isolutions

Hands-on Lab

Bot in a Day

isolutions

Inhalt

1 Einrichtung der Entwicklungsumgebung

Ihre Entwicklungsumgebung soll folgende Voraussetzungen erfüllen:

- Windows 10
- .NET Core 2.1
- Visual Studio 2017
- GIT-Client

1.1 Installation der Komponenten für Bot-Entwicklung

Folgende Komponenten müssen zusätzlich installiert werden:

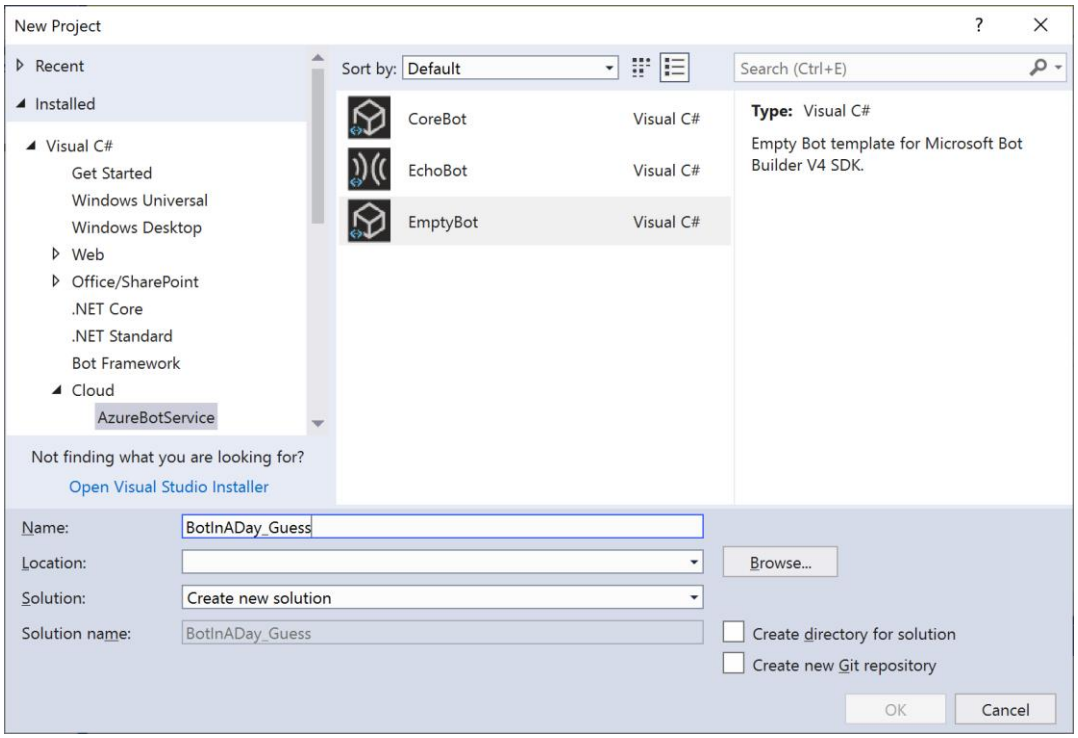
"Bot Application" Projektvorlage für Visual Studio:
https://marketplace.visualstudio.com/items?itemName=BotBuilder.botbuilderv4

Bot Framework Emulator:
https://github.com/Microsoft/BotFramework-Emulator/releases/download/v4.3.2/BotFramework-Emulator-4.3.2-windows-setup.exe

2 HandsOn – Guess the Number Bot

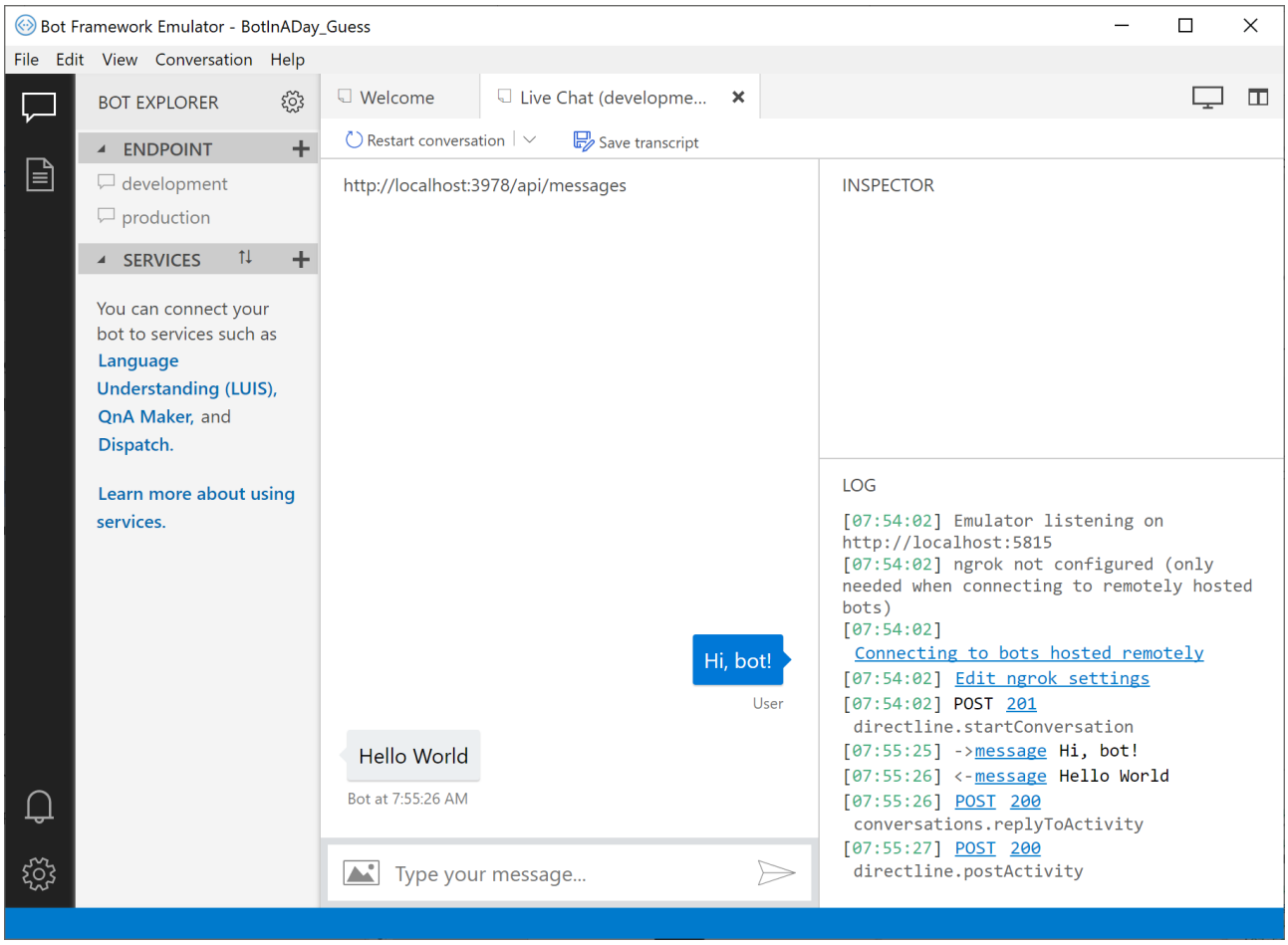
2.1 Erstellen eines neuen Bot-Projekts

Erstellen Sie ein neues Projekt namens BotInADay_Guess aus der "EmptyBot "-Vorlage in Visual Studio 2017:



Es wird empfohlen, die referenzierten Nuget-Pakete zu aktualisieren. Behalten Sie jedoch die .NET Core Version bei 2.1.

Sie können das Projekt nun kompilieren und ausführen. Starten Sie den Bot Framework Emulator (V4) und öffnen Sie die in dem Bot-Projekt vorhandene .bot-Datei. Der Emulator stellt die Verbindung zum Bot-Endpunkt und startet die Konversation.



Die in der Vorlage enthaltene Funktionalität reicht aus, um jede Nachricht von der Anwenderseite mit «Hello World!» zu beantworten.

2.2 Identifizieren der Kernelemente einer Bot-Lösung

Ihr neues Bot ist in der ersten Linie ein .NET Core Web-Projekt, somit enthält es

- Program.cs mit dem erstellen und ausführen eines WebHosts.
- Startup.cs mit der Konfiguration der Dienste und der Request-Pipeline. Das Bot-Framework bietet hier Extension-Methods, um Bots als Dienste zu konfigurieren.
- appsettings.json – die für .NET Core Anwendungen übliche Konfigurationsdatei enthält lediglich einen Verweis auf die .bot-Datei.

Die Konfigurationsdatei von Bot-Lösungen enthält Einstellungen für Bot-Komponenten in verschiedenen Konfigurationen und wird sowohl von dem Programmcode als auch von Bereitstellung- und Testtools wie Bot Framework Emulator verwendet. Bei einem neuen Bot sind darin Service Endpoints für Test und Produktion definiert. Die Nutzung dieser Konfigurationsoption ist nicht zwingend vorgeschrieben, mit ihrer Hilfe werden jedoch viele Aufgaben erheblich erleichtert.

> Die Referenzlösung zu diesem Workshop können Sie von Github über
> `git clone https://github.com/mc5eamus/BotInADay_HandsOn_V4.git`
> erhalten.

2.3 Hinzufügen eines neuen (Spiel-)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.

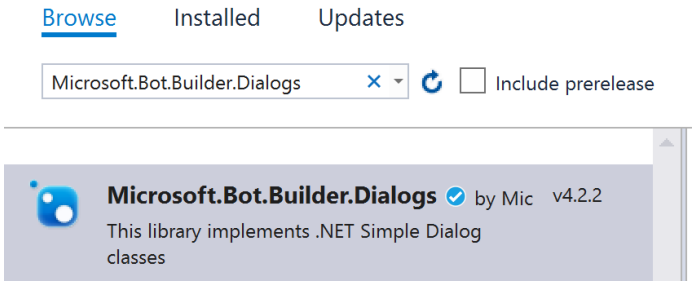
> Das Ergebnis dieses Kapitels trägt den Tag v1 in der Repository. Sie können es über
> `cd BotInADay_HandsOn_V4`
> `git checkout v1`
> abrufen.

Legen Sie im Projekt einen Ordner «Dialogs» an. Um den Zustand des Dialogs zu verwalten, legen Sie in dem neuen Ordner die Klasse *GuessState* an:

Dialogs/GuessState.cs

```
namespace BotInADay_Guess.Dialogs
{
    public class GuessState
    {
        // Random number
        public int Number { get; set; }
        // Number of turns user took to guess so far
        public int Counter { get; set; }
        // Personal record
        public int BestScore { get; set; }
    }
}
```

In der eigentlichen Dialogklasse machen wir von der Dialogs-Bibliothek des Bot Framework gebrauch. Öffnen Sie den NuGet Package Manager und installieren Sie das Paket *Microsoft.Bot.Builder.Dialogs*.



Legen Sie in dem Dialogs-Ordner eine neue Klasse *GuessDialog* an:

Dialogs/GuessDialog.cs

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

```
using Microsoft.Bot.Builder.Dialogs;
using Microsoft.Bot.Schema;
using Microsoft.Extensions.Logging;

namespace BotInADay_Guess.Dialogs
{
    public class GuessDialog : ComponentDialog
    {
        private const string NumberPrompt = "numberPrompt";
        private const string GuessFlow = "guessFlow";
        private readonly IStatePropertyAccessor<GuessState> guessStateAccessor;
        private readonly ILogger log;

        public GuessDialog(
            IStatePropertyAccessor<GuessState> guessStateAccessor,
            ILoggerFactory loggerFactory) : base(nameof(GuessDialog))
        {
            this.guessStateAccessor = guessStateAccessor;
            log = loggerFactory.CreateLogger<GuessDialog>();
            var waterfallSteps = new WaterfallStep[]
            {
                InitializeStateStepAsync,
                PromptForNumberStepAsync,
                EvaluateResultStepAsync
            };
            AddDialog(new WaterfallDialog(GuessFlow, waterfallSteps));
            AddDialog(new NumberPrompt<int>(NumberPrompt, ValidateGuess));
        }


        private async Task<DialogTurnResult> InitializeStateStepAsync(WaterfallStepContext
stepContext, CancellationToken cancellationToken)
        {
            var rnd = new Random();
            var state = await guessStateAccessor.GetAsync(
                stepContext.Context,
                () => new GuessState(),
                cancellationToken);
            state.Number = rnd.Next(1, 100);
            state.Counter = 0;
            log.LogInformation($"New round. Number is {state.Number}");
            await guessStateAccessor.SetAsync(stepContext.Context,
                state,
                cancellationToken);
            await stepContext.Context.SendActivityAsync("I have a number between 1 and 99 in
mind.", cancellationToken: cancellationToken);
            return await stepContext.NextAsync(cancellationToken: cancellationToken);
        }


        private async Task<DialogTurnResult> PromptForNumberStepAsync(
            WaterfallStepContext stepContext,
            CancellationToken cancellationToken)
        {
            var opts = new PromptOptions
            {
                Prompt = new Activity
                {
                    Type = ActivityTypes.Message,
```

```
                Text = "What's your guess?",
            },
        };
        return await stepContext.PromptAsync(NumberPrompt,
            opts,
            cancellationToken);
    }


    private async Task<DialogTurnResult> EvaluateResultStepAsync(
        WaterfallStepContext stepContext,
        CancellationToken cancellationToken)
    {
        var newBestScore = ".";
        var state = await guessStateAccessor.GetAsync(stepContext.Context,
cancellationToken: cancellationToken);
        var counter = state.Counter;
        if (state.BestScore == 0 || state.Counter < state.BestScore)
        {
            if (state.BestScore != 0)
            {
                newBestScore = $", it's {state.BestScore - state.Counter} better than
your previous best score!";
            }
            state.BestScore = state.Counter;
        }
        state.Counter = 0;
        await guessStateAccessor.SetAsync(stepContext.Context,
            state,
            cancellationToken);

        await stepContext.Context.SendActivityAsync($"Exactly! It took you {counter}
turns{newBestScore}", cancellationToken: cancellationToken);

        return await stepContext.EndDialogAsync(cancellationToken: cancellationToken);
    }


    private async Task<bool> ValidateGuess(PromptValidatorContext<int> promptContext,
        CancellationToken cancellationToken)
    {
        var state = await this.guessStateAccessor.GetAsync(promptContext.Context,
cancellationToken: cancellationToken);
        state.Counter++;
        await guessStateAccessor.SetAsync(promptContext.Context,
            state,
            cancellationToken);
        var value = promptContext.Recognized.Value;
        if (value <= 0 || value >= 100)
        {
            await promptContext.Context.SendActivityAsync("I'm pretty certain it's a
number between 1 and 99. Give it another try!", cancellationToken: cancellationToken);
            return false;
        }

        if (value < state.Number || value > state.Number)
        {
            var hint = value < state.Number ? "bigger" : "smaller";
```

```
            await promptContext.Context.SendActivityAsync($"My number is {hint}, try
again!", cancellationToken: cancellationToken);
            return false;
         }
         return true;
      }
   }
}
```

In dem Dialog-Code werden die sog. Property Accessors für den Zustand des Dialogs verwendet. Wir müssen in der Konfiguration der Services in Startup.cs festlegen, wie Unterhaltungs- und Benutzer-State behandelt werden:

Startup.cs

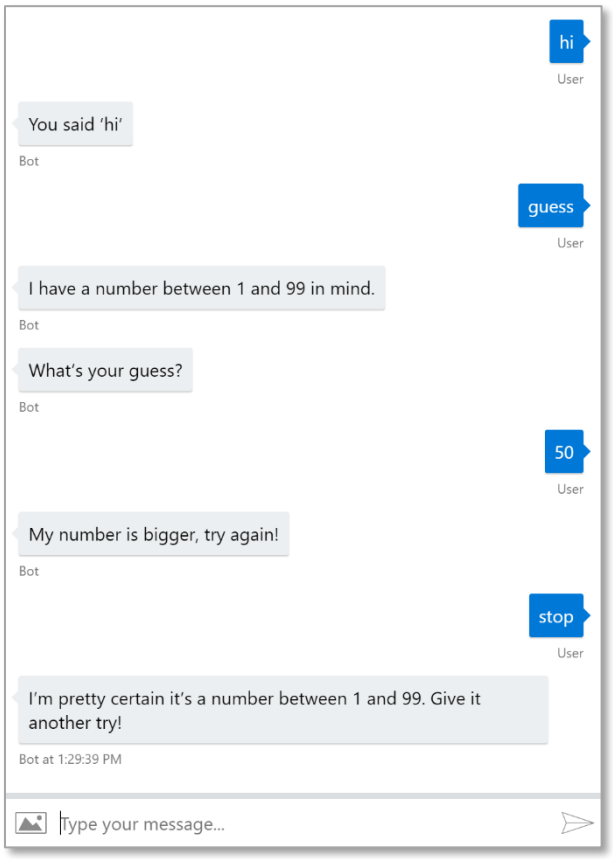
```
...
using Microsoft.Bot.Builder;
...

   public void ConfigureServices(IServiceCollection services)
   {
      IStorage dataStore = new MemoryStorage(); // For development purposes only

      var conversationState = new ConversationState(dataStore);
      services.AddSingleton(conversationState);

      var userState = new UserState(dataStore);
      services.AddSingleton(userState);

      ...
   }
}
```

Im letzten Schritt müssen wir dafür sorgen, dass der neue Dialog aufgerufen wird. In der *BotInADay_GuessBot.cs* wird der Message-Händler auf folgende Weise ergänzt:

BotInADay_GuessBot.cs

```
   public class BotInADay_GuessBot : IBot
   {
      private readonly UserState userState;
      private readonly ConversationState conversationState;
      private readonly IStatePropertyAccessor<GuessState> guessStateAccessor;
      private readonly IStatePropertyAccessor<DialogState> dialogStateAccessor;
      private readonly DialogSet dialogs;

      public BotInADay_GuessBot(
         UserState userState,
         ConversationState conversationState,
         ILoggerFactory loggerFactory)
      {
         this.userState = userState;
         this.conversationState = conversationState;
         guessStateAccessor = userState.CreateProperty<GuessState>(nameof(GuessState));
         dialogStateAccessor =
conversationState.CreateProperty<DialogState>(nameof(DialogState));
         dialogs = new DialogSet(dialogStateAccessor);
```

```
            dialogs.Add(new GuessDialog(guessStateAccessor, loggerFactory));
        }

        public async Task OnTurnAsync(
            ITurnContext turnContext,
            CancellationToken cancellationToken = default(CancellationToken))
        {
            // Create a dialog context since we're using dialogs
            var dc = await dialogs.CreateContextAsync(turnContext, cancellationToken);
            // if there's a current dialog, let it have control and process the input
            var dialogResult = await dc.ContinueDialogAsync(cancellationToken);

            if (turnContext.Activity.Type == ActivityTypes.Message)
            {
                // user sent us a message.
                if (!dc.Context.Responded) // did we send a response yet?
                {
                    // examine results from active dialog to decide how to act
                    switch (dialogResult.Status)
                    {
                        case DialogTurnStatus.Empty:
                            // there's nothing on the dialog stack
                            // so we must feel responsible for the communication
                            if ("guess" == turnContext.Activity.Text)
                            {
                                await dc.BeginDialogAsync(nameof(GuessDialog),
cancellationToken: cancellationToken);
                            }
                            else
                            {
                                await turnContext.SendActivityAsync($"You said
'{turnContext.Activity.Text}'", cancellationToken: cancellationToken);
                            }
                            break;

                        case DialogTurnStatus.Complete:
                            await dc.EndDialogAsync(cancellationToken: cancellationToken);
                            break;

                        case DialogTurnStatus.Waiting:
                            break;

                        case DialogTurnStatus.Cancelled:
                            break;

                        default:
                            await dc.CancelAllDialogsAsync(cancellationToken);
                            break;
                    }
                }
            }
            await conversationState.SaveChangesAsync(
                    turnContext, cancellationToken: cancellationToken);
            await userState.SaveChangesAsync(
                    turnContext, cancellationToken: cancellationToken);
        }
    }
```



Unser neues Bot kann nun getestet werden! Mit der Eingabe von «guess» wird der neue Dialog gestartet. Allerdings ist die Interaktion noch etwas geradlinig – das Bot hat wenig Empathie und versteht nicht, wenn der Anwender keine Lust mehr zum Spielen hat und aufhören bzw. aufgeben will.

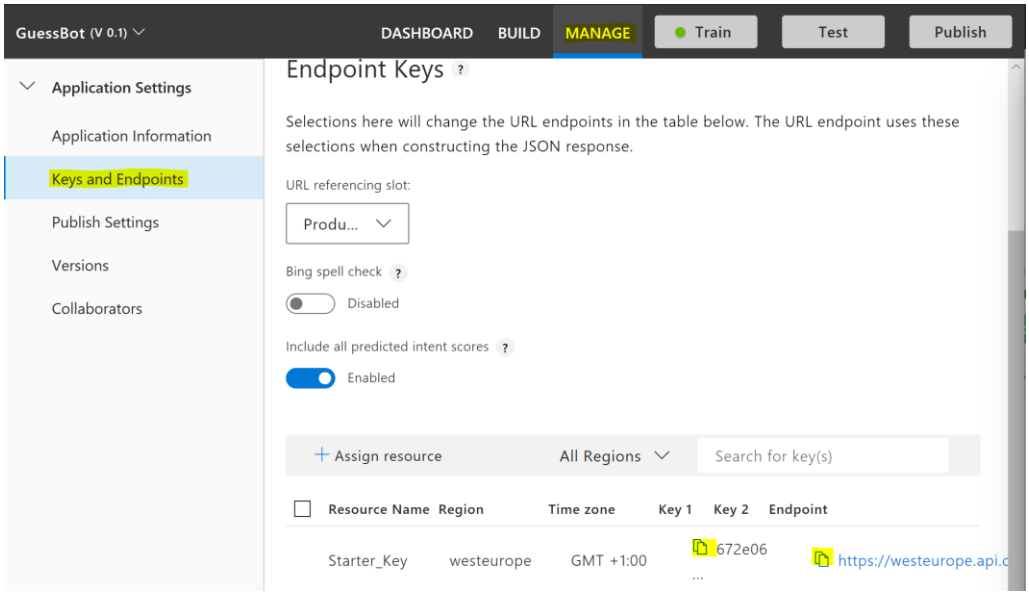
Das werden wir nun ändern.

2.4 Enter: Luis

LUIS – Language Unterstanding Intelligent Service – ist ein Dienst für die Verarbeitung von natürlichen Sprachen (NLP, Natural Language Processing). Die Nutzung in einem für die Entwicklungszwecke relevanten Ausmass ist kostenlos.

> Das Ergebnis dieses Kapitels trägt den Tag v2 in der Repository. Sie können es über
> `git checkout v2`
> abrufen.

Unter https://eu.luis.ai kann ein neues Modell erstellt werden, welches zunächst 3 Intents unterstützt: Game, Cancel und None. Experimentieren Sie mit diversen Äusserungsformen und publizieren Sie das fertige Modell. Endpoint-Information aus den Bot-Einstellungen werden in die BotInADay_Guess.bot übertragen.



Authoring Key kann über die Einstellungen des persönlichen Profils in LUIS (Click auf den Namen oben rechts) ermittelt werden.

BotInADay_Guess.bot als zentrale Anlaufstelle für alle Konfigurationsinformationen wird mit einem neuen Block für den Luis-Service ergänzt:

BotInADay_Guess.bot

```
{
    "type": "luis",
    "name": "basic-bot-LUIS",
    "region": "westeurope",
    "appId": "<Your LUIS App Key>",
    "authoringKey": "<Your LUIS Authoring Key>",
    "version": "0.1",
    "id": "basic-bot-LUIS"
}
```

Der Zugriff auf LUIS wird mit Hilfe der Bibliothek *Microsoft.Bot.Builder.AI.Luis* gewährleistet – installieren Sie das entsprechende Nuget-Packet. Um den Zugriff auf die Services wie LUIS zu vereinfachen, erstellen wir eine *BotServices*-Klasse, die als Factory für die Service-Instanzen gilt:

BotServices.cs

```
using Microsoft.Bot.Builder.AI.Luis;
using Microsoft.Bot.Configuration;

namespace BotInADay_Guess
{
    public class BotServices
    {
        public LuisRecognizer Luis { get; }

        public BotServices(BotConfiguration config)
        {
            var luis = config.Services.Find(_ => ServiceTypes.Luis.Equals(_.Type)) as
LuisService;
            var app = new LuisApplication(luis.AppId, luis.AuthoringKey,
luis.GetEndpoint());
            Luis = new LuisRecognizer(app);
        }
    }
}
```

Instanzen der neuen Klasse werden per Dependency Injection dem Bot zur Verfügung gestellt. Für diese Zwecke wird der relevante Teil der Konfiguration an die Stelle vor dem *services.AddBot*<..>-Aufruf übertragen und um die Konfiguration von *BotServices* als Singleton ergänzt:

Startup.cs

```
    ...

    var secretKey = Configuration.GetSection("botFileSecret")?.Value;

    // Loads .bot configuration file and adds a singleton that your Bot can access through
dependency injection.
    var botConfig = BotConfiguration.Load(@".\BotInADay_Guess.bot", secretKey);
    services.AddSingleton(sp => botConfig);
    services.AddSingleton(sp => new BotServices(botConfig));

    services.AddBot<BotInADay_GuessBot>(options =>
    {
        ...
```

Die neue Factory kann nun in dem Constructor des Bots verwendet werden. Eine weitere Methode GetTopIntent soll den Aufruf von LUIS übernehmen und den Intent mit der grössten Wahrscheinlichkeit zurückgeben:

BotInADay_GuessBot.cs

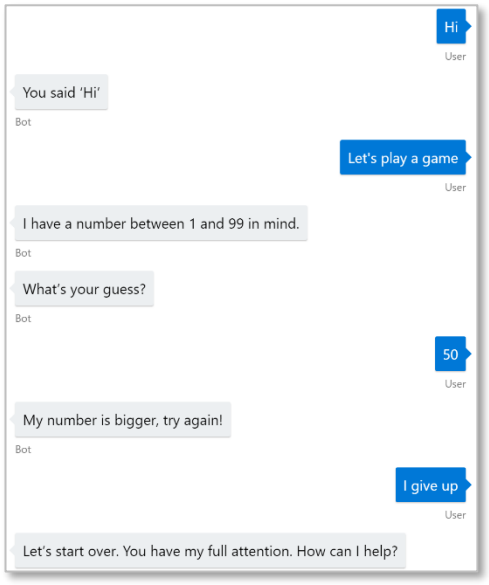
```
public class BotInADay_GuessBot : IBot
{
    ...
    private readonly BotServices services;

    public BotInADay_GuessBot(
        BotServices services,
        UserState userState,
        ConversationState conversationState,
        ILoggerFactory loggerFactory)
    {
        ...
        this.services = services;
        ...
    }

    ...
    private async Task<string> GetTopIntent(
            CancellationToken cancellationToken, ITurnContext ctx)
    {
        if (string.IsNullOrEmpty(ctx.Activity.Text))
        {
            return null;
        }

        var luisResults = await services.Luis.RecognizeAsync(ctx, cancellationToken);
        var topScoringIntent = luisResults?.GetTopScoringIntent();
        return topScoringIntent?.intent;
    }
```

Um den Service zu nutzen, wird die vom Anwender erhaltene Nachricht nun mit Hilfe von LUIS analysiert und ausgewertet:

BotInADay_GuessBot.cs

```
public async Task OnTurnAsync(
      ITurnContext turnContext,
      CancellationToken cancellationToken = default(CancellationToken))
{
   // Create a dialog context since we're using dialogs
   var dc = await dialogs.CreateContextAsync(turnContext, cancellationToken);

   if (turnContext.Activity.Type == ActivityTypes.Message)
   {
      var topIntent = await GetTopIntent(cancellationToken, turnContext);

      if ("Cancel".Equals(topIntent))
      {
         await dc.CancelAllDialogsAsync(cancellationToken);
         await turnContext.SendActivityAsync("Let's start over. You have my full
attention. How can I help?", cancellationToken: cancellationToken);
      }

      // if there's a current dialog, let it have control and process the input
      var dialogResult = await dc.ContinueDialogAsync(cancellationToken);
```

```
      // user sent us a message.
      if (!dc.Context.Responded) // did we send a response yet?
      {
         // examine results from active dialog to decide how to act
         switch (dialogResult.Status)
         {
            case DialogTurnStatus.Empty:
               // there's nothing on the dialog stack
               // so we must feel responsible for the communication
               if ("Game" == topIntent)
               {
                  await dc.BeginDialogAsync(nameof(GuessDialog), cancellationToken:
cancellationToken);
               }
               ...
```



Jetzt können wir unser Projekt ausführen. In dem neuen Szenario erscheint unser Bot bereits etwas intelligenter. Es erkennt, ob jemand spielen und aufhören will.

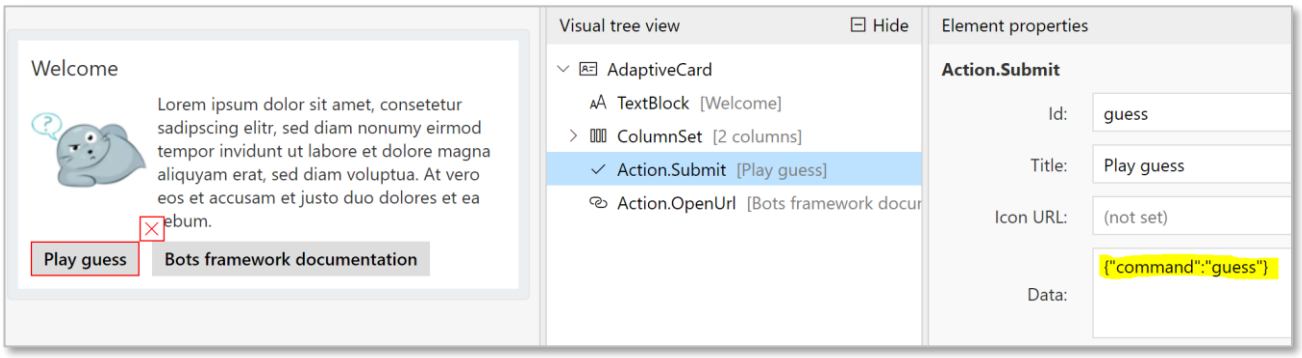
2.5 A warm welcome

> Das Ergebnis dieses Kapitels trägt den Tag v3 in der Repository. Sie können es über
> `git checkout v3`
> abrufen.

In dem Abschnitt werden wir Adaptive Cards nutzen, um den Benutzer zu begrüssen und ihm eine Auswahl an Handlungsoptionen zu bieten.

Das Definitionsformat von Adaptive Cards ist JSON, und obwohl wir einfache Cards direkt erstellen können, verwenden wir in diesem Teil des Tutorials einen Online-Editor: https://adaptivecards.io/designer/

Sie können die Karte nach Ihrem Geschmack gestalten, wir brauchen aber einen Button, mit dem der Spieldialog aufgerufen werden kann:



Um die Bearbeitung des Events vom Bot zu gewährleisten, werden die Button-Aktionsdaten auf *{"command":"guess"}* festgelegt.

Das Ergebnis wird als *MenuCard.json* im Dialogs-Ordner des Projekts gespeichert und seine Build Action auf *«Embedded resource»* gesetzt.

Medien, zu denen Adaptive Cards zählen, werden im Bot Framework als Attachments übermittelt. Mit den folgenden 2 Methoden in der Bot-Klasse wird aus der gespeicherten Adaptive Card JSON-Datei ein Attachment erzeugt und eine Willkommensnachricht an den Anwender gesendet.

BotInADay_GuessBot.cs

```
        private async Task WelcomeUser(
            CancellationToken cancellationToken,
            Activity activity,
            DialogContext dc)
        {
            var menuCard = CreateMenuAttachment();
```

```
            var response = activity.CreateReply();
            response.Attachments = new List<Attachment>() {menuCard};
            await dc.Context.SendActivityAsync(response, cancellationToken);
        }

        private Attachment CreateMenuAttachment()
        {
            var assembly = typeof(BotInADay_GuessBot).GetTypeInfo().Assembly;
            string adaptiveCard = null;
            using (var reader = new
StreamReader(assembly.GetManifestResourceStream("BotInADay_Guess.Dialogs.MenuCard.json")))
            {
                adaptiveCard = reader.ReadToEnd();
            }

            return new Attachment()
            {
                ContentType = "application/vnd.microsoft.card.adaptive",
                Content = JsonConvert.DeserializeObject(adaptiveCard),
            };
        }
```

Die Begrüssung soll ausgelöst werden, wenn ein Anwender eine neue Unterhaltung mit dem Bot startet – dieses Ereignis wird als Aktivität vom Typ *ActivityTypes.ConversationUpdate* – als Alternative zu *ActivityTypes.Message* – empfangen.

BotInADay_GuessBot.cs

```
    public async Task OnTurnAsync(
        ITurnContext turnContext,
        CancellationToken cancellationToken = default(CancellationToken))
    {
        ...

        if (turnContext.Activity.Type == ActivityTypes.Message)
        {
            ...
        }
        else if (turnContext.Activity.Type == ActivityTypes.ConversationUpdate)
        {
            if (turnContext.Activity.MembersAdded != null)
            {
                // Iterate over all new members added to the conversation.
                foreach (var member in turnContext.Activity.MembersAdded)
                {
                    // Greet anyone that was not the target (recipient) of this message.
                    if (member.Id != turnContext.Activity.Recipient.Id)
                    {
                        await WelcomeUser(cancellationToken, turnContext.Activity, dc);
                    }
                }
            }
        }
        ...
    }
```

Die Aktivierung des *«Play guess»*-Buttons soll zum gleichen Ergebnis führen, wie das Erkennen des Intents *«Game»* hinter der Texteingabe. Die Eigenschaft Value der jeweiligen Aktivität ist ein *JObject*, welches sich mit den entsprechenden Mitteln abfragen lässt:

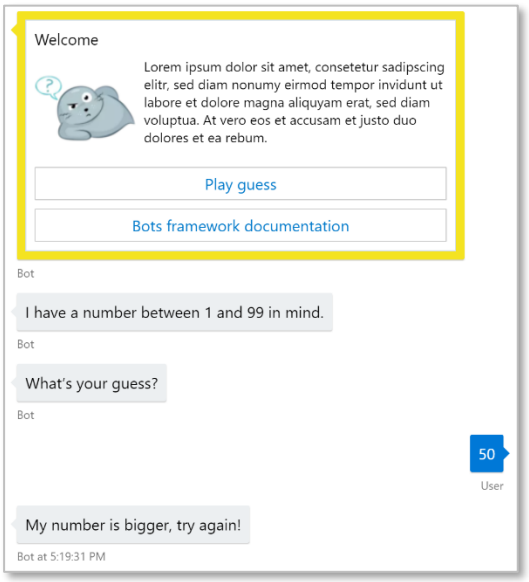
BotInADay_GuessBot.cs

```
if ("Game" == topIntent ||
      "guess" == (turnContext.Activity.Value as
      JObject)?.SelectToken("$..command")?.Value<string>())
{
   ...
```

Zusätzlich können wir die Welcome-Card anzeigen, wenn das Spiel beendet wird:

BotInADay_GuessBot.cs

```
var dialogResult = await dc.ContinueDialogAsync(cancellationToken);

if (dialogResult.Status == DialogTurnStatus.Complete)
{
   await WelcomeUser(cancellationToken, turnContext.Activity, dc);
}
```



Es ist wieder an der Zeit, das Projekt zu testen.

3 Ideen für die nächsten Schritte

3.1 Publizieren des Bot-Projekts auf Azure

Wenn Sie über eine Azure-Subscription verfügen, können Sie Ihr Projekt auf Azure publizieren. Eine ausführliche Beschreibung ist unter https://docs.microsoft.com/en-us/azure/bot-service/bot-builder-deploy-az-cli?view=azure-bot-service-4.0 zu finden.

3.2 Verwendung einer produktionstauglichen State Management Lösung

Für die Demonstrationszwecke haben wir die *MemoryStorage* verwendet, für den produktiven Einsatz benötigt ein Bot einen dauerhafteren Speicher. Microsoft bietet Bibliotheken für die Verwendung von Table Storage, CosmosDB und SQL Server, Sie können aber auch andere Speichermechanismen wie Redis Cache verwenden.

https://docs.microsoft.com/en-us/azure/bot-service/bot-builder-howto-v4-state?view=azure-bot-service-4.0

https://docs.microsoft.com/en-us/azure/bot-service/bot-builder-custom-storage?view=azure-bot-service-4.0

3.3 Telemetrie und Logging

Im Betrieb bieten die im Laufe der Nutzung entstehenden Informationen einen grossen Mehrwert für die Stabilisierung und Weiterentwicklung des Bots. Entwickler haben mehrere Möglichkeiten, Telemetrie-Daten zu sammeln und auszuwerten:

https://docs.microsoft.com/en-us/azure/bot-service/bot-builder-telemetry?view=azure-bot-service-4.0