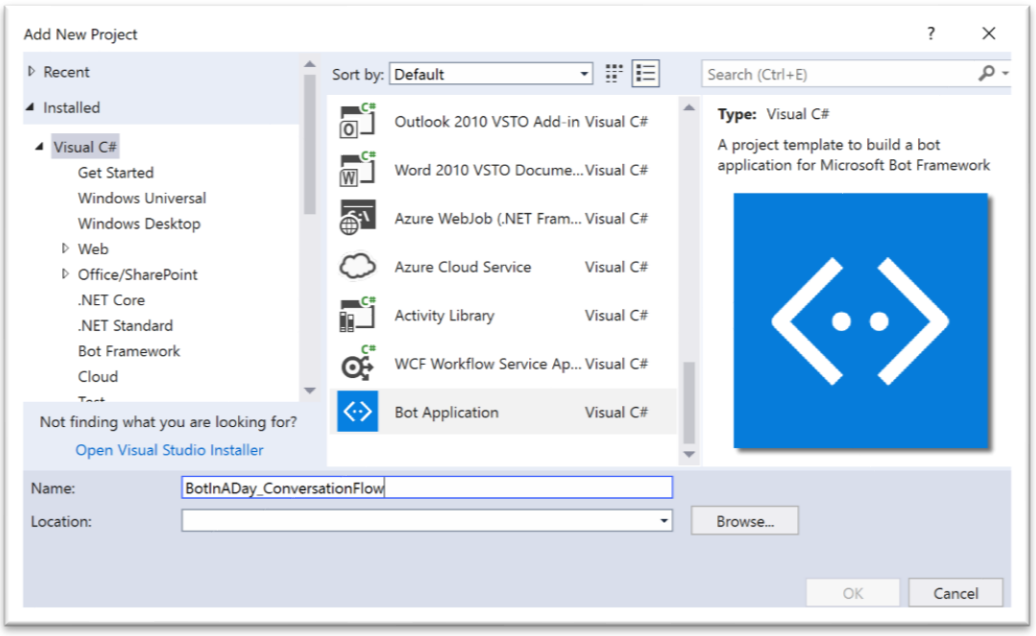
Bot Framework Emulator:
https://github.com/Microsoft/BotFramework-Emulator/releases/tag/v3.5.35

Optional: Azure Cosmos DB Emulator:
https://aka.ms/cosmosdb-emulator

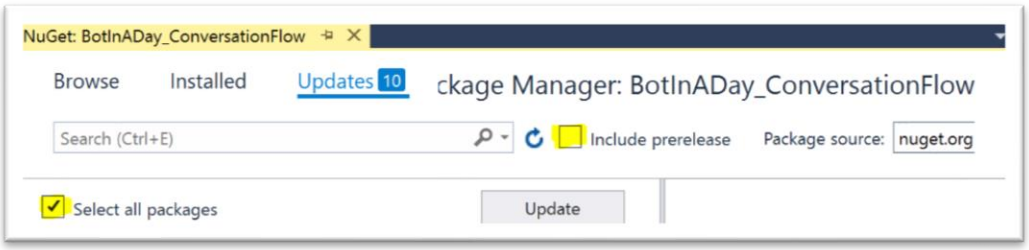
# 2 ConversationFlow

## 2.1 Erstellen eines neuen Bot-Projekts

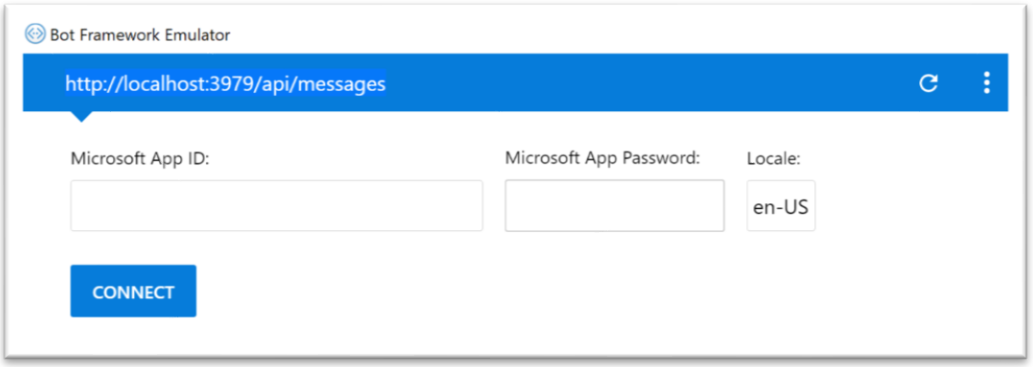
Erstellen Sie ein neues Projekt namens BotInADay_ConversationFlow aus der "Bot Application"-Vorlage in Visual Studio 2017:



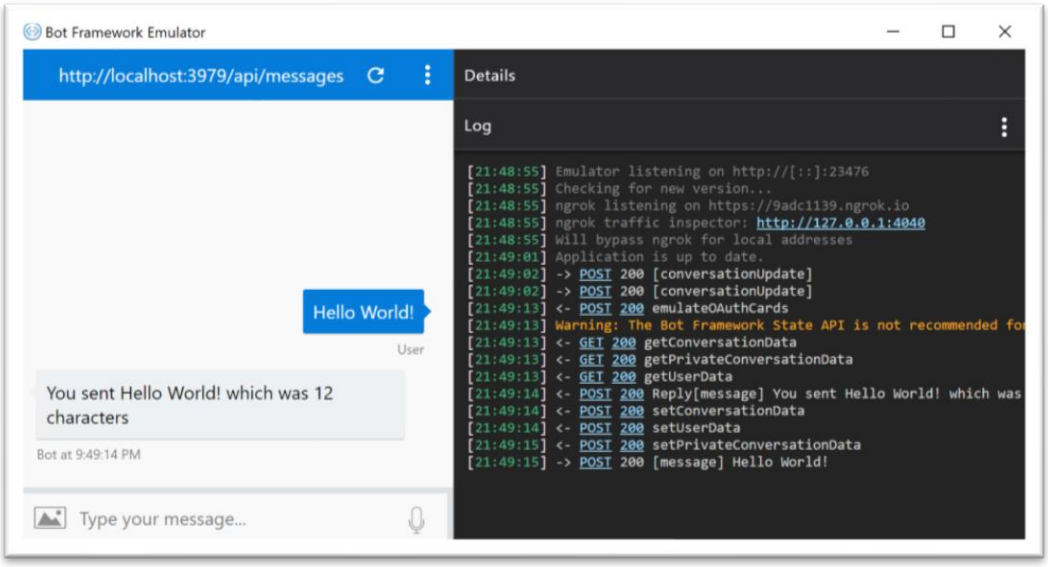

Es wird empfohlen, die referenzierten Nuget-Pakete zu aktualisieren. Lassen Sie die Einstellung "Include preresease" inaktiv – dieser Kurs verwendet Bot Framework API v3.




Sie können das Projekt nun kompilieren und ausführen. Starten Sie den Bot Framework Emulator und geben Sie http://localhost:3979/api/messages als Adresse und en-US oder de-DE als Locale ein. Lassen Sie Microsoft App ID und Microsoft App Password leer:



Ihr neues Bot soll auf die Angabe reagieren:




## 2.2 Hinzufügen eines neuen Dialogs

Wir werden nun die Funktionalität des Bots um einen weiteren Dialog ergänzen, in dessen Rahmen der Benutzer eine zufällige Zahl erraten soll, die beim Erstellen einer Dialoginstanz festgelegt wurde.

Fügen Sie eine neue Klasse unter dem Projektordner "Dialogs" namens *GuessTheNumberDialog* hinzu. Markieren Sie die Klasse mit dem *Serializable*-Attribut und geben Sie *IDialog<int>* (namespace Microsoft.Bot.Builder.Dialogs) als die zu implementierende Schnittstelle an. Der Typ *int* bezieht sich auf den Rückgabewert des Dialogs, in unserem Fall beinhaltet er die Anzahl der Versuche, die der Benutzer zum Erraten der Zahl benötigt hat – diese stellt *GuessTheNumberDialog* seinem Aufrufer zur Verfügung.

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

namespace BotInADay_ConversationFlow.Dialogs
{
    [Serializable]
    public class GuessTheNumberDialog : IDialog<int>
    {
       ...
    }
}
```

2 Felder für die Zufallszahl und die Anzahl der Versuche sowie ein Konstruktor werden hinzugefügt:

```
...
private readonly int number = 0;
private int counter = 0;

public GuessTheNumberDialog(int number)
{
    this.number = number;
}
...
```

Nun gilt es, die IDialog<int>-Schnittstelle zu implementieren. Sie beinhaltet lediglich eine Methode - *StartAsync(Microsoft.Bot.Builder.Dialogs.IDialogContext)*.

```
...
private const string ValidationError = "It has to be a number between 1 and 100.";
...
public async Task StartAsync(IDialogContext context)
{
    PromptDialog.Number(context, ResponseReceivedAsync,
        "I have a number between 1 and 100 in mind. Can you guess it?",
        ValidationError, 3, null, 1, 100);
}
...
```

*PromptDialog.Number* übenimmt folgende Aufgaben: Präsentieren einer Aufforderung zur Eingabe, Validierung der Eingabe (Typ, Grenzwerte), Wiederholung des Vorgangs bei inkorrektem Wert und Aufruf des Delegates *ResponseReceivedAsync* bei Erfolg.
Den Delegate können wir als eigenständige Methode implementieren:

```
...
private async Task ResponseReceivedAsync(
      IDialogContext context,
      IAwaitable<long> result)
{
    var guess = await result;
    counter++;
    if (guess == number)
    {
        await context.PostAsync($"Exactly! Well done!");
        context.Done(counter);
        return;
    }

    var hint = number > guess ? "bigger" : "smaller";
    PromptDialog.Number(context, ResponseReceivedAsync,
        $"Not quite. Try a {hint} one.",
        ValidationError, 3, null, 1, 100);
}
...
```

Den vom Benutzer eingegebenen Wert erhält man aus dem IAwaitable<long>, sein Typ entspricht der verwendeten Methode *PromptDialog.Number*. Nach dem Vergleich des eingegebenen Werts mit der Zufallszahl wird – im positiven Fall - die Steuerung zusammen mit

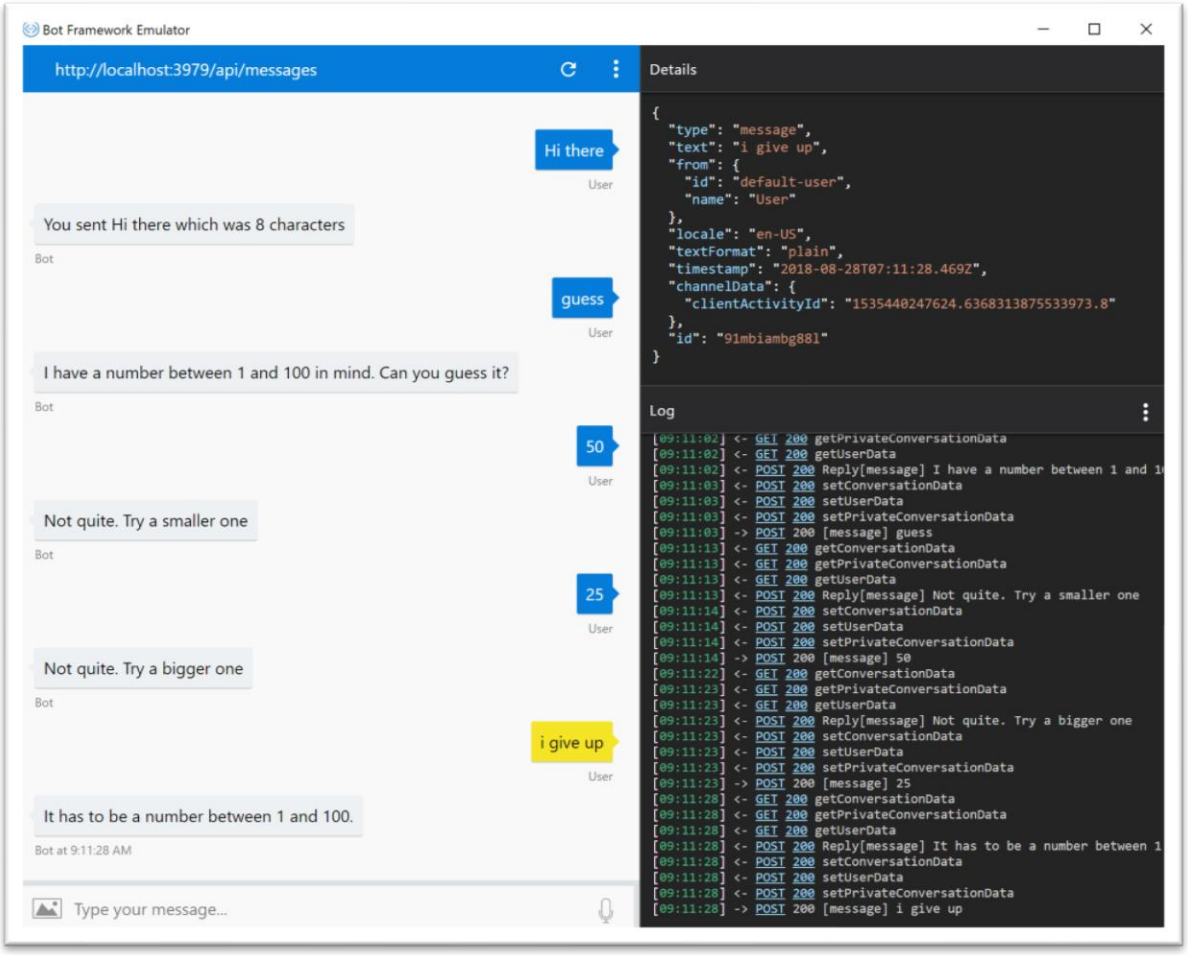
der gespeicherten Anzahl von Versuchen an den aufrufenden Dialog weitergegeben, bei Ungleichheit wird die Aufforderung zur Eingabe einer Zahl mit einem Hinweis erzeugt.

Wir müssen nun für den Aufruf des neuen Dialogs sorgen. Er wird aus dem vorhandenen *RootDialog* initiiert:

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

            if ("guess".Equals(activity.Text))
            {
                var number = (new Random(Guid.NewGuid().GetHashCode())).Next(1, 101);
                context.Call(new GuessTheNumberDialog(number), Guessed);
                return;
            }
            ...
        }

        private async Task Guessed(IDialogContext context, IAwaitable<int> result)
        {
            var count = await result;
            await context.PostAsync($"It took you {count} rounds to guess!");
            context.Wait(MessageReceivedAsync);
        }
        ...
```

Die bestehende Methode *MessageReceivedAsync* wird um eine Überprüfung der Benutzereingaben ergänzt, bei "guess" wird die Steuerung an den neuen Dialog zusammen mit der vorab generierten Zufallszahl weitergegeben. Das Ergebnis wird an den Delegate *Guessed* geliefert, wo die entsprechende Benachrichtigung angezeigt wird.

Das Projekt kann nun kompiliert und ausgeführt werden. In Bots Framework Emulator wird der hinzugefügte Dialog durch die Eingabe von "guess".




Die eventuelle Fehlersuche kann durch die Verwendung eines Try-Catch-Blocks um die Conversation.SendAsync-Methode in der Klasse *MessagesController* vereinfacht werden.

```
...
try
{
	await Conversation.SendAsync(activity, () => new Dialogs.RootDialog());
}
catch (Exception e)
{
	Console.WriteLine(e);
	throw;
}
...
```

# 3 Persistente Dialogdaten

## 3.1 Nutzung des Zustandsspeichers

Wir wollen nun die letzten 2 Spielergebnisse vergleichen, um dem Benutzer eine Rückmeldung zu geben. *IDialogContext* erlaubt einen einfachen Zugriff auf die im Kontext des Benutzers und

der Unterhaltung gespeicherten Daten. In diesem Beispiel werden *IDialogContext.UserData.SetValue* und *IDialogContext.UserData.TryGetValue* verwendet, um den letzten Spielstand zu verwalten:

```
...
private async Task Guessed(IDialogContext context, IAwaitable<int> result)
{
    var count = await result;
    string statInfo = null;
    if (context.UserData.TryGetValue("guess_stats", out int recent))
    {
        int diff = count - recent;
        switch (Math.Sign(diff))
        {
            case -1:
                statInfo = $", {-diff} less than last time";
                break;
            case 1:
                statInfo = $", {diff} more than last time";
                break;
            case 0:
                statInfo = ", exactly like last time";
                break;
        }
    }

    context.UserData.SetValue("guess_stats", count);

    var msg = $"Good quess. It took you {count} rounds{statInfo}!";
    await context.PostAsync(msg);

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

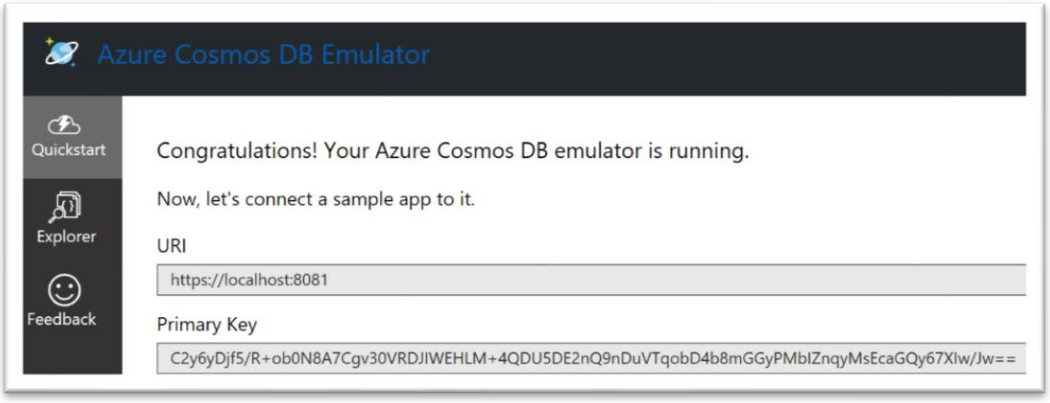
## 3.2 State Management mit CosmosDb

In der Produktionsumgebung soll man die Verwaltung der Dialogzustände sowie Benutzer- und Gesprächsdaten einem zuverlässigeren Dienst anvertrauen. Der Vertrag zwischen Bot Framework und dem Persistenzdienst ist in der Schnittstelle *IBotDataStore<BotData>* festgehalten, die bekanntesten realisierten Implementierungen greifen u.a. auf Azure Tables und Cosmos DB zurück.

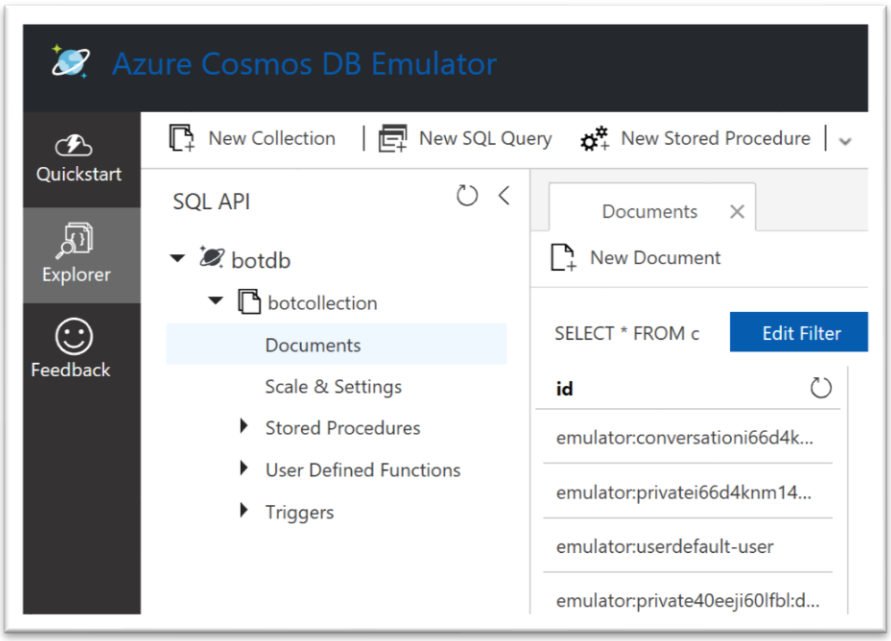
Unter https://docs.microsoft.com/en-us/azure/bot-service/dotnet/bot-builder-dotnet-state-azure-cosmosdb?view=azure-bot-service-3.0 erhält man ausführliche Informationen zur Nutzung von Cosmos DB als Bot Data Store. Die Konfiguration erfolgt in der Global.asax.cs:

```
        …
        protected void Application_Start()
        {
            var stateManager = ConfigurationManager.AppSettings["BotStateManager"];

            IbotDataStore<BotData> store = null;

            if ("DocumentDb".Equals(stateManager))
            {
                var uri = new Uri(ConfigurationManager.AppSettings["DocumentDbUrl"]);
                var key = ConfigurationManager.AppSettings["DocumentDbKey"];
                store = new DocumentDbBotDataStore(uri, key);
            }

            Conversation.UpdateContainer(
                builder =>
                {
                    if (store != null)
                    {
                        builder.Register(c => store)
                            .Keyed<IbotDataStore<BotData>>(AzureModule.Key_DataStore)
                            .AsSelf()
                            .SingleInstance();

                        builder.Register(c => new CachingBotDataStore(store,
CachingBotDataStoreConsistencyPolicy.EtagBasedConsistency))
                            .As<IBotDataStore<BotData>>()
                            .AsSelf()
                            .InstancePerLifetimeScope();
                    }
                });

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

Mit einem lokal installierten Azure Cosmos DB Emulator lässt sich die Funktionsweise von *DocumentDbBotDataStore* ausprobieren. Die Verbindungseinstellungen können von der Startseite des Emulators abgerufen und in die Web.config des Bot-Projekts übertragen werden.



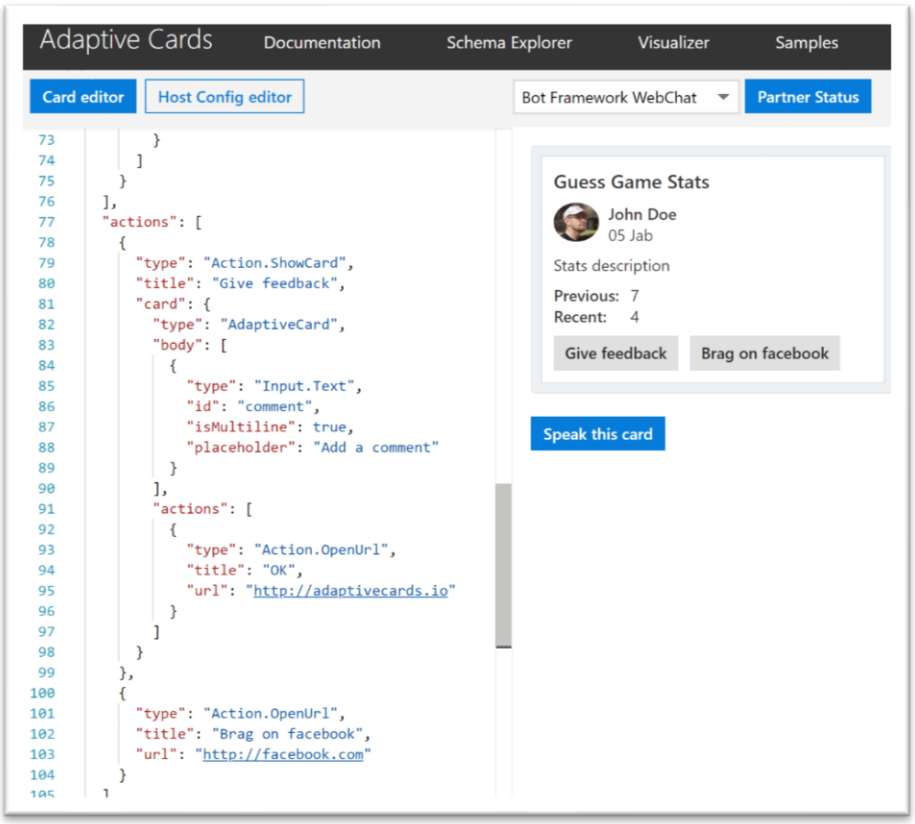
```
...
<add key="BotStateManager" value="DocumentDb" />
<add key="DocumentDbUrl" value="https://localhost:8081" />
<add key="DocumentDbKey"
value="C2y6yDjf5/R+ob0N8A7Cgv30VRDJIWEHLM+4QDU5DE2nQ9nDuVTqobD4b8mGGyPMbIZnqyMsEcaGQy67XIw
/Jw==" />
...
```




# 4 Adaptive Cards

## 4.1 Eigene Cards gestalten

Nach dem Spiel wollen wir dem Benutzer eine visuell ansprechende Vorstellung seines Ergebnisses liefern – Adaptive Cards eignen sich perfekt für diese Aufgabe. Unter http://adaptivecards.io/visualizer kann man eigene Cards erstellen und ihre optische Erscheinung auf verschiedenen Plattformen testen. Probieren Sie es aus!




## 4.2 Verwendung der Cards im Code

Laden Sie die Definition Ihrer Adaptive Card in Json-Formar herunter und legen Sie diese in Ihrem Bot-Projekt als statscard.json ab. Setzen Sie die *Build Action* auf *Embedded Ressource*.

Bevor wir die Platzhalter in der Adaptive Card mit realen werden überschreiben, muss die Karte deserializiert werden:

```
...
private static AdaptiveCard LoadCardTemplate()
{
    AdaptiveCard card;

    using (Stream stream = Assembly
        .GetExecutingAssembly()
        .GetManifestResourceStream("BotInADay_ConversationFlow.statscard.json"))
    using (var reader = new StreamReader(stream))
    {
        var json = reader.ReadToEnd();
        AdaptiveCardParseResult result = AdaptiveCard.FromJson(json);
        card = result.Card;
    }
    return card;
}
...
```

Die Abbildung der Card als DOM ermöglicht den Zugriff auf die einzelnen Elemente des Baums (hier wird nur ein Beispiel gegeben, die Struktur hängt von Ihrem Design ab):

```
...
private static AdaptiveCard GetGuessStatisticsCard(int actual, int last, string
description)
{
    var card = LoadCardTemplate();
    var timestampElement = ((card.Body[0] as AdaptiveCards.AdaptiveContainer).Items[1]
        as AdaptiveColumnSet)
        .Columns[1].Items[1] as AdaptiveTextBlock;

    timestampElement.Text = DateTime.Today.ToString("dd MMM");
    ...

    if (last == 0)
    {
        factSet.Facts.RemoveAt(0);
    }

    return card;
}
...
```

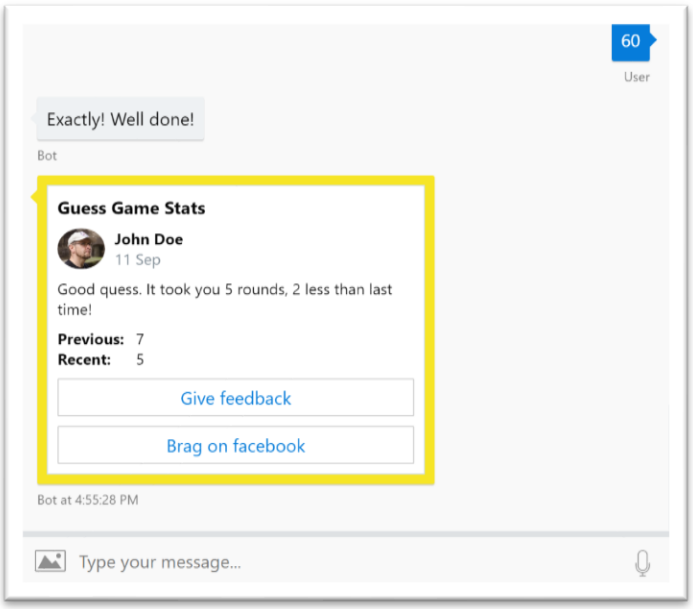
Adaptive Cards werden als Attachments einer Message übertragen:

```
...
private IMessageActivity GetAdaptiveCardMessage(IDialogContext context,
AdaptiveCards.AdaptiveCard card, string cardName)
{
    var message = context.MakeMessage();
    if (message.Attachments == null)
        message.Attachments = new List<Attachment>();
    var attachment = new Attachment()
    {
        Content = card,
        ContentType = "application/vnd.microsoft.card.adaptive",
        Name = cardName
    };
    message.Attachments.Add(attachment);
    return message;
}
...
```

Nun werden die neuen Methoden dafür verwendet, eine Instanz von *IMessageActivity* zu erhalten, die dann als Antwort zurückgeliefert wird:

```
...
private async Task Guessed(IDialogContext context, IAwaitable<int> result)
{

    ...
    var stats = GetGuessStatisticsCard(count, recent, msg);
    var message = GetAdaptiveCardMessage(context, stats, "Guess Game Stats");
    await context.PostAsync(message);
    ...
}
...
```

Und so könnte Ihr Ergebnis aussehen:



# 5 Ideen für die nächsten Schritte

## 5.1 Einbindung von Cognitive Services

Die Benutzereingabe lässt sich einer semantischen Analyse unterziehen, um die Ergebnisse für die Steuerung des Dialogverlaufs verwenden zu können. Neben LUIS, mit dem der Text im Hinblick auf die darin verborgene Absicht analysieren lässt, können sich weitere Services als nützlich erweisen:

- Extraktion von Schlüsselbegriffen: wie ist der Text thematisch eingeordnet?
- Sentiment Analysis: wie ist der Anwender drauf?
- Search: welcher bereits vorhandener Inhalt könnte die Fragestellung des Benutzers optimal adressieren?

## 5.2 FormFlow

FormFlow bietet eine dialoggestützte Erhebung von Informationen, gewissermassen ein Gegenstück zu Formularen aus der Welt von Desktop- und Webanwendungen. Die Aufgabe des Entwicklers besteht in der Definition des auszufüllenden Modells und ihre Ergänzung mit Hinweisen und Validierungsregeln, FormFlow übernimmt die schrittwese Dialogführung, Interpretation der Eingaben und Fallback-Szenarien in Fehlerfällen. Das Beispielprojekt bietet eine alternative Implementierung des Guess-Dialogs mit Hilfe von FormFlow. Weitere Informationen dazu finden Sie unter unter https://docs.microsoft.com/en-us/azure/bot-service/dotnet/bot-builder-dotnet-formflow?view=azure-bot-service-3.0.

## 5.3 Scorable Dialogs

Mit Hilfe von Scorable Dialogs können Interaktionsdesigner auf die durch den aktuellen Dialog aus dem Stack nicht abgedeckte Situationen reagieren (Entscheidung zum Abbrechen des Prozesses, Themen- bzw. Kontextwechsel, Aufruf von Hilfefunktion, Bitte um Weiterleitung des Gesprächs zu einem Support-Mitarbeiter etc.). Die registrierten *IScorable*-Instanzen erhalten die Benutzereingabe und können mit der Rückgabe eines Wertes im Bereich von 0 bis 1 entscheiden, wie relevant sie diese für ihren Zuständigkeitsbereich einschätzen. Der Scorable mit dem grössten Ergebnis kann den Dialog-Stack beeinflussen (z.B. einen Hilfe-Dialog aufrufen);