Bonus Chapter 4

Bots and Cognitive Services

WHAT'S IN THIS CHAPTER?

- Defining bots
- Creating dialog bots
- Using form flow
- Understanding users with LUIS

WROX.COM CODE DOWNLOADS FOR THIS CHAPTER

The wrox.com code downloads for this chapter are found at `www.wrox.com` on the Download Code tab. The source code is also available at `https://github.com/ProfessionalCSharp/ProfessionalCSharp7` in the directory `BotsAndCognitive`.

The code for this chapter is divided into the following major examples:

- SimpleBot
- LuisBot

WHAT IS A BOT?

A bot is a software agent that acts for a user or other programs. In the past when you needed support for a product, but you weren't ready to get in contact with a real person, you had to read an FAQ document. If that didn't work, you had to try to get in contact with a person on a phone hotline or a chat window. Depending on your support agreement, you might have needed to wait in a queue for several minutes. (I also had an experience with waiting hours instead of minutes.) Now, you can ask a bot a question, and the bot can answer like a real person. If the bot doesn't know the answer, the task can be forwarded to a real person. The bot can learn. When the real person provides results, the bot retains the information for the next time a customer asks that same question.

A bot can not only be used to answer support requests. Bots can offer information, take orders, help the user while the user is working in a program, create tweets on Twitter, chat with users, take user requests to do actions like playing music, add information to a calendar, and more. Bots can also alert the user, such as sending a notification that to be on time for a calendar event that the user should leave early from the current location because of high traffic or a delayed train.

Bots can be implemented in your app, but you can also use other channels where your bots can be activated. For example, bots can be used within Skype, Facebook, Microsoft Teams, Slack, and many other channels.

You can implement ways for the user to communicate with bots by entering free-form text, using controls to select from lists, clicking buttons, and communicating via speech.

If you already have Web APIs (covered in Chapter 32, "Web API"), you can use them from your apps and your bots. With some scenarios, bots might be the main use for your Web APIs.

This chapter makes use of Microsoft Azure—to create and use bots as well as to use Azure Cognitive services. If you don't already have an Azure account, you can start with a 30-day free trial at `https://azure.microsoft.com/free/`. After that time, several services remain free (like shared website hosting with an `azurewebsites.net` domain), but you can switch any time to a paid model. What does it cost after the free time? Bot services can be hosted using App Services where you pay for a specific plan that includes a certain amount of CPU, memory, and disk sizes, or you can use a consumption-based plan where you pay for the number of calls used. Depending on the load and usage of your bots, one or the other variant can be the best, but the implementation is nearly the same. Using the consumption plan you're more restricted on the memory needed, and there's a maximum time your service may run with every invocation.

Let's start creating a simple bot before we get into more complex scenarios.

CREATING A DIALOG BOT

You can create a bot directly in the Microsoft Azure portal `https://portal.azure.com`. In the Azure Marketplace, in the AI + Cognitive Services category you can find the Web Bot App and Functions Bot. With the Functions Bot, you can select either a Consumption Plan or an App Service Plan. With the Consumption Plan you pay for use, and with the App Service Plan you reserve an App Service with allocated CPU and memory. With a Web Bot App, you need an App Service Plan; paying based on consumption is not an option.

The Functions Bot is based on Azure Functions, which are covered in Chapter 32. The Functions Bot makes use of C# script (C# scripts with the file extension `csx`). C# Script makes it easy to make code changes in the web browser using the Azure Portal.

When you use the Web Bot App, the project is based on the Bot Builder SDK (`https://dev.botframework.com/`). Select the Web Bot App to create your first Bot Service (see Figure BC4-1). You need to specify the name of the bot that needs to be unique. In case you have multiple Azure subscriptions, you can select the one your bot should be associated with. Resource groups are useful for grouping resources. If you create resources for a test, put them in the same group. This way, you can delete all the resources after you finished the test simply by deleting the resource group; you don't need to search for all the resources separately.

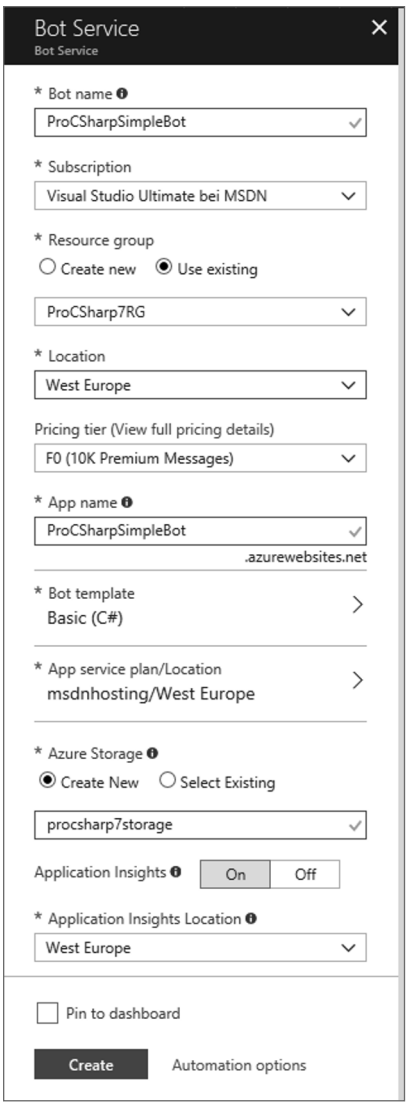

FIGURE BC4-1

Pay attention to the pricing tier offered. Click the pricing information to see the full pricing details; from there you can select the best pricing tier. At the time of this writing, Bot Services offer a free tier that is limited to 10,000 messages a month. This option might serve your purpose.

The App name is the domain name that will be used to access the service via URL. When you select the Bot template, you have the choice for some predefined templates to create a Bot. You start with the Basic (C#) template to create a simple bot. With a Web Bot App, you also need to specify the App Service Plan. When you create a bot, you also need an Azure Storage account. This one is not only used as storage for diagnostic information but also as storage for the bot state. By default, Azure Table Storage is used, but you can change this easily to use the Azure Cosmos DB.

As the bot is created using the template, you can immediately test it from the portal (see Figure BC4-2). When you navigate to the bot, you can select the option Test in Web Chat, and can send messages to the bot. The bot does nothing more than to echo the messages, and it adds a number that is incremented while the communication goes on. This number can be reset by sending the `reset` message.

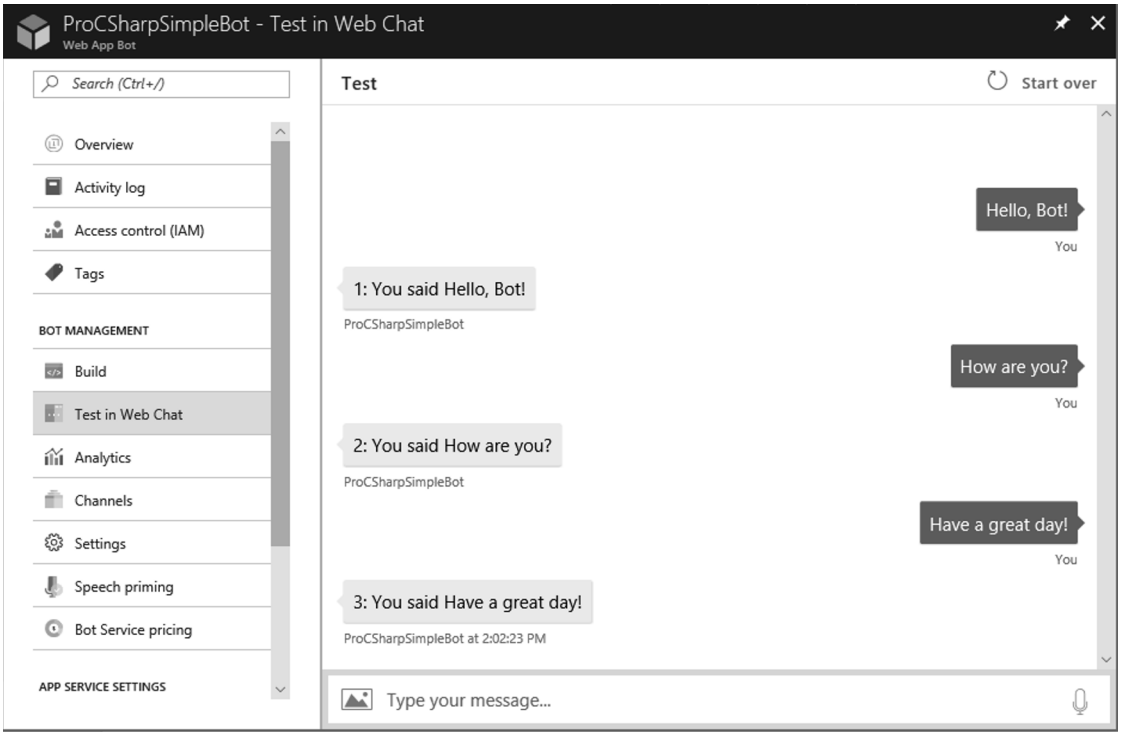

FIGURE BC4-2

Directly from the portal you can download the source code in a zip file. You can open the downloaded source code using Visual Studio and start communication with the bot using the Bot Framework Emulator.

Let's get into the source code of the bot.

> **NOTE** *At the time of this writing, the Microsoft Bot Builder SDK v3 that is used with the Azure Bot Services is based on the .NET Framework; a .NET Core version is not yet available. Bots are created using ASP.NET Web API; you'll see small differences compared to the Web API you've read in Chapter 34, "Web API." A .NET Core version is planned for version 4, but currently neither a Beta nor an Alpha version is available. Check* `https://github.com/ProfessionalCSharp/MoreSamples` *for additional samples as the .NET Core version becomes available.*

Configuration the State Service

With the startup of the app, the dependency injection (DI) container for the bot is configured. The Microsoft Bot Builder v3 uses Autofac for the DI container. The `Conversation` class (namespace `Microsoft.Bot.Builder.Dialogs`) holds a reference to the DI container that can be accessed using the `Container` property. To configure the container, the `UpdateContainer` method of the `Conversation` class is invoked. The `UpdateContainer` method defines a parameter of type `Action` delegate with a `ContainerBuilder`. Autofac supports chaining of modules. In a module, you can configure a list of services by invoking the `RegisterModule` method and passing a new `AzureModule` instance. The `AzureModule` class registers a list of services needed for the bot functionality. Next, a `TableBotDataStore` object is created that is used to store the state of the bot in the Azure Table Storage. The `Register` method of the `ContainerBuilder` registers the `TableBotDataStore` for the `IBotDataStore` contract interface (code file `DialogBotSample/Global.asax.cs`):

```
protected void Application_Start()
{
  Conversation.UpdateContainer(
    builder =>
    {
      builder.RegisterModule(new AzureModule(Assembly.GetExecutingAssembly()));

      var store = new TableBotDataStore(
        ConfigurationManager.AppSettings["AzureWebJobsStorage"]);

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

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

> **NOTE** *The Microsoft Bot Builder SDK v3 is using Autofac for the dependency injection container. This will likely change to* `Microsoft.Extensions.DependencyInjection` *with the next version of the SDK. The* `RegisterModule` *method of Autofac is a method to register a list of services. With the DI container* `Microsoft.Extensions.DependencyInjection`*, you've seen similar functionality with extension methods such as* `AddMvc`*. Instead of using the* `Register` *method and the* `SingleInstance` *method, the* `AddSingleton` *extension method can be used with the Microsoft container.*

Instead of using Azure Table Storage, you can use Azure Cosmos DB or SQL Database. To change the Azure Table Storage to Cosmos DB, you just need to create a `DocumentDbBotDataStore` instead of the `TableBotDataStore` and register this one:

```
var store = new DocumentDbBotDataStore(docDbEmulatorUri, docDbEmulatorKey);
builder.Register(c => store)
  .Keyed<IBotDataStore<BotData>>(AzureModule.Key_DataStore)
  .AsSelf()
  .SingleInstance();
```

Receiving Bot Messages

Messages are received in the Web API controller. With ASP.NET Web API, the Web API controller derives from the base class `ApiController`. As the controller is named `MessagesController`, the URL to the controller is `/api/Messages`. The `MessagesController` has the `BotAuthentication` attribute annotated. This attribute specifies that only registered channels can be used to access the bot. While testing the bot with a local emulator, you can remove this attribute, but you should add it back before deploying to Azure Services. The `MessagesController` defines only a single public `Post` method. This method is the entry point for the communication; the client sends an HTTP POST request containing the information for an Activity. The Activity is the heart of the communication between the client and the bot service. The content is defined by the schema `application/vnd.microsoft.activity` and contains the members participating with the communication, information about the channel that is used, and data entered or spoken by the user.

Activities received from the channel are not only messages entered by the user but also system messages. System messages are handled in the `HandleSystemMessage` method. Here you can implement code when users are added to the conversion, show some information when a user types, and react to ping requests. Normal user-initiated messages are of type `ActivityTypes.Message`. When such a message arrives, the `Conversation` class is used to send a dialog to keep the conversation running. With the code as it is created from the template, the `EchoDialog` class is used. As the `EchoDialog` class only returns the same information, the sample code gives the user options to reserve a table in a restaurant or to order from the menu. This is implemented in the `RootDialog` class used here (code file `DialogBotSample/Controllers/MessagesController.cs`):

```
[BotAuthentication]
public class MessagesController : ApiController
{
  [ResponseType(typeof(void))]
  public virtual async Task<HttpResponseMessage> Post(
    [FromBody] Activity activity)
  {
    if (activity == null) throw new ArgumentNullException(nameof(activity));

    if (activity.GetActivityType() == ActivityTypes.Message)
    {
      await Conversation.SendAsync(activity, () => new RootDialog());
    }
    else
    {
      HandleSystemMessage(activity);
    }
    return Request.CreateResponse(HttpStatusCode.OK);
  }

  private Activity HandleSystemMessage(Activity message)
  {
    if (message.Type == ActivityTypes.DeleteUserData)
    {
    }
```

```
      else if (message.Type == ActivityTypes.ConversationUpdate)
      {
      }
      else if (message.Type == ActivityTypes.ContactRelationUpdate)
      {
      }
      else if (message.Type == ActivityTypes.Typing)
      {
      }
      else if (message.Type == ActivityTypes.Ping)
      {
      }
      return null;
    }
  }
```

Defining a Dialog

A dialog needs to be serializable (marked with the `Serializable` attribute) and implement the generic interface `IDialog`. This interface defines the method `StartAsync` that is invoked when the dialog is started. Within the implementation of this method, invoking the `Wait` on the `IDialogContext` suspends the dialog until the user has sent a message. When a message is received, the method `MessageReceivedAsync` is invoked. The message received is accessed from the `IAwaitable`; this contains an `IMessageActivity` with the input from the user that can be accessed using the `Text` property. If the user entered text containing "help" or "support," the request is forwarded to the `SupportDialog`. If any other text is entered, the `ShowOptions` method is invoked where the user gets information about the options to select (code file `DialogBotSample/Dialogs/RootDialog.cs`):

```
[Serializable]
public class RootDialog : IDialog<object>
{
  private const string FindNumberOption = "find a number";
  private const string OrderLunchOption = "order lunch";
  private const string ReserveTableOption = "reserve a table";

  public Task StartAsync(IDialogContext context)
  {
    context.Wait(MessageReceivedAsync);
    return Task.CompletedTask;
  }

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

    if (message.Text.ToLower().Contains("help") ||
      message.Text.ToLower().Contains("support"))
    {
      await context.Forward(new SupportDialog(), ResumeAfterSupportDialog,
        message, CancellationToken.None);
    }
    else
    {
      ShowOptions(context);
    }
  }
  //...
}
```

With the `Forward` from the previous code snippet, the `SupportDialog` from the next code snippet is immediately started. Because of the `Forward`, the `Wait` method in `StartAsync` does not wait for a new message; it immediately processes the message in the `MessageReceivedAsync` method. A ticket number is generated, and a message is returned to the user by calling the `PostAsync` method. Invoking the `Done` method closes the current dialog and returns to the parent dialog (code file `DialogBotSample/Dialogs/SupportDialog.cs`):

```
[Serializable]
public class SupportDialog : IDialog<int>
{
  public Task StartAsync(IDialogContext context)
  {
    context.Wait(MessageReceivedAsync);
    return Task.CompletedTask;
  }

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

    var ticketNumber = new Random().Next(0, 10000);

    await context.PostAsync($"Your message '{message.Text}' was registered. " +
      "Once we resolve it; we will get back to you.");

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

What are the options you have with the `IDialogContext` to step in the stack of the conversion? Until now you've seen the `Wait` and `Forward` methods to change the dialog stack, and the `PostAsync` method to communicate with the user. The `IDialogContext` interface derives from the interfaces `IDialogStack`, `IBotContext`, `IBotData`, and `IBotToUser`. The following table lists what you can do with the context.

INTERFACE	MEMBER	DESCRIPTION
`IDialogStack`	`Call`	Calls a child dialog and adds it to the top of the stack
	`Done`	The current dialog is completed. Passes the result to the parent dialog.
	`Fail`	The current dialog is on error. Returns an exception to the parent dialog.
	`Forward`	Calls a child dialog and forward the message to the child dialog.
	`Post`	Posts an event to the queue.
	`Reset`	Resets the stack.
	`Wait`	Suspends the current dialog and wait for a message to arrive.
`IBotContext`	`Activity`	Accesses the activity associated with the conversation.

continues

(continued)

INTERFACE	MEMBER	DESCRIPTION
IBotData	UserData ConversationData PrivateConversationData FlushAsync LoadAsync	Reads and writes data in the IBotDataStore and accesses bot data associated with the user and associated with the conversation.
IBotToUser	MakeMessage	Creates an IMessageActivity message that can be sent to the user.
	PostAsync	Sends a message to the user.

In the parent dialog, the RootDialog, the method ResumeAfterSupportDialog is now invoked. A ResumeAfter delegate to this method was passed when forwarding the request to the SupportDialog with the Forward method. Here, the result from the SupportDialog is retrieved (the ticket number), and a message is posted to the user. The next Wait waits for the next message from the user, and invokes the MessageReceivedAsync method as soon as the message arrives (code file DialogBotSample/Dialogs/RootDialog.cs):

```
private async Task ResumeAfterSupportDialog(IDialogContext context, IAwaitable<int> result)
{
  var ticket = await result;
  await context.PostAsync(
    $"Thanks for contacting support. Your ticket number is {ticket}");
  context.Wait(MessageReceivedAsync);
}
```

When you run the app and the Bot Framework Emulator, you can enter text like "I need help with this bot," which forwards the request to the SupportDialog as shown in Figure BC4-3.

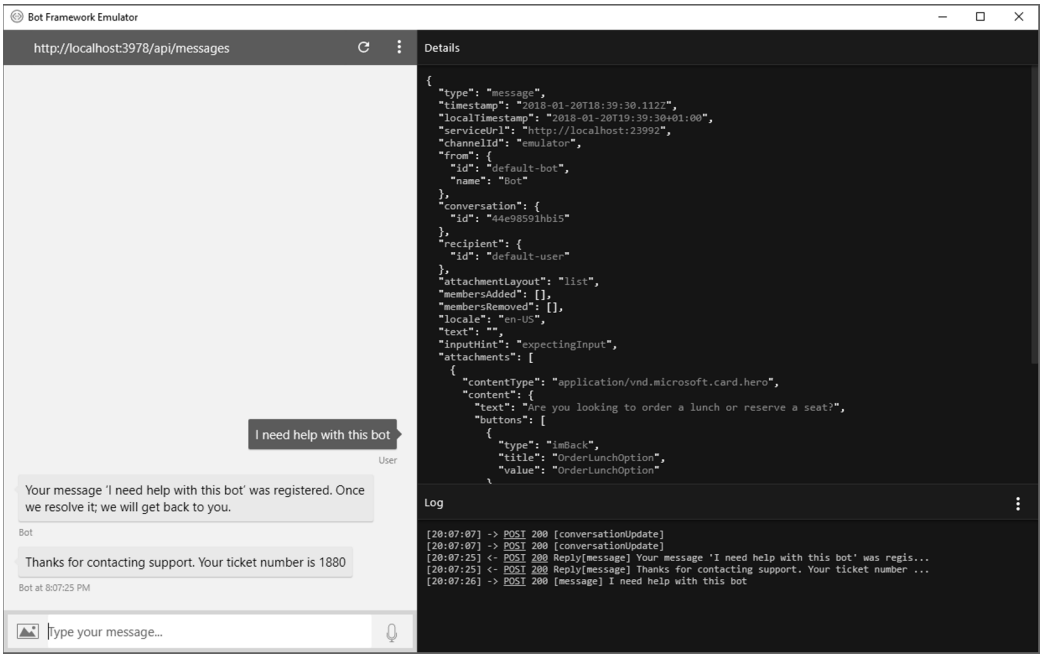

FIGURE BC4-3

Using the PromptDialog

The `ShowOptions` method that is invoked when the user doesn't ask for help displays another dialog; it prompts the user for a choice on one of the two options previously defined with the help of `PromptDialog.Choice`. The second parameter of the `Choice` method defines the method that should be invoked when the user made a choice (`OnOptionSelected`). The third parameter defines the different options, the fourth parameter defines introduction text, the fifth parameter defines the text when the user selects an invalid option, and the last parameter defines the number of retries the user can do in cases when the information is invalid (code file `DialogBotSample/Dialogs/RootDialog.cs`):

```
private void ShowOptions(IDialogContext context)
{
  PromptDialog.Choice(context, OnOptionSelected,
    new List<string>() { OrderLunchOption, ReserveTableOption },
    "Are you looking to order a lunch or reserve a seat?",
    "Not a valid option", 3);
}
```

The user is given the choice to order lunch or reserve a table, as shown in Figure BC4-4. With this option, the user does not need to specify the answer exactly as written in the options; the user can, for example, type "order" or "lunch," or the user can type "1" or "first"; each of these cases will result in lunch being ordered.

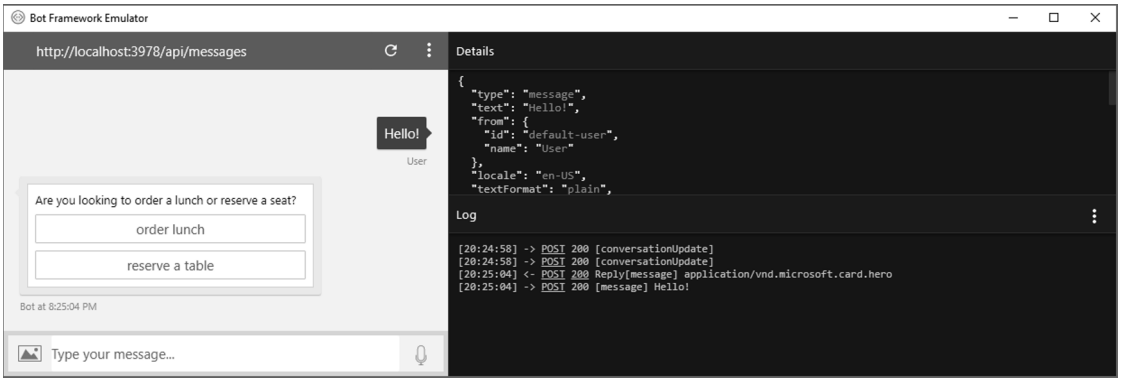

FIGURE BC4-4

When the user selects an option, the `OnOptionSelected` method is invoked. Depending on the selection, the `OrderLunchDialog` or the `ReserveTableDialog` dialogs are used for the ongoing communication. In case the user made too many wrong attempts, the `TooManyAttemptsException` is thrown (code file `DialogBotSample/Dialogs/RootDialog.cs`):

```
private async Task OnOptionSelected(IDialogContext context,
  IAwaitable<string> result)
{
  try
  {
    string optionSelected = await result;

    switch (optionSelected)
    {
      case OrderLunchOption:
        context.Call(new OrderLunchDialog(), ResumeAfterOptionDialog);
        break;
```

```
        case ReserveTableOption:
          context.Call(new ReserveTableDialog(), ResumeAfterOptionDialog);
          break;
      }
    }
    catch (TooManyAttemptsException ex)
    {
      await context.PostAsync("Ooops! Too many attemps :(. " +
        "But don't worry, I'm handling that exception and you can try again!");

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

The `PromptDialog` class offers different simple ways to ask questions of the user. You've seen the choice of different options, but there are others, as shown in the following table.

PROMPTDIALOG METHOD	DESCRIPTION
`Choice`	Prompts the user for one of a list of choices
`Confirm`	Asks a yes/no question
`Number`	Asks for a number of type long or type double
`Text`	Asks for a string

You can also create iterations with these dialogs. The `FindNumberDialog` creates a random number between 1 and 50 and asks for the correct number a maximum of six times. With the `StartAsync` method of the `FindNumberDialog`, the first `PromptDialog.Number` is started to ask the user for a number. As the number is entered, the method `ProcessNumber` is invoked where the number is checked for correctness, and `PromptDialog.Number` is invoked again where `ProcessNumber` is invoked recursively until the number is correctly guessed or the maximum number of iterations is reached (code file `DialogBotSample/Dialogs/FindNumberDialog.cs`):

```
[Serializable]
public class FindNumberDialog : IDialog<int>
{
  private int _theNumber;
  private bool _success = false;
  private int _loop = 0;

  public async Task StartAsync(IDialogContext context)
  {
    _theNumber = new Random().Next(1, 50);
    await context.PostAsync(
      "Find a number between 1 and 50 with a max of 6 attempts");
    PromptDialog.Number(
      context,
      new ResumeAfter<long>(ProcessNumber),
      "enter a number between 1 and 50");
  }

  private async Task ProcessNumber(IDialogContext context,
    IAwaitable<long> result)
  {
    var number = await result;
    _loop++;
    if (number == _theNumber)
    {
      _success = true;
    }
```

```
        else if (number < _theNumber)
        {
          await context.PostAsync("too small, try again");
        }
        else
        {
          await context.PostAsync("too big, try again");
        }
        if (!_success && _loop < 6)
        {
          PromptDialog.Number(
            context,
            ProcessNumber,
            "enter a number between 1 and 50");
        }
        else
        {
          string message;
          if (_success)
          {
            message = $"Success - you made it! The correct number is {_theNumber}";
          }
          else
          {
            message = $"Better luck next time. The correct number is {_theNumber}";
          }
          await context.PostAsync(message);
          context.Done(_theNumber);
        }
      }
    }
```

Figure BC4-5 shows a run of the application to guess the correct number.

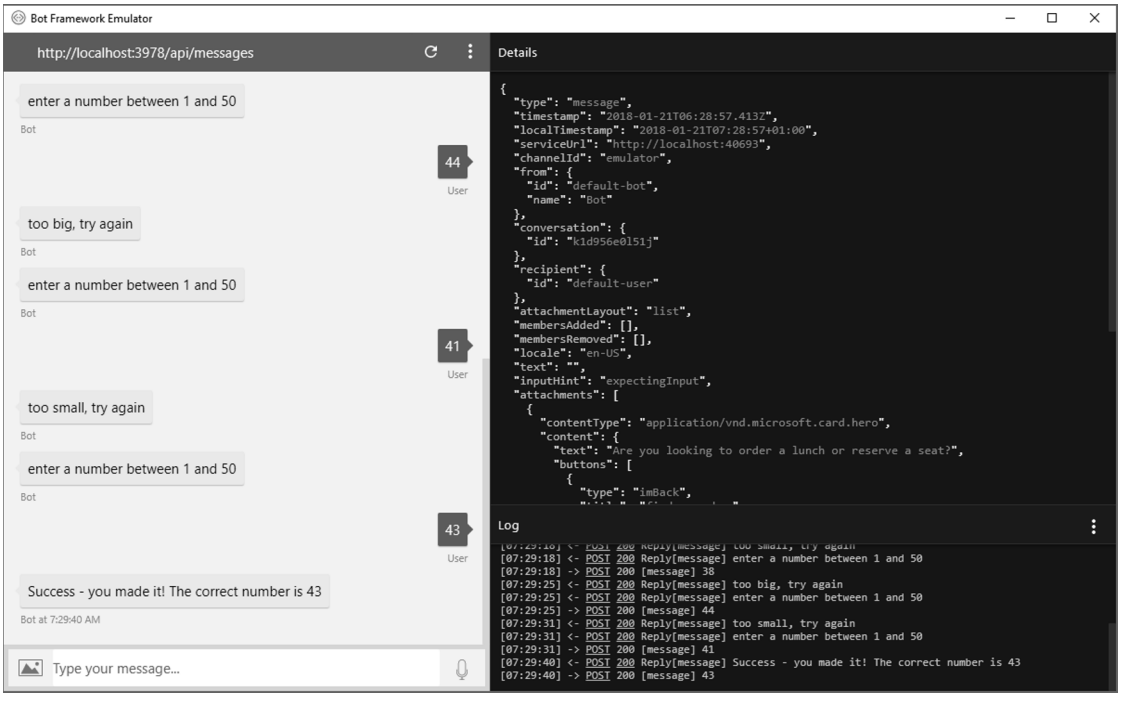

FIGURE BC4-5

USING FORM FLOW FOR DIALOGS

Often you need to collect some information from a user. By using Form Flow (namespace `Microsoft.Bot.Builder.FormFlow`), you can define a class with properties where you define what information is needed by the user. With the `ReserveTableDialog`, the `ReserveTableQuery` is used. The `ReserveTableQuery` class defines the properties `Date`, `Time`, and `People`—values needed from the user to reserve a table. By adding annotations to these properties, you can define what to ask the user, and what information is needed. The `Prompt` attribute defines the text that is sent to the user. With the `Numeric` attribute, you define a number range that can be entered by the user (code file `DialogBotSample/Dialogs/ReserveTableQuery.cs`):

```
[Serializable]
public class ReserveTableQuery
{
  [Prompt("Please enter the {&} for the reservation")]
  public DateTime? Date { get; set; }

  [Prompt("Please enter the {&}")]
  public DateTime? Time { get; set; }

  [Numeric(1, 50)]
  [Prompt("For how many {&} should be reserved?")]
  public int? People { get; set; }
}
```

> **NOTE** *The* `{&}` *placeholder defined with the* `Prompt` *attribute is part of the pattern language from the Bot Framework. The curly braces identify elements that will be replaced at runtime. Using* `&` *shows the description of the current field (or property in the sample code), which is by default the name.*

The `ReserveTableDialog` is implemented like the other dialogs shown earlier that implement the interface `IDialog`. The `StartAsync` method first posts a welcome message. Next, the `FormDialog` is created using the `FormDialog.FromForm` method. This method needs a first parameter of type `BuildFormDelegate<T>`, which requires a method returning `IForm<T>` (which is `IForm<ReserveTableQuery>` in the sample code). After the form dialog is created, the `reserveTableQuery` form dialog is called as child dialog, and it's put on top of the stack by invoking the `Call` method on the context (code file `DialogBotSample/Dialogs/ReserveTableDialog.cs`):

```
[Serializable]
public class ReserveTableDialog : IDialog<object>
{
  public async Task StartAsync(IDialogContext context)
  {
    await context.PostAsync("Welcome to reserving a table!");

    var reserveTableQuery = FormDialog.FromForm(BuildReserveTableForm,
      FormOptions.PromptFieldsWithValues);

    context.Call(reserveTableQuery, ResumeAfterReserveTableFormDialog);
  }
  //...
}
```

The `BuildReserveTableForm` method that is passed to the `FormDialog.FromForm` method to create a form makes use of the `FormBuilder`. The `FormBuilder` class defines methods to create the form using a fluent API. First, the property `Date` of the `ReserveTableQuery` class needs to be filled by invoking the `Field`

method that passes the name of this property. The `Confirm` method confirms the user input by asking the user if the date is correct. Next, all the remaining properties of the class `ReserveTableQuery` are asked—using the `FormBuilder` method `AddRemainingFields`. On completion of the user input, the lambda expression assigned to the field `reservationAnswer` is invoked to confirm the reservation using the `PostAsync` method on the context. After defining the form, the form is built by invoking the `Build` method (code file `DialogBotSample/Dialogs/ReserveTableDialog.cs`):

```
public IForm<ReserveTableQuery> BuildReserveTableForm()
{
  OnCompletionAsyncDelegate<ReserveTableQuery> reservationAnswer =
    async (context, reservation) =>
      await context.PostAsync($"Thanks. Reserving {reservation.People} " +
        $"seats on {reservation.Date:D} at {reservation.Time:t}");

  return new FormBuilder<ReserveTableQuery>()
    .Field(nameof(ReserveTableQuery.Date))
    .Confirm("Looking to reserve a seat at {Date:d}?")
    .AddRemainingFields()
    .OnCompletion(reservationAnswer)
    .Build();
}
```

Figure BC4-6 shows a run of the application to reserve a table. The user has different options to enter the date and time—for example, to reserve the table at 2:00 p.m., the user can also specify a time of 14:00.

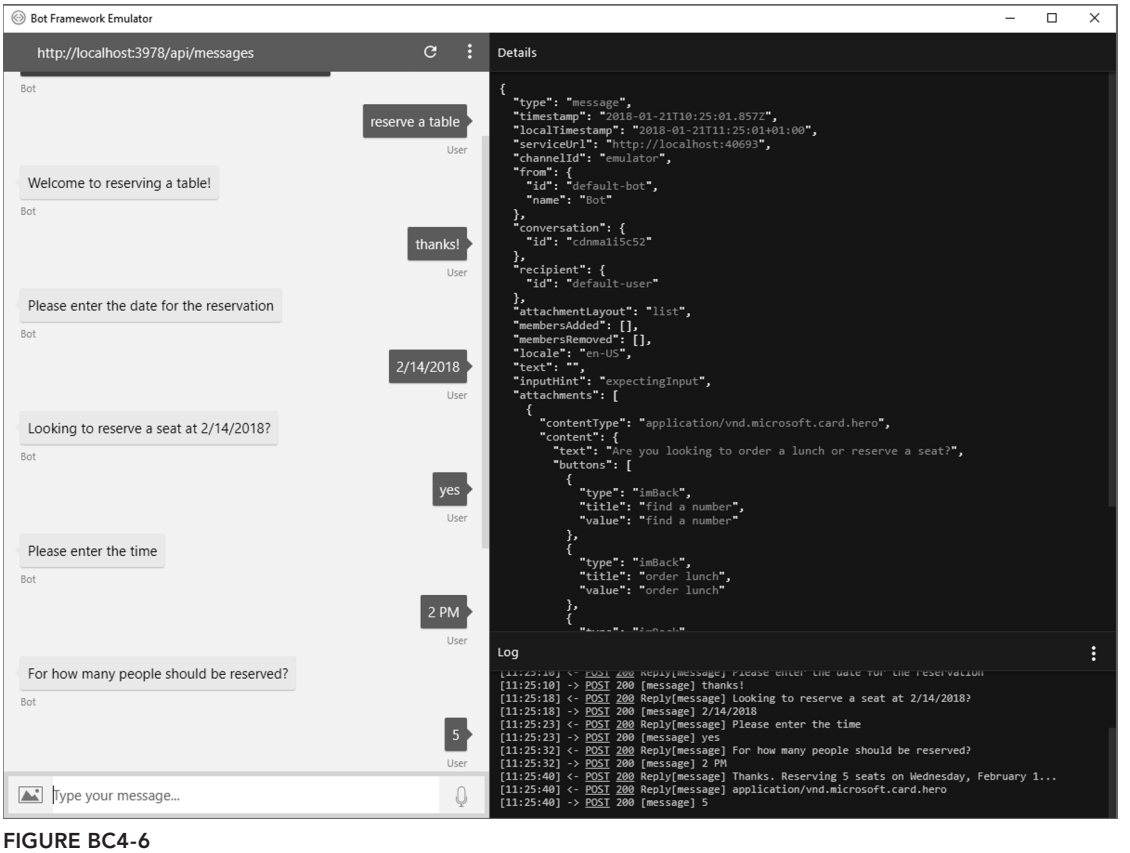

FIGURE BC4-6

CREATING A HERO CARD

Instead of just communicating via messages, the bot can show cards. Cards started with Facebook as ads in the timeline and web apps. Nowadays, many different cards are used to approve reservations, show weather information, provide reminders, show information on airline updates, receipts, and much more.

The Bot Framework SDK supports these cards as shown in the following table.

CARD	DESCRIPTION
Animation Card	Plays an animated GIF or a short video
Audio Card	Plays an audio file
Hero Card	A card with a large image, text, and buttons
Thumbnail Card	A card with a single thumbnail image, text, and buttons
Receipt Card	A card to provide a receipt to a user
SignIn Card	A card with controls to initiate the sign-in process
Video Card	Plays videos

After the user successfully enters the reservation, it's a good idea to show a *hero card*. The address of the method `ResumeAfterReserveTableFormDialog` is passed to the `context.Call` method with the `ResumeAfter` delegate, thus this method is invoked when the reservation is completed.

> **NOTE** *With the reservation of the table, a real app should do more than return a message via the bot; it also should invoke the Web API from the restaurant to submit the registration information. Because there's nothing special about how this is done from bots, the sample app does not implement this functionality.*

With the sample implementation, a hero card is created and shown to the user. The `HeroCard` class is defined in the `Microsoft.Bot.Connector` namespace. Such a card can have a title, subtitle, text, images, and buttons associated with it. With the sample code, the `Title`, `Subtitle`, and `Images` properties are filled. The `HeroCard` is added to the attachments of the `resultMessage` created using the `MakeMessage` method. Finally, the message is sent using `PostAsync` (code file `DialogBotSample/Dialogs/ReserveTableDialog.cs`):

```
public async Task ResumeAfterReserveTableFormDialog(IDialogContext context,
  IAwaitable<ReserveTableQuery> result)
{
  try
  {
    var reservation = await result;

    // TODO: call the reservation API of the restaurant
    var resultMessage = context.MakeMessage();

    var heroCard = new HeroCard
    {
```

```
        Title = "Reservation",
        Subtitle = $"for {reservation.People} at " +
          $"the date {reservation.Date:D} and time {reservation.Time:t}",
        Images = new List<CardImage>()
        {
          new CardImage
          {
            Url = "https://kantinem101.blob.core.windows.net/" +
              "menuimages/Hirschragout_250"
          }
        }
      };
      resultMessage.Attachments.Add(heroCard.ToAttachment());

      await context.PostAsync(resultMessage);
    }
    catch (FormCanceledException ex)
    {
      string reply = "You canceled the operation. " +
        "Quitting from the reservation.";
      await context.PostAsync(reply);
    }
    finally
    {
      context.Done<object>(null);
    }
  }
```

The conversation from the table reservation is now completed, and the hero card is shown as in Figure BC4-7.

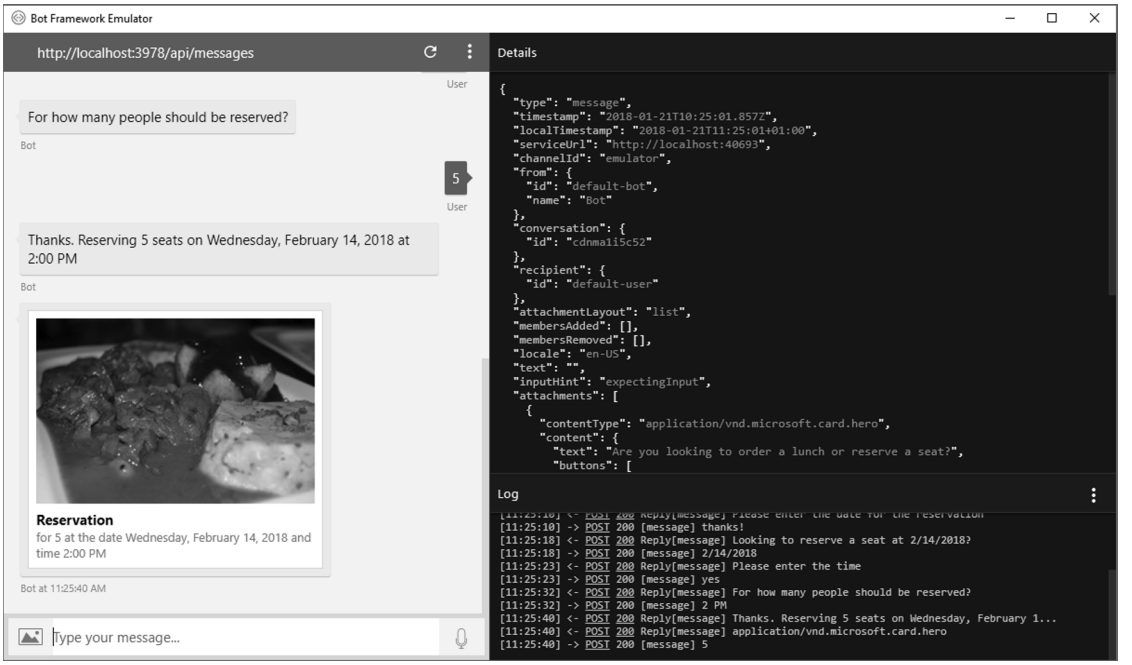

FIGURE BC4-7

> **NOTE** *Cards can be used with many different technologies and websites; you can use cards on Twitter, Skype, Windows toasts, and more. However, every website showing cards makes use of a different technology. The new technology* Adaptive Cards *should solve this. You can write cards with JSON and use them with different renderers in your bot, Skype, Teams, and Kik, as well as with WPF, Windows apps, and Xamarin apps. Adaptive Cards is currently available in preview; check out* `https://adaptivecards.io` *for samples, the schema explorer, and visualizers to see how the cards are adapted for different looks based on where they are used.*

BOTS AND LUIS

Bots can easily make use of Microsoft Azure Cognitive Services. Cognitive Services offer intelligent algorithms using artificial intelligence (AI) to understand text users enter and speech. You can use it to analyze images and videos for its contents, find recommendations, and use semantic search. Of course, Cognitive Services can also be used from other kind of apps aside from bots.

Language Understanding Intelligence Services (LUIS) is a service that understands what the user wants. The samples in the previous sections checked for specific terms to be part of the query—for example, "help" or "support." Using LUIS, the bot's understanding of the user is a lot more sophisticated.

The LUISBotSample is based on the Language Understanding template from the Microsoft Azure Web App Bot templates. Here, a dialog is used that derives from the base class `LuisDialog`. A `LuisDialog` contains methods with `IDialogContext` and `LuisResult` parameters that have a `LuisIntent` annotated. An intent specifies what needs to be achieved. Predefined intents that are automatically created from the Azure template are `Greeting`, `Cancel`, and `Help`. The intents are clear with these terms. You can remove the predefined intents and add custom ones. An intent to create a reservation for a restaurant could be named `Restaurant.Reservation`. This intent needs to be added to the LUIS service.

With the dialog, depending on the message received, if it matches an intent, the corresponding method is invoked. The template-generated code invokes the same method with all intent-annotated methods: `ShowLuisResult`. This method is invoked to show information about the intent. The sample code is a little simplified from the generated code to just define one method and to apply multiple `LuisIntent` attributes to this method (code file `LuisBotSample/Dialogs/BasicLuisDialog.cs`):

```
[Serializable]
public class BasicLuisDialog : LuisDialog<object>
{
  public BasicLuisDialog() : base(new LuisService(new LuisModelAttribute(
    ConfigurationManager.AppSettings["LuisAppId"],
    ConfigurationManager.AppSettings["LuisAPIKey"])))
  {
  }

  [LuisIntent("Greeting")]
  [LuisIntent("None")]
  [LuisIntent("Help")]
  [LuisIntent("Cancel")]
  public async Task SeveralIntents(IDialogContext context, LuisResult result)
  {
    await this.ShowLuisResult(context, result);
```

```
    }

    private async Task ShowLuisResult(IDialogContext context, LuisResult result)
    {
      await context.PostAsync($"You have reached {result.Intents[0].Intent}. " +
        $"You said: {result.Query}");
      context.Wait(MessageReceived);
    }
  }
```

When you run the Bot Framework Emulator, you can talk to the bot and see to which intent the message maps, as shown in Figure BC4-8.

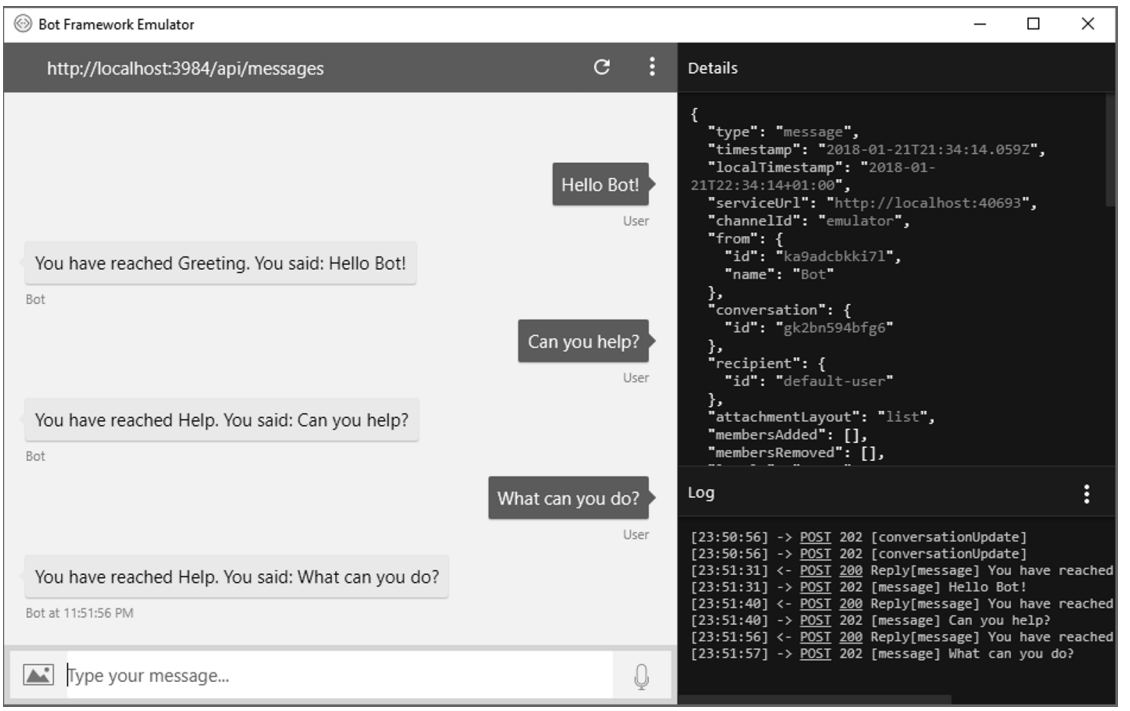

FIGURE BC4-8

Defining Intents and Utterances

For using LUIS, you need to register your app with `https://luis.ai`. When you create the bot from Microsoft Azure using the Language Understanding template, the registration happens automatically. Intents for `Greeting`, `Cancel`, and `Help` are created automatically.

An intent is defined by multiple utterances like shown in Figure BC4-9. The user can say, “Hi bot,” “Hiya,” “How are you doing today?” “Good night,” “Hello bot,” and “How are you doing?” which all means the same intent—Greeting.

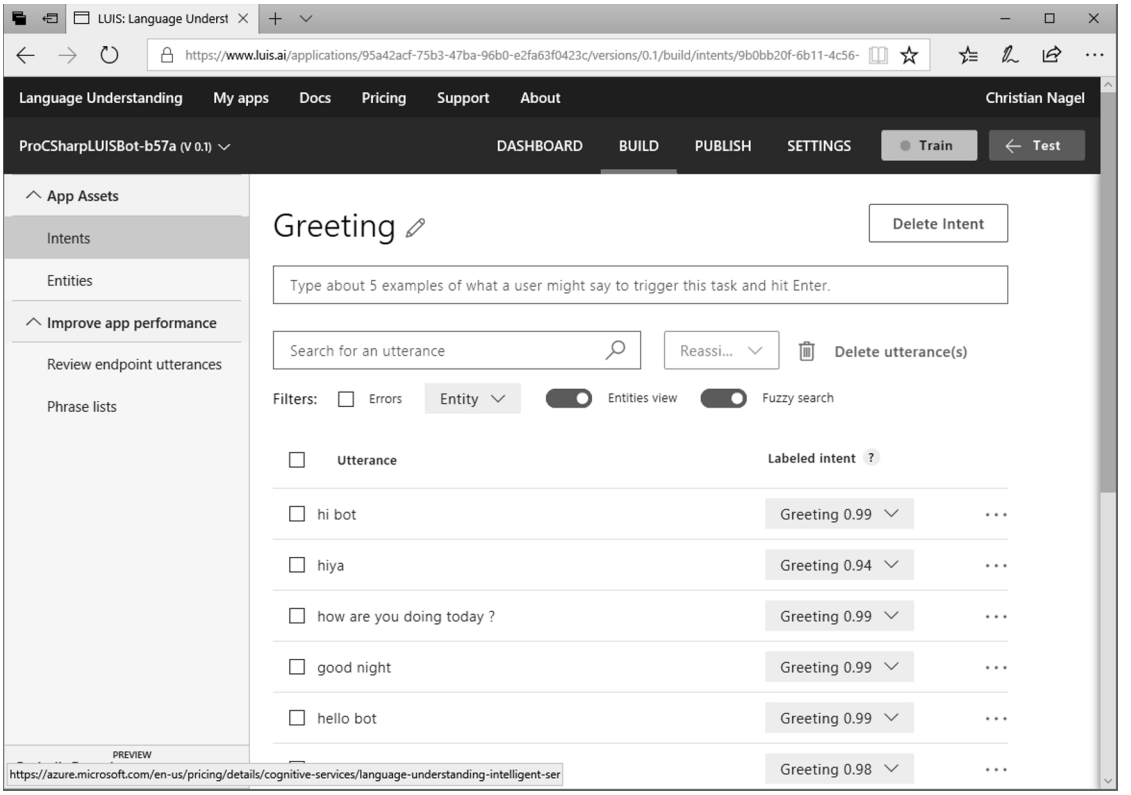

FIGURE BC4-9

You can also specify entities for your utterances to easily get some information about what the user says. For example, if a user says, "I want to reserve a table for 5 people on Saturday at 6," you can get the number of people, the date from Saturday, and the time from 6. On the LUIS website you can specify prebuilt entities such as `number`, `email`, `url`, `money`, `phonenumber`, prebuilt domain entities such as `Calendar.Subject`, `Events.Address`, `Music.ArtistName`, `Note.Text`, `Places.Cuisine`, `Taxi.Address`, and many more.

With the LUIS website, you can create the intent `RestaurantReservation`, and add phrases such as

- Please reserve a seat on February 14.
- I want a table for 5 people on Wednesday at 6 p.m.
- Can I reserve a table for 10 on Sunday at 19:00?
- I would like to reserve a table.
- Can I reserve a table?

With these phrases, words within the phrase can be clicked to map it to an entity. With the intent `RestaurantReservation`, the entities from the following list are defined:

- `Reservation.Weekday`
- `Reservation.Time`
- `Reservation.Day`
- `Reservation.Number`
- `Reservation`

The entities `Reservation.Weekday`, `Reservation.Time`, and so on are simple entities, whereas the `Reservation` entity is a composite entity that combines the other entities for the reservation (see Figure BC4-10).

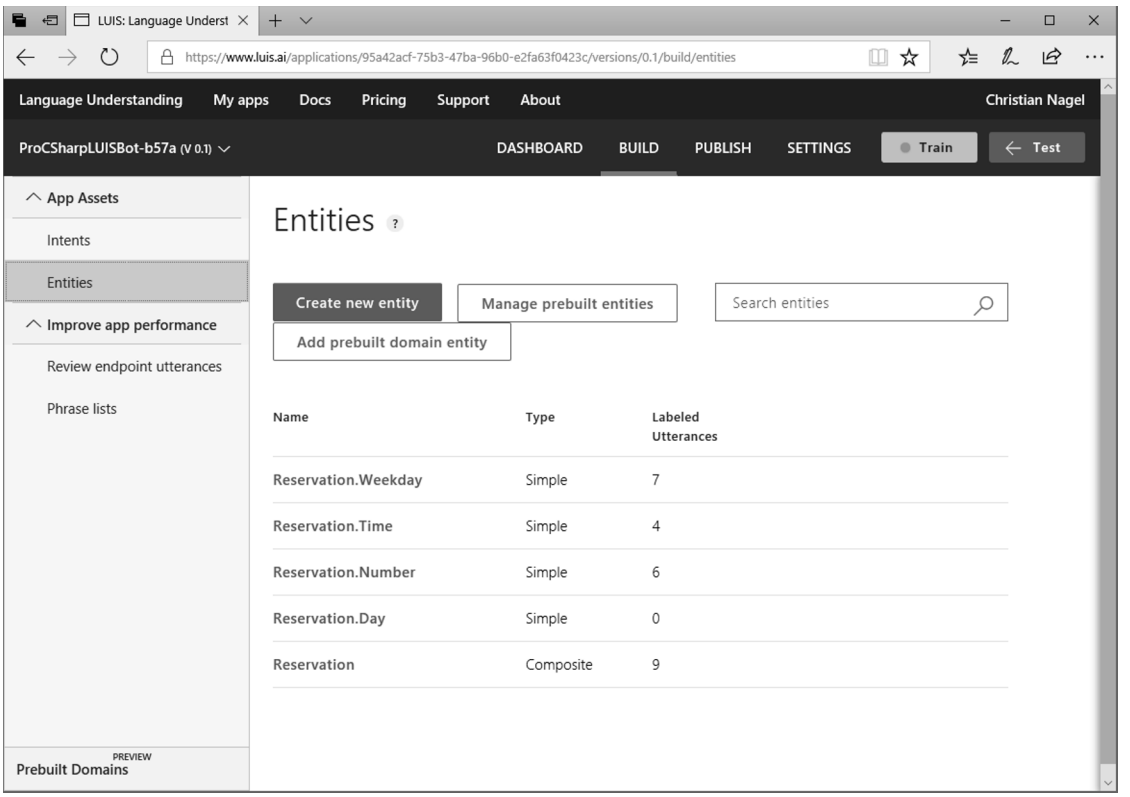

FIGURE BC4-10

With the intent `RestaurantReservation`, several phrases are defined that map to entities (see Figure BC4-11).

After the intents and entities are defined, LUIS can be trained; you just click the Train button. Before the intents are active, you also need to publish the LUIS app which can be done from the LUIS website. You can copy the application ID and the key needed for the bot from the LUIS website as well.

All the information needed for the `RestaurantReservation` intent is collected with the `RestaurantReservation` class. This class allows the `Day`, the `Time`, and the `Number` (number of people for the reservation) to be entered. Instead of the `Day`, `Weekday` can be set; this way, the next weekday is assumed. This class also defines attributes to use it from a dialog asking the user for the needed information. Also, the entities defined in the LUIS website have constants defined in this class to make it easily accessible. (code file `LuisBotSample/Dialogs/RestaurantReservation.cs`):

```
[Serializable]
public sealed class RestaurantReservation
{
  public DayOfWeek Weekday
```

```
    {
      get => Day?.DayOfWeek ?? DayOfWeek.Sunday;
      set => Day = DateTime.Today.AddDays(value.DaysUntilNext());
    }

    [Prompt("Please enter the {&} for the reservation")]
    public DateTime? Day { get; set; }

    [Prompt("Please enter the {&}")]
    public DateTime? Time { get; set; }

    [Numeric(1, 50)]
    [Prompt("For how many people should be reserved?")]
    public int? Number { get; set; }

    public override string ToString() =>
      $"Reservation for {Number} people on {Day:d} at {Time:t}";

    public const string Reservation_Day = "Reservation.Day";
    public const string Reservation_Number = "Reservation.Number";
    public const string Reservation_Time = "Reservation.Time";
    public const string Reservation_Weekday = "Reservation.Weekday";
  }
```

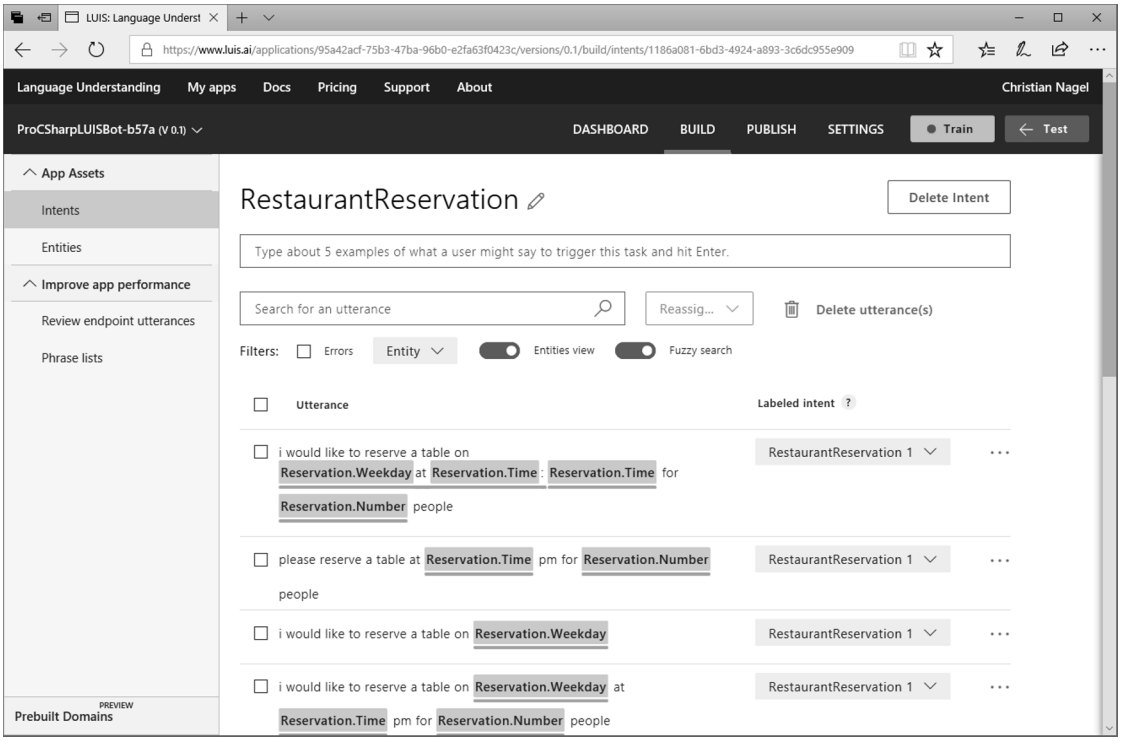

FIGURE BC4-11

Accessing Recommendations from LUIS

When receiving an intent from LUIS, you can use the `LuisResult` to read how likely it's really the intent. The `LuisResult` contains a property called `Intents` that contains a list of `IntentRecommendation` objects. The `IndentRecommendation` contains a `Score` property of type double that gives the information how likely it's the intent. If multiple intents match, you can see all these intents with different scores. Part of the `LuisResult` are also entities and composite entities accessible from the `Entities` and `CompositeEntities` properties.

With the `LuisResult`, you can use the extension method `TryFindEntity` to check whether an entity is contained in the result. Here, the constants defined with the `RestaurantReservation` class are used to check for the entities. If the entity is found, a conversion to the needed type is tried. After the values for the entities are retrieved, a new `FormDialog` is created to ask the user for the missing data (code file `LuisBotSample/Dialogs/BasicLuisDialog.cs`):

```
[LuisIntent("RestaurantReservation")]
public async Task RestaurantReservationIntent(IDialogContext context,
  IAwaitable<IMessageActivity> activity, LuisResult result)
{
  var message = await activity;
  await context.PostAsync($"Welcome to the reservation system! " +
    $"Analyzing your message '{message.Text}'...");

  var reservation = new RestaurantReservation();
  if (result.TryFindEntity(RestaurantReservation.Reservation_Weekday,
    out EntityRecommendation weekdayRecommendation))
  {
    if (Enum.TryParse(weekdayRecommendation.Entity, true,
      out DayOfWeek weekday))
    {
      reservation.Weekday = weekday;
    }
  }

  if (result.TryFindEntity(RestaurantReservation.Reservation_Day,
    out EntityRecommendation dayRecommendation))
  {
    if (DateTime.TryParse(dayRecommendation.Entity, out DateTime day))
    {
      reservation.Day = day;
    }
  }

  if (result.TryFindEntity(RestaurantReservation.Reservation_Time,
    out EntityRecommendation timeRecommendation))
  {
    if (DateTime.TryParse(timeRecommendation.Entity, out DateTime time))
    {
      reservation.Time = time;
    }
  }

  if (result.TryFindEntity(RestaurantReservation.Reservation_Number,
    out EntityRecommendation numberRedommendation))
  {
```

```
    if (int.TryParse(numberRedommendation.Entity, out int number))
    {
      reservation.Number = number;
    }
  }

  var reservationForm = new FormDialog<RestaurantReservation>(
    reservation,
    BuildRestaurantReservationForm,
    FormOptions.PromptInStart, result.Entities);
  context.Call(reservationForm, ResumeAfterRestaurantReservation);
}

public async Task ResumeAfterRestaurantReservation(IDialogContext context,
  IAwaitable<RestaurantReservation> result)
{
  await context.PostAsync("See you soon!!!");
}
```

Using a Form Flow with Active Checks

Next, a Form Flow is created that asks for all values that couldn't be retrieved from the received `LuisResult`. Using the `FormBuilder`, all the properties of the needed `RestaurantReservation` type are asked if they are active. They are active and included in the flow only if the value of the properties is currently null. The comparison with null is done in the lambda expression that is passed to the second parameter of the `Field` method. With an optional third parameter, you can also implement a validation method to check whether the user entered a valid value. After completion of all inputs, a final reservation message is posted (code file `LuisBotSample/Dialogs/BasicLuisDialog.cs`):

```
private IForm<RestaurantReservation> BuildRestaurantReservationForm() =>
  new FormBuilder<RestaurantReservation>()
    .Field(nameof(RestaurantReservation.Day), state => state.Day == null)
    .Field(nameof(RestaurantReservation.Time), state => state.Time == null)
    .Field(nameof(RestaurantReservation.Number), state => state.Number == null)
    .OnCompletion(OnCompleteRestaurantReservation)
    .Build();

public async Task OnCompleteRestaurantReservation(IDialogContext context,
  RestaurantReservation reservation)
{
  await context.PostAsync(
    $"Thanks for the reservation on {reservation.Day:d} "+
      $"at {reservation.Time:t} for {reservation.Number} people");
}
```

When you run the Bot Emulator, you can ask for a reservation like "I would like to reserve a table on Sunday for 5 people." Using the weekday, the next Sunday is calculated, and the value `5` defines the number of seats needed. Now an additional question for the time is asked (see Figure BC4-12). Finally, the complete reservation information is posted.

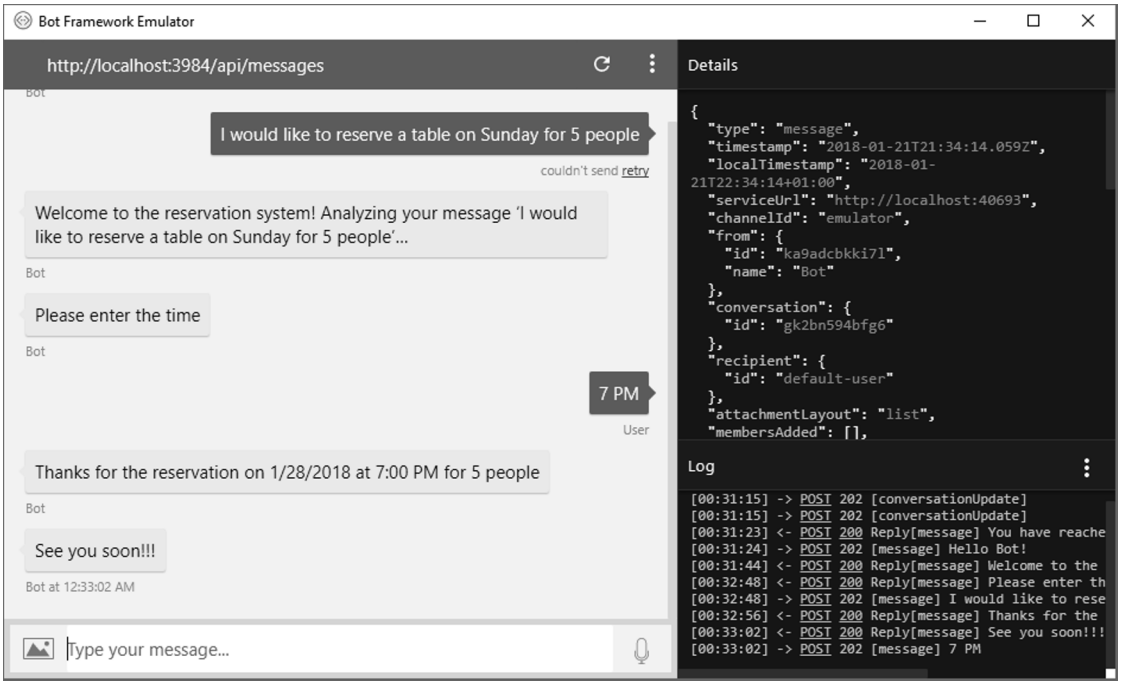

FIGURE BC4-12

SUMMARY

This introduced you to bots using the Microsoft Bot Framework and Microsoft Azure. You've seen how to create a dialog to keep the communication going with the user to receive all the information needed to do some tasks. You've seen how to create simple dialogs and how to use predefined dialogs with the `PromptDialog` class. You were also introduced to how Form Flow automatically fills all the properties needed for a model class with the help of the `FormDialog` and the `FormBuilder` classes.

To make it even more comfortable for the user, you can use cognitive services such as LUIS to understand the user's intent.