

Synergy .NET Bot Tutorial

Introduction

This tutorial will explain how to use Synergy .NET and Microsoft's Bot Framework to build a simple chat bot. The goal of the tutorial is to give you the basic skills that you can use to build more advanced bots.

By the end of the tutorial, you will have a bot that will look up employees from a list and display relevant information. It will use the [Jodah Veloper](#) characters as sample data, but any data could be used instead.

Importantly, the bot created in this tutorial is only designed to run locally using IIS Express and the Bot Framework Emulator. It should be possible to run the bot online, but it would require steps beyond the scope of this tutorial. For example:

- Setting up an Azure Web App service
- Registering the Bot on <https://dev.botframework.com/>
- Adding the App ID and password to the web.config file
- Configuring Synergy licensing for the bot (e.g. device licensing)
- Adding code to set the DAT environment variable at runtime so the bot can access data
- Etc.

Some of these topics are covered in other materials on the GitHub repository that hosts this tutorial: <https://github.com/Synergex/BotFrameworkExample>. However, the tutorial itself will only deal with building the code for the bot and testing it locally.

Requirements

In order to complete this tutorial, you will need a computer with the following software installed (these minimum versions are the versions used to test the tutorial; earlier versions may also work but are not recommended):

- Windows 10, April 2018 Update, 64-bit
- Microsoft Visual Studio 2017, Version 15.7
- Synergy/DE 10.3.3e [SDE] - Both 32-bit and 64-bit
- Synergy DBL Integration for Visual Studio [SDI] 10.3.3e (Build 2376)
- Bot Framework Emulator (version 3.5.36): <https://github.com/Microsoft/BotFramework-Emulator/releases> (Choose the .exe download listed under the "Latest release" tag, e.g. "botframework-emulator-Setup.3.5.36.exe".)
- .NET Framework 4.6

Some instructions are based on the configuration used to develop and test the tutorial, and may vary from one system to another.

Additionally, you will need a Visual Studio project template from the BotFrameworkExample GitHub repository: <https://github.com/Synergex/BotFrameworkExample/tree/master/Templates>. Part 1 of the tutorial will explain how to obtain and install the template.

Completing the Tutorial

Specific steps you should complete are marked by a checkbox bullet point:

☐ Do this step!

All other text is intended to provide additional information, instead of specific steps to follow. This includes other bullet points:

- This is just descriptive text.
- 1. So is this.

If you are reading a printed copy of this tutorial, you can keep track of your progress by checking off the checkboxes as you complete the steps.

Most of the steps will involve adding code to source files. Each page in the tutorial should establish context, so it's clear where new code should be added. For reference purposes, the Appendix of this tutorial contains the final code for each source file in the finished solution (except files that were created by the Bot project template and unaltered in the tutorial). Some steps may have you copy code directly from the appendix, especially when the new code is similar to code you entered earlier in the tutorial.

Part 1: Initial Setup

This tutorial will start from scratch, assuming only that all required software is installed.

Visual Studio Templates

This tutorial will use a Visual Studio project template as the starting point for the Employee Bot. The template is provided as part of the <https://github.com/Synergex/BotFrameworkExample> repository. There are actually two templates there: “BOT Library and Web API Host” and “BOT Library”. We’ll be using the first one, since it includes an additional project that lets you host the Bot as a web service (the second one is intended to be added to an existing solution that already has a web service).

- ☐ Download the template here:
<https://github.com/Synergex/BotFrameworkExample/raw/master/Templates/Synergex%20PSG/BOT%20Library%20and%20Web%20API%20Host.zip>

Once you’ve downloaded the template, you should install it so Visual Studio can access it.

- ☐ Open File Explorer and go to Visual Studio’s ProjectTemplates folder. For Visual Studio 2017, this would be “C:\Users\username\Documents\Visual Studio 2017\Templates\ProjectTemplates”.
- ☐ Create a subfolder named “Synergy”.
- ☐ Inside that folder, create another folder named “Synergex PSG”.
- ☐ Copy the “BOT Library and Web API Host.zip” file into that folder (without unzipping it).

Now the template should be installed and ready to use.

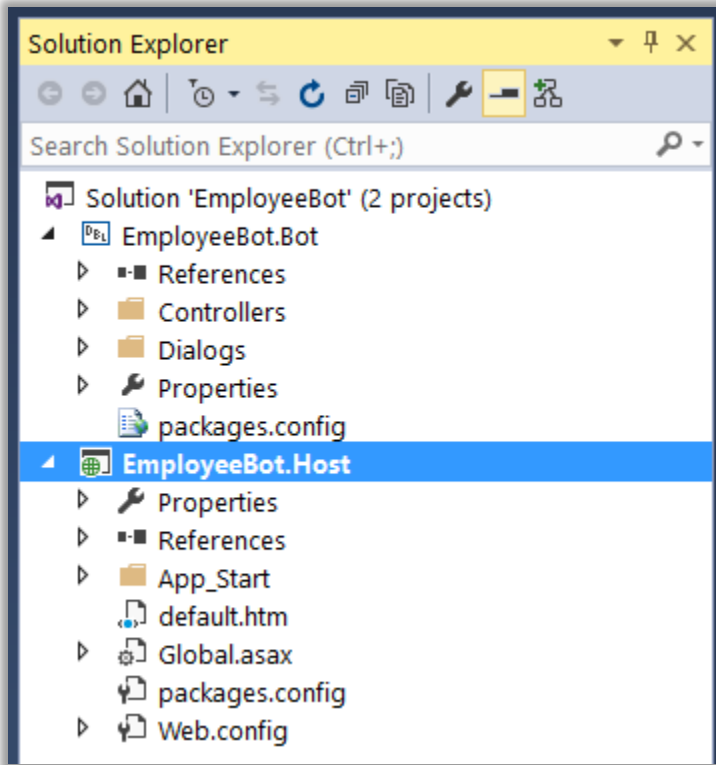
New Bot project

Let’s get started on the Bot project itself.

- ☐ Open Visual Studio.
- ☐ Open the New Project dialog (“File > New > Project…”).
- ☐ On the sidebar, select “Synergy/DE > Synergex PSG”.
- ☐ Select the “BOT Library and Web API Host” project.
- ☐ In the Name field, enter “EmployeeBot”.
- ☐ Click OK to create the project.

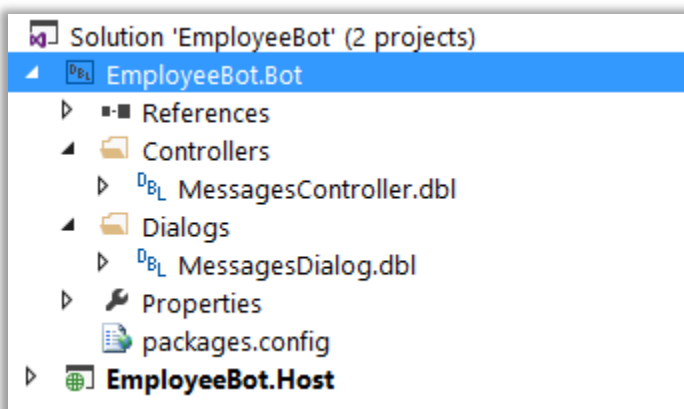
In Solution Explorer, the new solution should look like this:

Building a Bot with Synergy .NET



As you can see, there are two projects in the solution. “EmployeeBot.Bot” is a Synergy .NET class library that’s configured to access the Microsoft.Bot.Builder NuGet package, and handle all of the logic for the Bot. “EmployeeBot.Host” is a web service used to host the Bot. Some of the files in the Host project – such as Global.asax and Web.config – are important if you want to host the Bot online. However, for the purposes of this tutorial, we can leave EmployeeBot.Host as it is for now and just focus on EmployeeBot.Bot.

- ☐ In Solution Explorer, minimize the EmployeeBot.Host project and expand the Controllers and Dialogs folders in EmployeeBot.Bot:



The MessagesController class handles direct input from the user. Theoretically, you could create a bot that used a Controller class to handle all input and output. However, in the default behavior for projects

Building a Bot with Synergy .NET

created from this template, the MessagesController class sends all messages to the MessagesDialog, which then determines what to send back to the user. We'll follow this behavior in this tutorial.

- ☐ Double-click on MessagesDialog.dbl to open it.

Currently, the heart of the MessagesDialog class is the MessageReceivedAsync method:

```
public async method MessageReceivedAsync, @Task
context, @IDialogContext
argument, @IAwaitable<IMessageActivity>
proc
    data message, @IMessageActivity, await argument
    data msgtext = (message.Text == ^null) ? "" : message.Text
    data length = msgtext.Length

    ;; Send reply
    await context.PostAsync(String.Format("You sent {0} which has a length of {1}
characters.",msgtext,length))
    context.Wait(MessageReceivedAsync)
endmethod
```

This method takes whatever the user said and echoes it back, along with the length of the message. Let's see this in action.

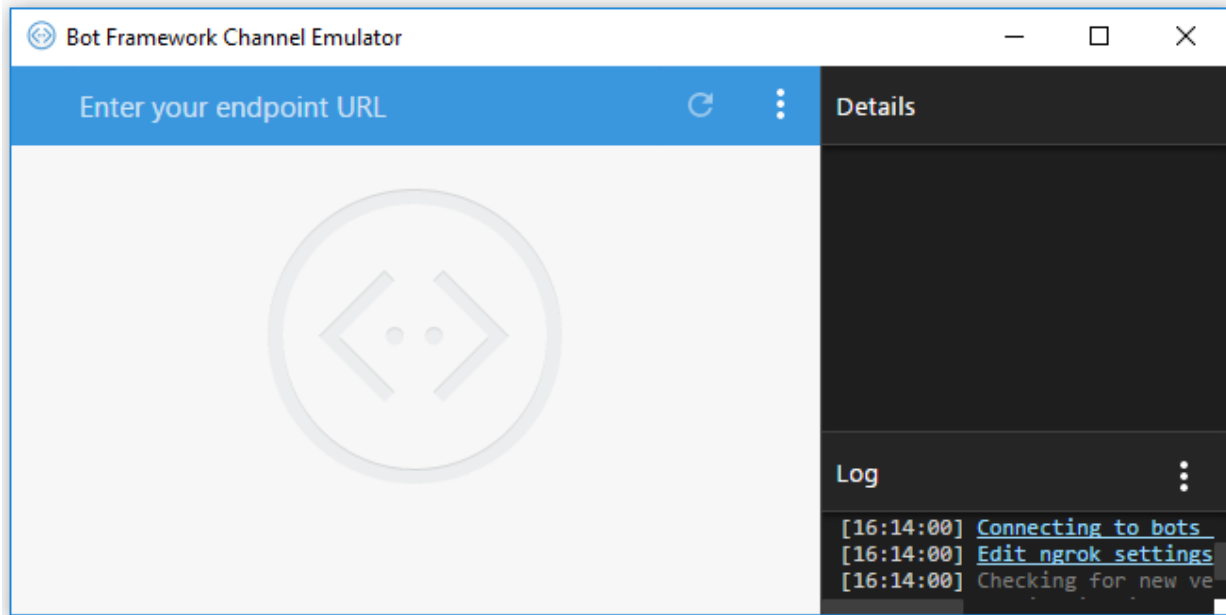
Bot Framework Emulator

- ☐ Select "Build > Build Solution" from the menu.
- ☐ Click the debugging start button to launch the web service locally using the default web browser (click the down arrow next to the button if you want to select a different browser as the default).

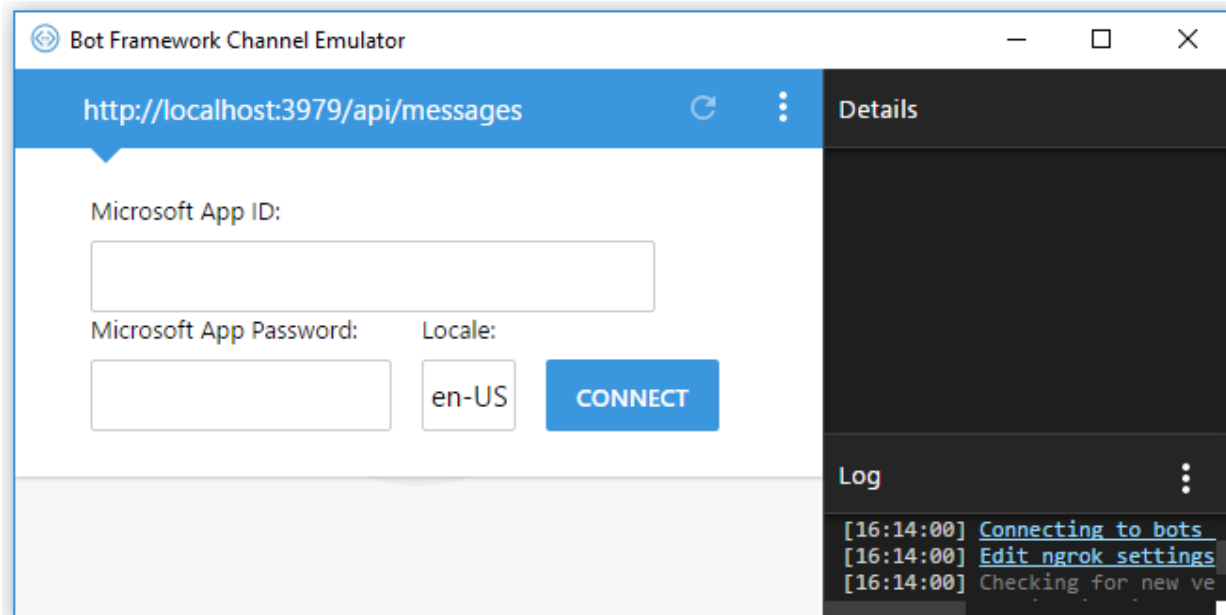


- ☐ In the browser window, select the address (e.g. "http://localhost:3979/" and copy it to the clipboard.
- ☐ Open the Bot Framework Emulator.

Building a Bot with Synergy .NET

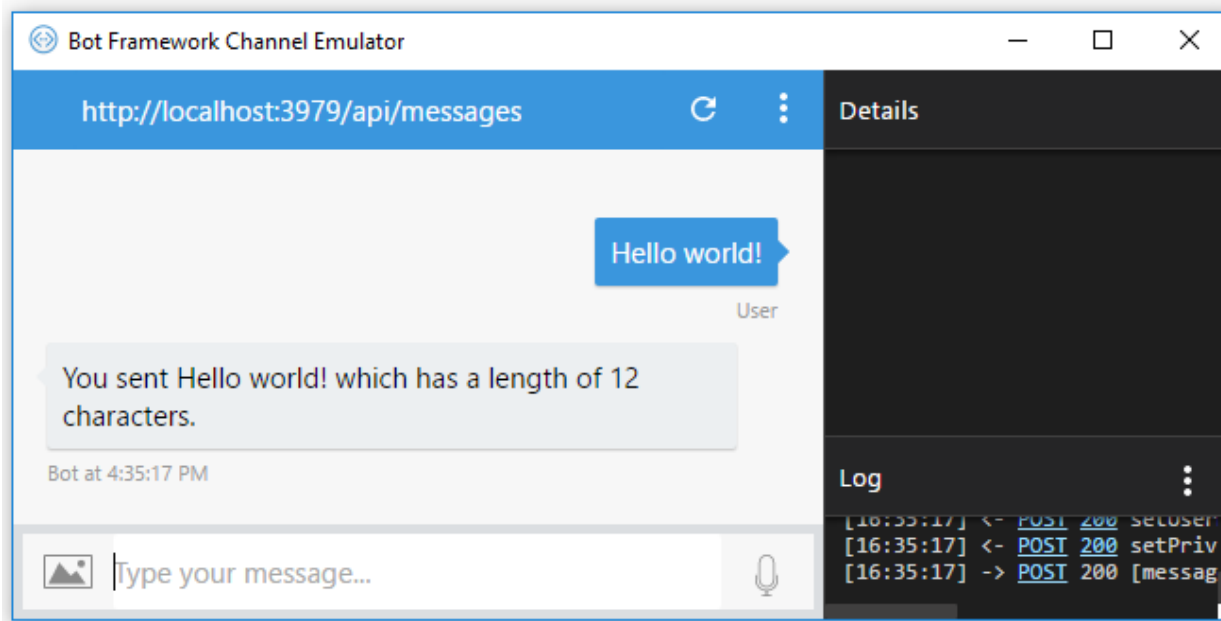


- ☐ Right-click on “Enter your endpoint URL” and paste in the address you copied from the browser.
- ☐ In the same field, type “api/messages” after the address.

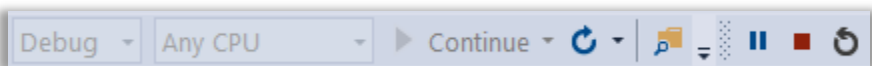


- ☐ Leave the App ID and App Password fields blank, and click CONNECT.
- ☐ Type a message and wait for a response.

Building a Bot with Synergy .NET



- ☐ Go back to Visual Studio and click the debugging stop button to shut down the bot.



You can either leave the emulator running or close it down and open it again before testing the bot again. Future instructions will assume that the emulator is already running. Note that if you close and reopen the emulator, the address that you entered before will be shown in a drop-down list from the address field. This will allow you to connect to the same endpoint without typing the address each time. If you connect to a bot that uses an App ID and Password, those values will also be stored with the address.

Customizing the Response

So far, we've seen what the default bot can do. Now let's customize the response with a help message. That way, anyone using the bot can type 'help' and get a list of the bot's capabilities. We'll add a new source file to contain the new code.

- ☐ In Solution Explorer, right-click on the Dialogs folder (inside the EmployeeBot.Bot project) and click "Add > New Item..."
- ☐ In the "Add New Item" dialog box, select "Code File" from the Synergy category.
- ☐ Type in "ProcessQuery.dbl" as the file's name, and click Add.

Instead of putting the code in a new class, we'll convert MessagesDialog into a partial class and just add new methods to the class in the new file.

- ☐ In MessagesDialog.dbl, add the "partial" keyword to the class definition:
`public partial class MessagesDialog implements IDialog<object>`
- ☐ In ProcessQuery.dbl, add the following code to define another part of the partial class:

Building a Bot with Synergy .NET

```
import System.Text.RegularExpressions
import System.Threading.Tasks
import Microsoft.Bot.Builder.Dialogs

namespace EmployeeBot.Bot.Dialogs

    public partial class MessagesDialog implements IDialog<object>

    endclass

endnamespace
```

You may notice that the class definition in MessagesDialog.dbl includes the {**Serializable**} attribute. This only needs to occur once, and will generate a build error if it's included more than once. So we're not adding it here.

- ☐ In ProcessQuery.dbl, add the following method inside the class:

```
public async method ProcessQueryAsync, @Task
    context,      @IDialogContext
    msgtext,      string
proc

    await context.PostAsync(msgtext)
    mreturn
endmethod
```

This asynchronous method will simply accept a string and send it out to the user. Let's modify the MessageReceivedAsync method to call this new method instead of sending messages back itself.

- ☐ In MessagesDialog.dbl, delete the four lines from 28 (**data** length...) to 31 (**await** context.PostAsync...).
- ☐ In place of those lines, add the following code:

```
await ProcessQueryAsync(context,msgtext)
```

- ☐ Select "Build > Build Solution" from the menu to make sure that everything builds correctly.

Now the Bot will still echo messages in the ProcessQueryAsync method, but it isn't actually processing any queries. Let's change that.

- ☐ In ProcessQuery.dbl, add the following code immediately after the proc statement in the ProcessQueryAsync method:

```
;If a user asks for help, send information about the Bot's abilities
if (Regex.IsMatch(msgtext,"help",RegexOptions.IgnoreCase))
begin

end
```

This code uses a regular expression to match the word 'help' in the input string. It probably isn't necessary to use a regular expression at this point, but we might as well introduce them here since we'll be using them later on.

- ☐ In ProcessQuery.dbl, add the following code inside the begin-end block you just created:

Building a Bot with Synergy .NET

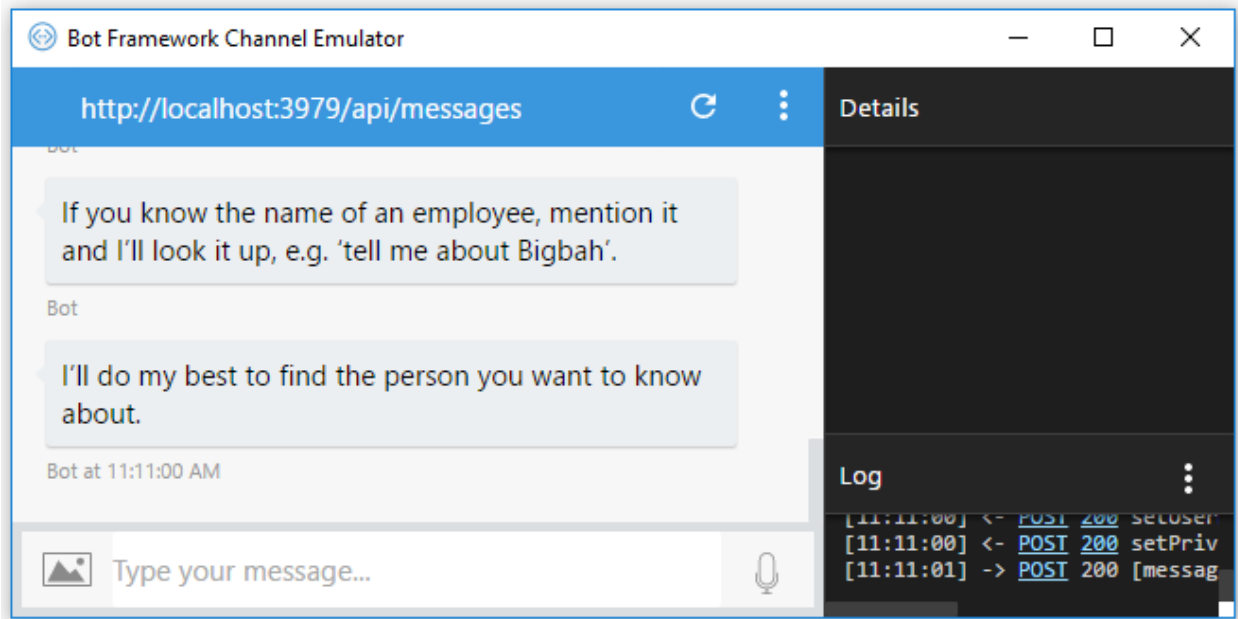
```
await context.PostAsync("Greetings from EmployeeBot.")
await Task.Delay(200)
await context.PostAsync("I'm here to tell you about employees.")
await Task.Delay(200)
await context.PostAsync("If you know the ID of an employee, type it in: e.g 'E2'.")
await Task.Delay(200)
await context.PostAsync("If you know the name of an employee, mention it and I'll look
it up, e.g. 'tell me about Bigbah'.")
await Task.Delay(200)
await context.PostAsync("I'll do my best to find the person you want to know about.")
return
```

This code will provide instructions to users, and tell them what the Bot can do. The `Task.Delay` statements provide a brief delay between each line of text, so the user won't be overwhelmed by seeing too much text at once. Let's see this in action.

Testing

- ☐ Select "Build > Build Solution" from the menu.
- ☐ Click on the Debugging start button to debug the solution.
- ☐ In the emulator, type "help".

The emulator should display the help message:



- ☐ In Visual Studio, stop debugging the Bot.

Building a Bot with Synergy .NET

Part 2: Defining Data

The help message said that the Bot would give data about employees. However, before it does that, it needs to have the data. So let's make some.

- ☐ In Solution Explorer, right-click on the EmployeeBot solution and click "Add > New Project...".
- ☐ In the "Add New Project" dialog box, select the "Synergy/DE > Windows" category, and select the "Console App (.NET Framework)" project type.
- ☐ Give the project the name "CreateDataFiles" and click OK.

We'll use this project to create and populate the data files. First, we need to define an Employee data structure that can be used in this project and in the bot itself. We can do that using a Repository project.

Repository Project

The Repository project type allows you to reference an existing Synergy Repository in a Synergy project, or even to build a new Repository from a schema file. The Repository project type is classified as a Traditional Synergy project, but Synergy .NET projects can reference a Repository project just like Traditional Synergy projects can. Follow these steps to create a new Repository:

- ☐ In Solution Explorer, right-click on the EmployeeBot solution and click "Add > New Project...".
- ☐ In the "Add New Project" dialog box, select the "Synergy/DE > Traditional" category, and select the "Synergy/DE Repository" project type.
- ☐ Give the project the name "Repository" and click OK.
- ☐ Open the Repository's schema file (repository.scm).
- ☐ In repository.scm, add the following Synergy Data Language code (also found in the Appendix on page 47):

```
Structure EMPLOYEE   DBL ISAM
  Description "Employee information"

Field EMPLOYEEID   Type AUTOSEQ   Size 8
  Description "ID"
  Readonly
  Nonnull

Field FIRSTNAME    Type ALPHA     Size 20
  Description "The employee's first name"

Field LASTNAME     Type ALPHA     Size 20
  Description "Employee's last name"

Field TITLE        Type ALPHA     Size 64
  Description "Job title"

Field IMAGE        Type ALPHA     Size 100
  Description "URL for a picture of the employee"

Key EMPLOYEEID     ACCESS   Order ASCENDING   Dups NO
  Description "ID"
  Segment FIELD    EMPLOYEEID   SegType SEQUENCE

Key FIRSTNAME      ACCESS   Order ASCENDING   Dups YES
```

Building a Bot with Synergy .NET

Description "The employee's first name"
Segment FIELD FIRSTNAME SegType ALPHA

Key LASTNAME ACCESS Order ASCENDING Dups YES
Description "Employee's last name"
Segment FIELD LASTNAME SegType ALPHA

Key TITLE ACCESS Order ASCENDING Dups YES
Description "Employee's job title"
Segment FIELD TITLE SegType ALPHA

- ☐ Build the Repository project.
- ☐ To ensure that the build was successful, go to Visual Studio's menu bar and click "Tools > Synergy/DE Repository".
- ☐ In the Repository program's menu bar, click "Modify > Structures". You should see the new EMPLOYEE structure:



STRUCTURE NAME	DESCRIPTION	TYPE
EMPLOYEE	Employee information	DBL

- ☐ Close the Repository program.

Now we have a Repository, so we need to make the other projects aware of it.

- ☐ In Solution Explorer, right-click on "References" inside the CreateDataFiles project, and click "Add Reference...".
- ☐ On the "Projects" tab of the Reference Manager, check the checkbox next to "Repository", and click OK.
- ☐ In Solution Explorer, right-click on "References" inside the EmployeeBot.Bot project, and click "Add Reference...".
- ☐ On the "Projects" tab of the Reference Manager, check the checkbox next to "Repository", and click OK.

Now the other projects can access the Repository, so we'll be able to start creating the data files.

Data Files

At this point, we have a Repository that will structure the data files. However, there are still a few set-up steps before we can create the data files themselves.

- ☐ In Solution Explorer, right-click on the EmployeeBot.Host project and click "Add > New Folder".
- ☐ Give the folder the name "Data".

This will create a folder that can hold the data files. The files don't have to be here for local testing. However, if you want to the Bot as an online web service, you should include the data files with the host project so they can be copied to the server.

- ☐ In Solution Explorer, right-click on the CreateDataFiles project, and click "Properties".

Building a Bot with Synergy .NET

- ☐ Go to the Common Properties tab, and make sure that the “Use common properties:” checkbox is checked.
- ☐ Add the variable “DAT” with the value “\$(SolutionDir)EmployeeBot\EmployeeBot.Host\Data”:

DAT	\$(SolutionDir)EmployeeBot\EmployeeBot.Host\Data
-----	--

- ☐ Save and close the current file.
- ☐ In Solution Explorer, right-click on the EmployeeBot.Bot project, and click “Properties”.
- ☐ Go to the Common Properties tab, and make sure that the “Use common properties:” checkbox is checked.
- ☐ Save and close the current file.

Now we can start building the data files.

- ☐ Open the Program.dbl file in the CreateDataFiles project.
- ☐ In Program.dbl, add the following code to the data division (directly under “main”):

```
.include "EMPLOYEE" repository, record="employee", END
```

- ☐ In Program.dbl add the following code to the procedural division (directly under “proc”):

```
data filename, string, "DAT:employees.ism"
data filechan, int

isamc(filename, 212, 3,
& "START=1, LENGTH=8, NAME=EMPLOYEEID, TYPE=SEQUENCE",
& "START=9, LENGTH=20, NAME=FIRSTNAME, TYPE=ALPHA, DUPS",
& "START=29, LENGTH=20, NAME=LASTNAME, TYPE=ALPHA, DUPS",
& "START=49, LENGTH=64, NAME=TITLE, TYPE=ALPHA, DUPS")
```

This code will create an empty file with the expected record size and keys.

Employee Records

Now we’ll have the program open the file it just created and fill it with data.

- ☐ Continuing to edit Program.dbl, add:

```
open(filechan, "U:I", filename)
```

- Store the record for Jodah Veloper:

```
employee.firstname = "Jodah"
employee.lastname = "Veloper"
employee.title = "Software Engineer"
employee.image = "http://www.jodahveloper.com/wp-content/uploads/2017/07/Jodah.png"
store(filechan, employee)
```

We’ll want to add records for the other employees:

- Bigbah Smann (CEO)
<http://www.jodahveloper.com/wp-content/uploads/2017/07/Bigbah.png>
- Manny Jurr (Product Development Manager)

Building a Bot with Synergy .NET

<http://www.jodahveloper.com/wp-content/uploads/2017/08/Manny.png>

- Mark Etting (Director of Communications)
<http://www.jodahveloper.com/wp-content/uploads/2017/07/Mark.png>
- Vi Sprezz (Senior Executive)
<http://www.jodahveloper.com/wp-content/uploads/2017/07/Vi.png>
- Connie Sultant (Professional Services Specialist)
<http://www.jodahveloper.com/wp-content/uploads/2017/07/Connie.png>
- Jodah Bot (Artificial Intelligence) < No image >

You could add the code for each employee manually, but it would be easier to just copy it from the Appendix.

- ☐ Copy the code for the remaining employees from the Appendix (on pages 36-37) and paste it into Program.dbl.
- ☐ If you didn't copy it already, add this line just before the endmain statement:

`close filechan`

- ☐ Build the solution.
- ☐ Right-click on the CreateDataFiles project in Solution Explorer, and click "Debug > Start new instance" to run the program. This will create and populate the files.

Part 3: First Name Basis

We now have the data file the bot will use to retrieve employee data. So let's write some code to access the file. While we eventually want the bot to parse the user's input and search on a variety of queries, we'll start by handling a basic case. We'll let the user enter the first name of an employee, and we'll return any records that match.

Data Definitions

In order to access the file, the program will need to know where the file is and what type of data it contains. We could code this information into each source file that needs it, but it would be easier to put it all in one location.

- ☐ In Solution Explorer, right-click on the EmployeeBot.Bot project and click "Add > New Item...".
- ☐ Select the "Text File" type.
- ☐ Enter the name "DataDefinitions.def" and click Add.
- ☐ In DataDefinitions.def add the following code:

```
.include "Employee" repository, structure="emp_struct", end
.define DATAFILE, "DAT:employees.ism"
```

This will tell the program where to locate the employees.ism file, and will let it load the data into variables based on the emp_struct structure. We'll want to include this definition file in each source file that needs access to the file, but we need to do some setup first.

- ☐ In Solution Explorer, right-click on the EmployeeBot.Bot project, and click "Properties".
- ☐ Go to the Common Properties tab, and add the variable "DEF" with the value "\$\$(SolutionDir)EmployeeBot\EmployeeBot.Bot":

DAT	\$(SolutionDir)EmployeeBot\EmployeeBot.Host\Data
DEF	\$(SolutionDir)EmployeeBot\EmployeeBot.Bot

- ☐ Switch to the Compile tab and click the checkbox next to "Generate warnings instead of errors for duplicate structures and enumerations". This will allow the structure definition to be included in multiple files without causing build errors.
- ☐ Save and close the current file.

Select Statement

- ☐ In Solution Explorer, right-click on the "Dialogs" folder under the EmployeeBot.Bot project, and click "Add > Class...".
- ☐ Give the file the name "GetEmployeeByFirstName".

This will create a class named "GetEmployeeByFirstName", but we actually want a partial class with a static method by that name. That way we can expand the class by adding new source files, each with its own method to access the data in different ways.

- ☐ In GetEmployeeByFirstName.dbl, find the line that defines the class, and edit it to say:

```
public partial class RetrieveData
```

Building a Bot with Synergy .NET

The class imports three namespaces by default, but we'll want to add two more: "System.Collections" so the method can return an ArrayList, and "Synergex.SynergyDE.Select" so the method can use the Select class.

- ☐ At the top of GetEmployeeByFirstName.dbl, just after the other import statements, add these lines:

```
import System.Collections
import Synergex.SynergyDE.Select
```

- ☐ Move a little lower in the file, after the namespace definition and before the class definition, and add a reference to the definition file we made earlier:

```
.include "DEF:DataDefinitions.def"
```

- ☐ Now go inside the RetrieveData class (just before endclass) and add the following:

```
public static method GetEmployeeByFirstName, @ArrayList
    employeeName, string
endparams
proc
    data arraylist = new ArrayList()

    mreturn arraylist
endmethod
```

Once we're done coding it, this method will accept an employee's first name, search for the name in the data file, and return any matching records. This is a static method, so it can be called without instantiating the RetrieveData class.

It returns an ArrayList because it might find more than one record to match the input name, so it requires a collection to hold the results. ArrayList is a good collection type to use in this case, because it can work with non-CLS structures (such as Repository structures).

Let's add the code to search for the name.

- ☐ Between the "data arraylist..." and "mreturn arraylist" lines, add the following:

```
data emp_rec, emp_struct
disposable data fobj, @From, new From(DATAFILE,emp_rec)
disposable data sobj, @Select, new Select(fobj,
&      (NoCaseWhere)(emp_rec.firstname .eqs. employeeName))
```

That will create a record to hold an employee's data, a From object pointing to the data file, and a Select object that will find records where the "firstname" field matches the employeeName parameter. The Select object uses NoCaseWhere so it won't be case-sensitive when matching records. Note that the From and Select objects are defined as "disposable". That will cause the garbage collector to dispose of those objects, which can help the program to use memory more efficiently. That doesn't matter much if you're just testing the bot locally, but it could have a significant impact on a bot running as a web service with many simultaneous users.

Building a Bot with Synergy .NET

This Select object will return all of the records with a firstname field that matches the search string. Now we just have to put those records in an ArrayList and return them to the calling routine. We can do this with a foreach loop.

- ☐ Continuing to edit GetEmployeeByFirstName.dbl (right after the code you just added and before the mreturn line), add the following:

```
foreach emp_rec in sobj
begin
    arraylist.Add(emp_rec)
end
```

Process Query

Now that we have a method to search the employee records, let's call it.

- ☐ At the top of ProcessQuery.dbl (right after the other import statements) add:

```
import System.Collections
```

- ☐ Inside the ProcessQueryAsync method, add the following code right after the proc statement:

```
data EmployeeList, @ArrayList, new ArrayList()
data isSearchOver, boolean, false
```

EmployeeList is an ArrayList we'll use to hold all matching records, and isSearchOver is a flag we'll use later on to help eliminate irrelevant results.

- ☐ Jump ahead in the method to after the begin/end block, and delete the line that says "await context.PostAsync(msgtext)".
- ☐ In its place, add the following:

```
;Search by firstname
EmployeeList = RetrieveData.GetEmployeeByFirstName(msgtext)
```

Send Response

You'll notice that we removed the call to "context.PostAsync" which previously echoed back the user's input. That's because the ProcessQueryAsync method would get too big if it included the code to send responses to the user as well as code to process the user's input and gather the results. Instead, we'll create a new method in a new source file that will handle most responses.

- ☐ In Solution Explorer, right-click on the Dialogs folder inside the EmployeeBot.Bot project and click "Add > New Item..."
- ☐ Select the "Code File" item type.
- ☐ Type in "SendResponse.dbl" as the file's name, and click Add.
- ☐ Add the following import statements at the top of SendResponse.dbl:

```
import System.Collections
import System.Threading.Tasks
import Microsoft.Bot.Builder.Dialogs
import Microsoft.Bot.Connector
```

- ☐ Add the EmployeeBot.Bot.Dialogs namespace and include the definitions file:

Building a Bot with Synergy .NET

```
namespace EmployeeBot.Bot.Dialogs

    .include "DEF:DataDefinitions.def"

endnamespace
```

- ☐ The new method will still be part of the partial MessagesDialog class, so inside the namespace in SendResponse.dbl add the class definition:

```
public partial class MessagesDialog implements IDialog<object>

endclass
```

- ☐ Inside the class, add a definition for the SendResponseAsync method:

```
public async method SendResponseAsync, @Task
    context,                @IDialogContext
    msgtext,                string
    EmployeeList,           @ArrayList
endparams

proc

    mreturn
endmethod
```

- ☐ Now add the following code after proc and before mreturn:

```
data emp_rec, emp_struct

;Loop through employee records and display them
foreach emp_rec in EmployeeList
begin
    await context.PostAsync(emp_rec.firstname + " " + emp_rec.lastname +
        & " (E" + %string(emp_rec.employeeid)+ ")")
    await Task.Delay(200)
end
```

That code will send out basic information about any employees in the EmployeeList. Now that we have a method that will send a response to the user, we just need to call it.

- ☐ Go back to the ProcessQueryAsync method (ProcessQuery.dbl), and add the following code just before mreturn:

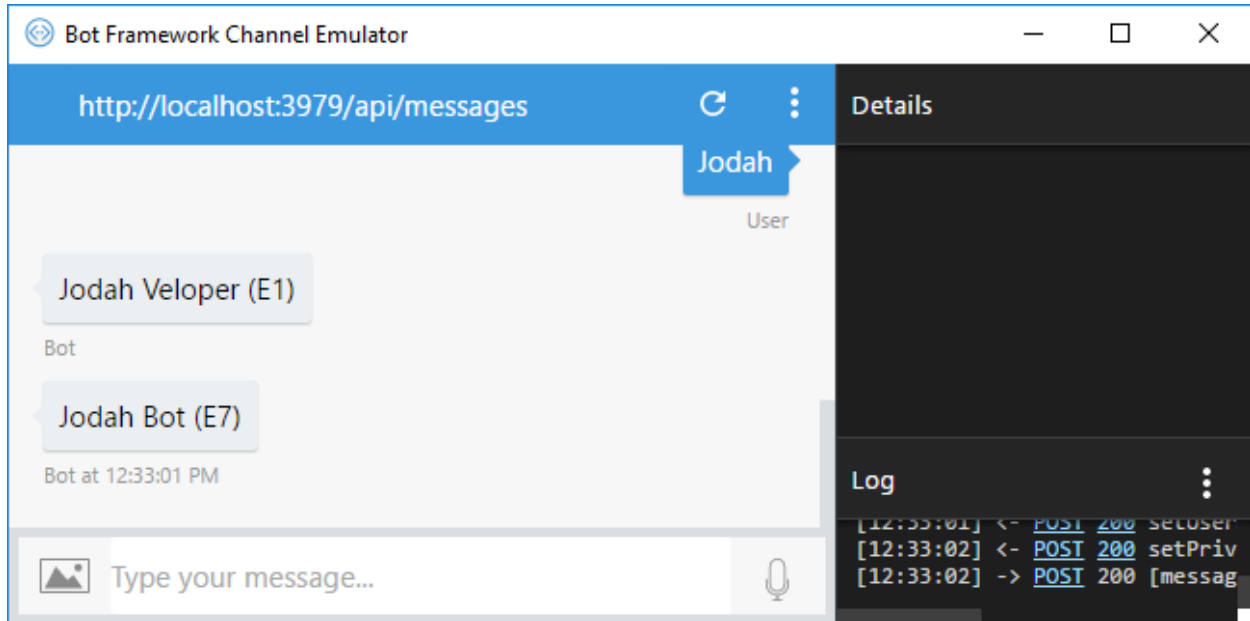
```
await SendResponseAsync(context, msgtext, EmployeeList)
```

Testing

- ☐ Build the solution and start debugging.
- ☐ In the emulator, type: "Jodah" (without the quotation marks).

The emulator should return a response for "Jodah Veloper" and "Jodah Bot":

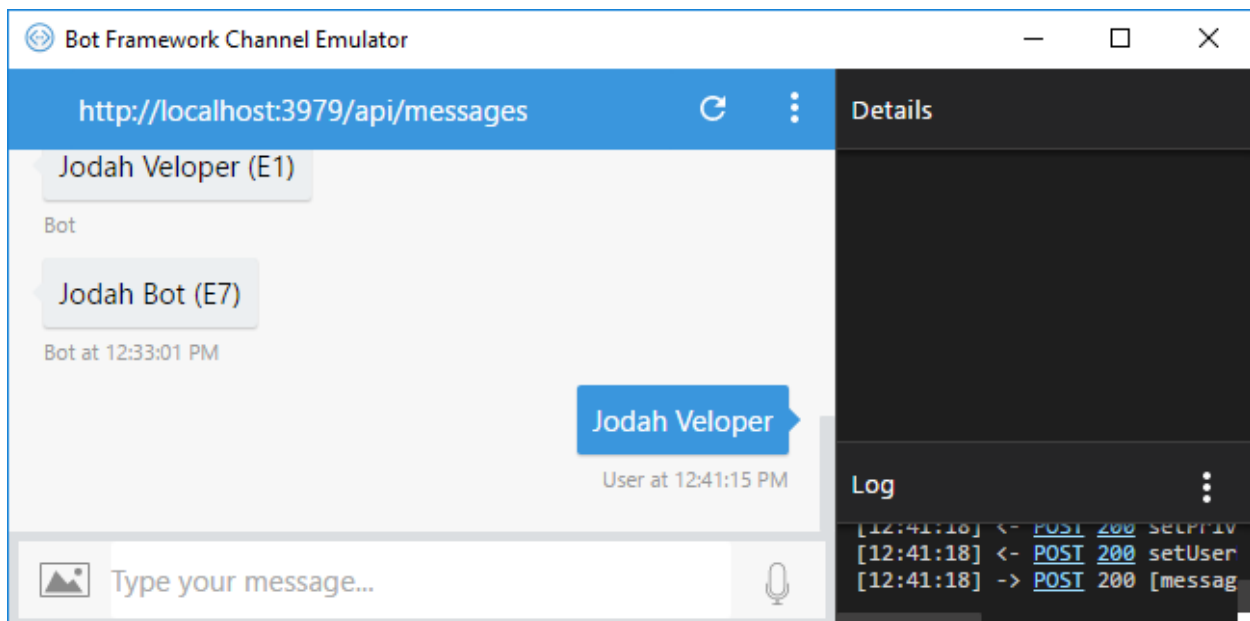
Building a Bot with Synergy .NET



Right now, the bot will only return a result if the query is an exact match for the first name of an employee. For example, it will not return a result for “Jodah Veloper”, because there is no employee record with the first name of “Jodah Veloper”.

- ☐ In the emulator, type “Jodah Veloper”.

The emulator won’t return anything:



In a later step, we’ll fix this so that a search for “Jodah Veloper” will return the correct employee. For now, let’s modify the bot to send a message when it can’t find anything, so the user won’t have to wonder if the bot’s just ignoring the query.

- ☐ In Visual Studio, stop debugging the program.

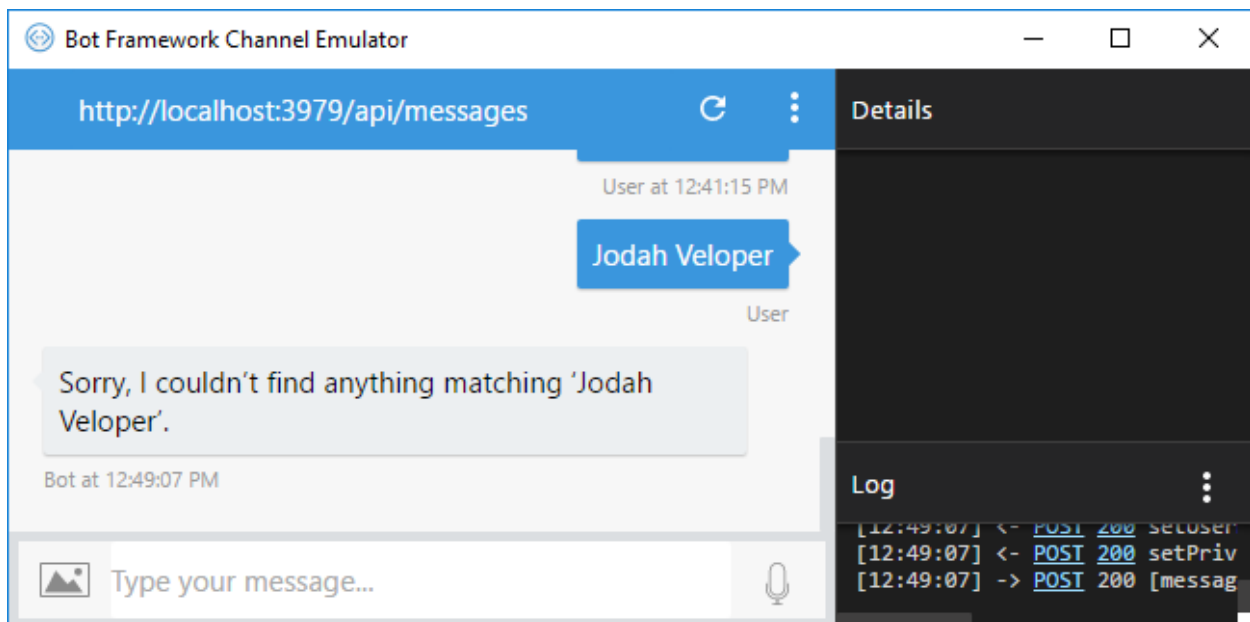
Building a Bot with Synergy .NET

- ☐ In `SendResponse.dbl`, go to the top of the `SendResponseAsync` method (just below the proc statement) and add the following code:

```
if (EmployeeList.count == 0)
begin
    await context.PostAsync("Sorry, I couldn't find anything matching '" +
        & msgtext + "'")
    mreturn
end
```

- ☐ Build the solution and start debugging.
- ☐ In the emulator, type: "Jodah Veloper".

This time, the emulator will show the bot's response, indicating that it's working, but that it can't find what the user wants:



- ☐ In Visual Studio, stop debugging.

Part 4: By the Numbers

In the previous part, we saw that some queries could result in multiple records. This makes sense because multiple employees can have the same first name; the `firstname` field is not unique. However, the `employeeid` field is defined as a sequence autokey, and is therefore unique. Each ID number matches one and only one employee. In this part of the tutorial, we'll modify the bot to search by employee ID.

Select Statement

- ☐ In Solution Explorer, right-click on the "Dialogs" folder under the EmployeeBot.Bot project, and click "Add > New Item...".
- ☐ Select the "Code File" type.
- ☐ Give the file the name "GetEmployeeByID".
- ☐ Click the "Add" button.

This file will contain another method with code similar to the `GetEmployeeByFirstName` method you created in the previous section.

- ☐ Copy all of the code in `GetEmployeeByFirstName.dbl` and paste it into `GetEmployeeByID.dbl`.
- ☐ In `GetEmployeeByID.dbl`, change the method's name to `"GetEmployeeByID"`.
- ☐ In the data division, change the parameter's name from `"employeeName"` to `"employeeID"`, and change its type from `"string"` to `"int"`.
- ☐ In the definition for the `Select` object, delete the line starting with `"& (NoCaseWhere)"`, and in its place add:

```
& (Where)(emp_rec.employeeid .eq. employeeID))
```

Process Query

Now we want to call this method when the user uses an employee's ID number in the query. We could use the same sort of code we used before, and only return a record when the user's query exactly matches an employee's ID. Instead, we'll use regular expressions to match a wider range of natural-language queries, like "who is employee e1?".

You may have noticed in the previous section that the bot uses the form of "E#" to report an employee's ID number, e.g. "Jodah Veloper (E1)". Let's assume that the user will use the same format when searching for an employee by ID. So if we detect the letter 'E' followed by an integer, we'll extract the number and use it to search for an employee.

- ☐ In `ProcessQuery.dbl`, add the following code after the code for the help message and before the call to `GetEmployeeByFirstName`:

```
;Search employees by ID [match E#]
if (Regex.IsMatch(msgtext, "E\d+", RegexOptions.IgnoreCase))
begin
    data eid, int, ExtractEmployeeID(msgtext)
    EmployeeList = RetrieveData.getEmployeeByID(eid)
    if (EmployeeList.Count > 0) isSearchOver = true
end
```

Building a Bot with Synergy .NET

This code checks the input text for a match to the regular expression “E\d+” (a capital ‘E’ followed by one or more digits). If that returns true, we’ll want to extract the number so we can use it as the search parameter. The code does this with a call to “ExtractEmployeeID”, but that method doesn’t exist yet. So let’s create it.

- ☐ Near the end of ProcessQuery.dbl (after the end of the ProcessQueryAsync method, but still inside the MessagesDialog class), add a new method definition:

```
;Extract employee ID from messagetext
private method ExtractEmployeeID, int
    inputstring, string
endparams
proc
    data messagetext = inputstring
    messagetext = Regex.Match(messagetext, "E\d+", RegexOptions.IgnoreCase).Value
    messagetext = Regex.Replace(messagetext, "E", "", RegexOptions.IgnoreCase)
    data eid, int
    eid = messagetext
    mreturn eid
endmethod
```

This method matches the regular expression “E\d+”, removes everything but the number, and returns the number as an integer. We now have code that will match a regular expression, find a number, and use the number to find and return an employee. There just one more thing to do before the code for this part of the tutorial is complete.

Currently, the code in ProcessQueryAsync includes calls to both GetEmployeeByFirstName and GetEmployeeByID. Since an employee’s ID number is unique, we can assume that a user searching by ID wants that particular employee (as opposed to a search by first name, where a user searching for “Jodah” may want “Jodah Veloper” or “Jodah Bot”). Therefore, if the user’s input included an ID number that matches an employee, we should stop the search there and not check whether a first name search also results in a match.

- ☐ Near the end of the ProcessQueryAsync method, locate the line that says “EmployeeList = RetrieveData.GetEmployeeByFirstName(msgtext)”.
- ☐ At the start of the line, add a test for “isSearchOver” being false, keeping the rest of the code as it is:

```
if (!isSearchOver) EmployeeList = RetrieveData.GetEmployeeByFirstName(msgtext)
```

Because the code that matches an employee ID sets “isSearchOver = true” if it finds any results, the bot will now skip the first name search if there’s a match for the employee ID. This isn’t really necessary right now, since no employee has a name like “E1”, so if any query matched on ID, it wouldn’t match on first name. However, skipping the name search can save some processing, and can be useful in the next part of the tutorial when the queries will become more complicated.

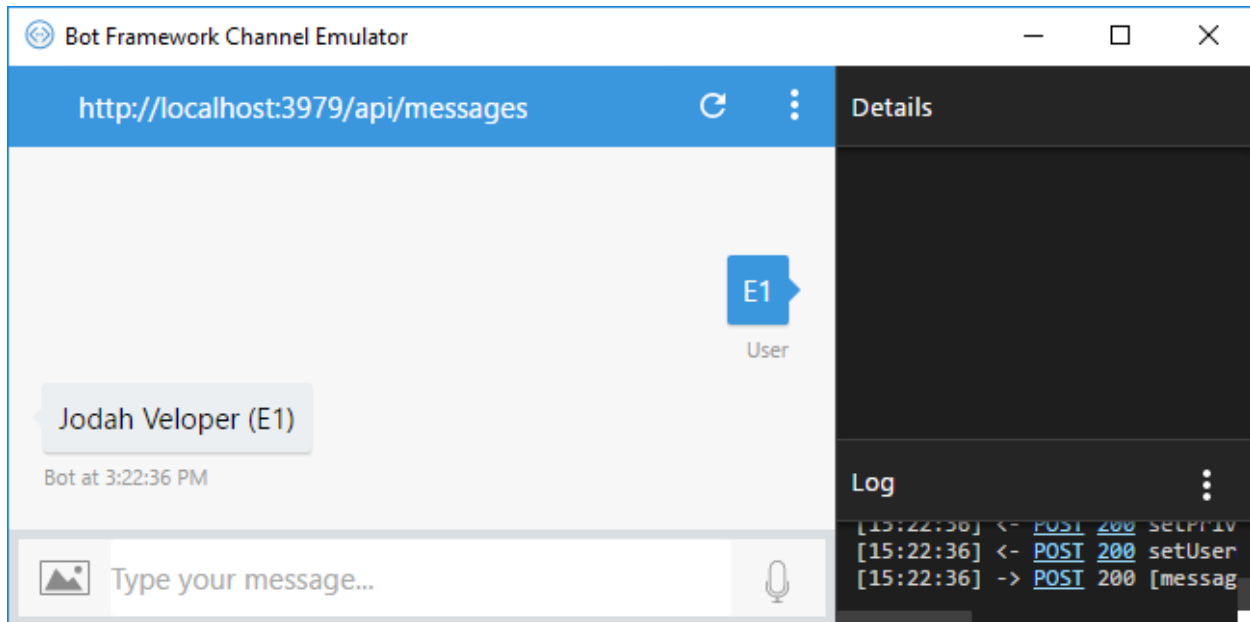
Now let’s test the bot.

Testing

- ☐ Build the solution and start debugging.
- ☐ In the emulator, enter “E1”.

Building a Bot with Synergy .NET

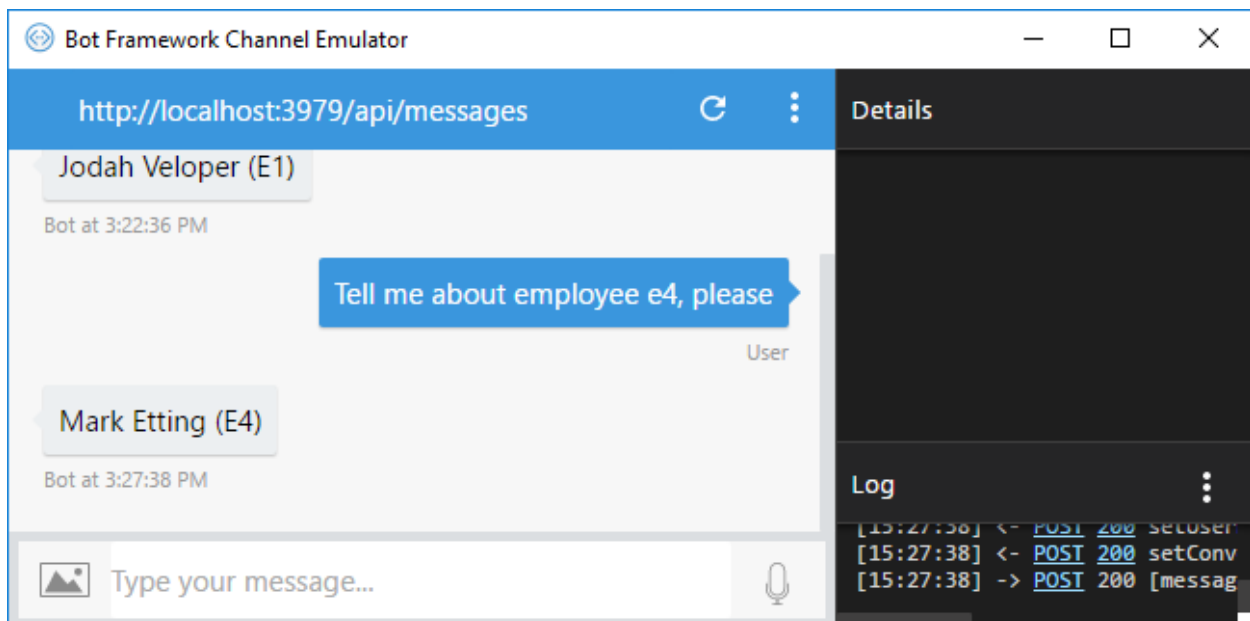
The emulator should find one result:



Because the code matches a regular expression instead of a verbatim search of the user input, you can use a more complicated query and still get the right answer.

- ☐ In the emulator, type “Tell me about employee e4, please”.

The emulator will return with a match for employee E4:



- ☐ In Visual Studio, stop debugging.

Part 5: By Any Other Name

So far, the bot can search employees by first name and ID number. There are two more keys in employee records: last name and job title. In this part of the tutorial, we'll add code to search by those keys as well. We need to make some other improvements to the code as well.

The first name search only returns exact matches. If we just extended that to last names and job titles, then we would get a result only if the first name is an exact match, if the last name is an exact match, or if the job title is an exact match. However, we couldn't include both first and last names in the same query. A query like "Who is Vi Sprezz" would return zero results, because that query as a whole isn't the first name, last name, or job title of any employee.

What we want it to do is break the query down into individual words, and search with the words instead of the entire query.

More Select Statements

First of all, let's add new methods to search by last name and job title. Once again, the `GetEmployeeByFirstName` class can serve as the template.

- ☐ In Solution Explorer, right-click on the "Dialogs" folder under the EmployeeBot.Bot project, and click "Add > New Item...".
- ☐ Select the "Code File" type.
- ☐ Give the file the name "GetEmployeeByLastName".
- ☐ Click the "Add" button.
- ☐ Copy all of the code in `GetEmployeeByFirstName.dbl` and paste it into `GetEmployeeByLastName.dbl`.
- ☐ In `GetEmployeeByLastName.dbl`, change the method's name to "GetEmployeeByLastName",
- ☐ In the definition for the Select object, replace "emp_rec.firstname" with "emp_rec.lastname".

We can search by last name now, so let's complete the set with a job title search.

- ☐ In Solution Explorer, right-click on the "Dialogs" folder under the EmployeeBot.Bot project, and click "Add > New Item...".
- ☐ Select the "Code File" type.
- ☐ Give the file the name "GetEmployeeByTitle".
- ☐ Click the "Add" button.
- ☐ Copy all of the code in `GetEmployeeByFirstName.dbl` and paste it into `GetEmployeeByTitle.dbl`.
- ☐ In `GetEmployeeByTitle.dbl`, change the method's name to "GetEmployeeByTitle".
- ☐ In the data division, change the parameter's name from "employeeName" to "employeeTitle".
- ☐ In the definition for the Select object, delete the line starting with "& (NoCaseWhere)", and in its place add:

```
& NoCaseWhere.Contains(emp_rec.title,employeeTitle))
```

Notice that this Select statement uses `NoCaseWhere.Contains` instead of an equality test like the other methods. This is necessary because some employees have multi-word job titles. When we start breaking down queries into individual words, those job titles would never get a match (e.g. "Software Engineer"

Building a Bot with Synergy .NET

would not match “Software” or “Engineer”). Using “Contains” will allow us to match those values as expected.

Tokenize Queries

Now we can add the code to tokenize the query – to break it down into individual words.

- ☐ Near the end of ProcessQuery.dbl (after the last endmethod statement, but still inside the class), add the following method definition:

```
;Split the message text into tokens
private method tokenizeQuery, @ArrayList
    msgtext,      string
endparams

proc
    data token, string
    data tokens, [#]string
    data tokenlist, @ArrayList, new ArrayList()

    mreturn tokenlist
endmethod
```

- ☐ Just before the mreturn statement, add the following code:

```
data separators, [#]char, new char[#] {" ", ",", ".", "?"}
tokens = msgtext.Split(separators)
```

That will define a set of common separators, and will use a method from the .NET string class to split up the input string into the tokens between those separators. The text might be hard to read, so if you aren’t copying and pasting this code, the separators are:

- A space,
- A comma,
- A period, and
- A question mark.

- ☐ Below the code you just added (just before the mreturn statement), add the following:

```
foreach token in tokens
begin
    if (!Regex.IsMatch(token, "E\d+", RegexOptions.IgnoreCase))
    begin
        tokenlist.Add(token)
    end
end
```

This code will filter out any token that matches the “E\d+” regular expression, since we deal with those elsewhere. All other tokens are added to the list. If you wanted, you could extend this to filter out common words like “the”, that users are likely to type, but that won’t return any useful results. However, we’ll leave it like this for the purposes of this tutorial.

Building a Bot with Synergy .NET

Putting it Together

Now that we have the code to search by last name and title, and code to tokenize a query, let's put everything together and see how it works.

- ☐ Go to the ProcessQueryAsync method (in ProcessQuery.dbl) and edit the comment “;Search by firstname” to say “;Search by name or title”.
- ☐ Directly below that comment is an “if” statement. Delete everything on that line after “(!isSearchOver)”.
- ☐ Below the “if” statement, add a begin/end block. The code should look like this:

```
;Search by name or title
if (!isSearchOver)
begin

end
```

- ☐ Inside the begin/end block, add the following code:

```
data tokenList, @ArrayList, new ArrayList()
tokenList = tokenizeQuery(msgtext)
data token, string
foreach token in tokenList
begin
    EmployeeList.AddRange(RetrieveData.GetEmployeeByFirstName(token))
    EmployeeList.AddRange(RetrieveData.GetEmployeeByLastName(token))
    EmployeeList.AddRange(RetrieveData.GetEmployeeByTitle(token))
end
if (EmployeeList.Count > 0) isSearchOver = true
```

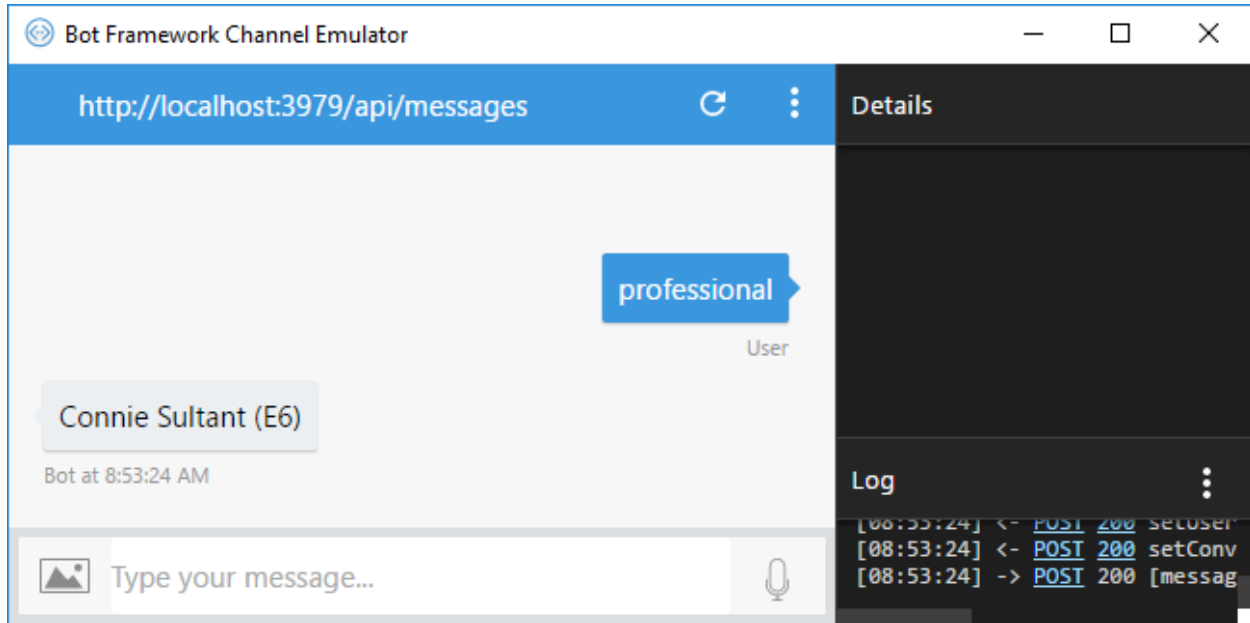
This code will tokenize the input query, and check each word for matches on first name, last name, and job title. Note that the code uses the AddRange method to combine all results into a single ArrayList. If we used a simple assignment operator like we did before, the EmployeeList would be replaced whenever one of the methods was called, likely resulting in the loss of some matching records.

Testing

- ☐ Build the solution and start debugging.
- ☐ In the emulator, type “professional”.

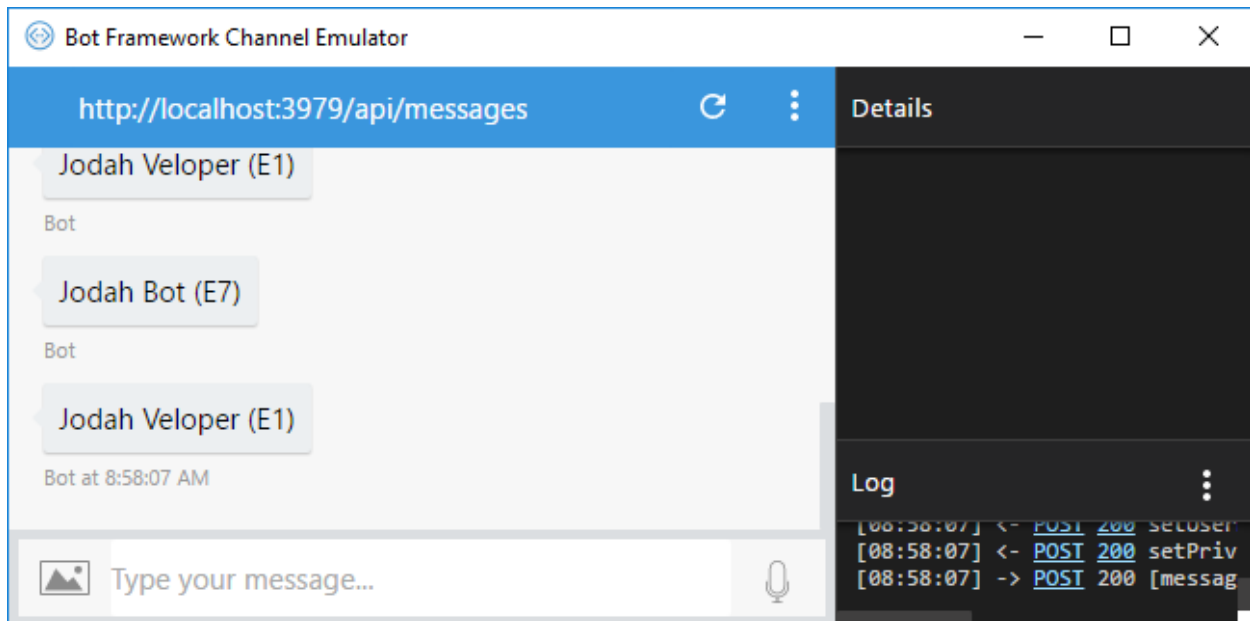
The bot will respond with Connie Sultant, since her job title is “Professional Services Specialist”:

Building a Bot with Synergy .NET



- ☐ Now type in “Jodah Veloper”.

The bot will give three results: “Jodah Veloper”, “Jodah Bot”, and “Jodah Veloper” again:



This is certainly progress over Part 3, where a query for “Jodah Veloper” returned no results. However, it still isn’t what we want or expect (i.e. that a query for “Jodah Veloper” would return a single result for Jodah Veloper). In the next part of the tutorial, we’ll work on fixing this problem

- ☐ In Visual Studio, stop debugging.

Part 6: Eliminating Duplicates

The current code returned three results to a query for “Jodah Veloper”. That’s because it broke down the query into “Jodah” and “Veloper” and tried to find matches for both search terms. For “Jodah”, it found first name matches for Jodah Veloper and Jodah Bot, so it added those results to the list. For “Veloper”, it found a last name match for Jodah Veloper, so it added that record to the list again. This naïve approach can help test our code while developing the bot, but we wouldn’t want to use it in the final version.

There are two specific improvements we would want to make:

1. Show each unique record no more than once. Eliminate duplicate results.
2. Assume that records that match multiple terms are more likely to be relevant than records that only match a single term. Eliminate less relevant results.

The second improvement may not always apply; if we were building a search engine, for example, we may want to keep less relevant results, but give them lower priority than other results. However, for this bot we’ll assume the user wants to find a single employee record, and we’ll only show multiple records if we can’t tell which one most closely matches the query.

In this part of the tutorial, we’ll implement an algorithm that will make the two improvements listed above. We’ll leave the `ProcessQueryAsync` method as it is. The `SendResponseAsync` method will receive the full list of results, and then process it to match our criteria before sending a response to the user.

- ☐ In the `SendResponseAsync` method (`SendResponse.dbl`), find the line that says “`data emp_rec, emp_struct`”.
- ☐ Beneath that line, add:

```
data generalEmployeeList, @ArrayList, new ArrayList()  
data generalIDList, @ArrayList, new ArrayList()  
data relevantEmployeeList, @ArrayList, new ArrayList()  
data relevantIDList, @ArrayList, new ArrayList()
```

These `ArrayList`s are two tiers of results. The algorithm will loop through all results and add them to the `generalEmployeeList` (tier 1). If it encounters a result that’s already in the `generalEmployeeList`, it will instead add it to the `relevantEmployeeList` (tier 2). Once the loop is complete, there will be one of two outcomes:

1. The `relevantEmployeeList` is empty, which means that no record matched more than one search term, and any results in the `generalEmployeeList` should be returned to the user. Or,
2. The `relevantEmployeeList` contains one or more records. These records match at least two search terms, and should be returned to the user (instead of `generalEmployeeList`, which may contain results that only match one term).

We could extend this algorithm to account for records that match three search terms, but to keep things simple we’ll just treat those the same as all other relevant records.

The algorithm uses separate `Employee` and `ID` lists, because it’s easier to test an integer `ArrayList` to see if it contains a specific integer than it would be to test an `ArrayList` of records.

Building a Bot with Synergy .NET

- ☐ Below the ArrayList definitions you just added (in SendResponseAsync), add the following code:

```
;filter employee list to remove duplicates
foreach emp_rec in EmployeeList
begin
    if (generalIDList.Contains(emp_rec.employeeid)) then
    begin
        ;If the current employee record is already in the general list,
        ;then it's a duplicate that matched multiple search terms.
        ;Add it to the relevant list, unless it's there already.
        if (relevantIDList.Contains(emp_rec.employeeid) == false)
        begin
            relevantEmployeeList.Add(emp_rec)
            relevantIDList.Add(emp_rec.employeeid)
        end
    end
    else
    begin
        ;If it isn't in the general list already, add it.
        generalEmployeeList.Add(emp_rec)
        generalIDList.Add(emp_rec.employeeid)
    end
end
```

That will loop through the results and populate the two tiers, as explained above. Now we need to check whether the relevantEmployeeList has any records, and proceed accordingly.

- ☐ Add the following code:

```
;if there aren't any records more relevant than others,
;treat all records as relevant
if (relevantIDList.count == 0)
begin
    relevantIDList = generalIDList
    relevantEmployeeList = generalEmployeeList
end
```

Now just modify the code to show the relevant records instead of all the records.

- ☐ Find the line that says “`foreach emp_rec in EmployeeList`” (shortly below the code you just added), and modify it to say:

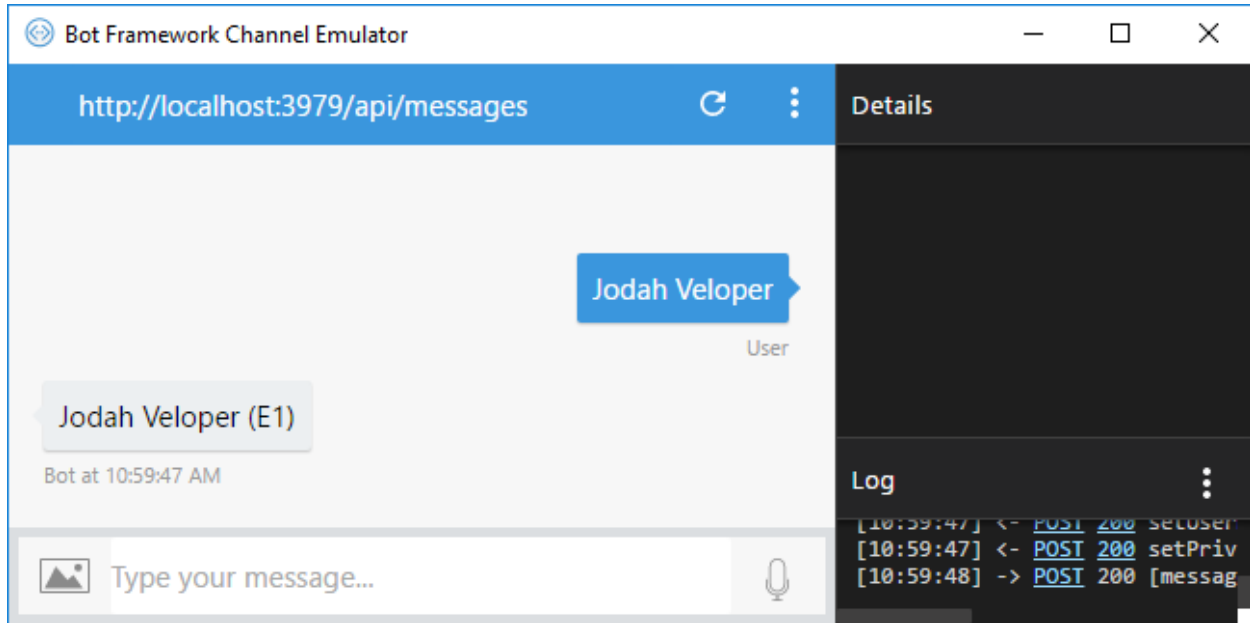
```
foreach emp_rec in relevantEmployeeList
```

Testing

- ☐ Build the solution and start debugging.
- ☐ In the emulator, type “Jodah Veloper”.

This time, the emulator will show the one record we want:

Building a Bot with Synergy .NET



☐ In Visual Studio, stop debugging.

The bot now has complete functionality. It can search employees by ID, name, or job title. It doesn't get confused by multi-word queries, or by extraneous words that don't work as search terms. Also, it knows how to eliminate duplicate records from its results and only show relevant records.

However, we want the bot to give some more information about the employees than their names and ID numbers. In the next section, we'll modify the way that the bot displays employee records.

Part 7: Going Into Detail

The bot currently displays employee records by returning the first and last name and the ID number. However, the employee records contain two additional fields: title and image. We'll want the bot to include both of these fields in its response.

We assume that the user only wants to see a single employee record at a time. If a user's query results in multiple matching records, we don't want to fill the screen with details about all of the matching employees. Instead, we want to show the user a list of matching employees, and prompt the user to search by ID to find a specific match.

So we'll modify the code to send different responses depending on whether there is only one relevant record or more than one.

- ☐ Near the end of the `SendResponseAsync` method (`SendResponse.dbf`), locate the `foreach` loop that starts with `"foreach emp_rec in relevantEmployeeList"`.
- ☐ Immediately above the loop, add the following code:

```
if (relevantEmployeeList.count > 1) then
begin
    ;there's more than one record, so just show basic information
```

- ☐ Indent the `foreach` loop so that it lines up with the comment you just added, to indicate that it's inside the `begin/end` block for the `if` statement.
- ☐ After the `foreach` loop ends, add the following:

```
    await context.PostAsync(
        & "For more details, search for an employee ID, e.g. 'E' +
        & %string(emp_rec.employeeid) + '"")
end
```

This code will handle the case when there are multiple matching records. Now we'll handle the case where there's a single relevant record, so we can go into more detail. We don't need to check whether the relevant list has zero or one record, because we handled the case of zero matching records previously, so the code could only get to this point if there is a single relevant record.

- ☐ Below the code you just added, add:

```
else
begin
    ;there's only one record, so show more details
    foreach emp_rec in relevantEmployeeList
    begin
        await context.PostAsync(emp_rec.firstname + " " + emp_rec.lastname +
            & " (E" + %string(emp_rec.employeeid)+ ")")
        await Task.Delay(200)
        await context.PostAsync(emp_rec.title)
        await Task.Delay(200)
    end
end
```

Building a Bot with Synergy .NET

Even though the relevantEmployeeList only contains a single record, a foreach loop is still the easiest way to access the record. This loop displays the employee's first name, last name, and ID number, as before. Then, after a brief pause, it displays the employee's job title.

Picture This

The final change we want to make is for the bot to show a picture of the employee. You may recall that not every employee record included an image URL. So we'll want to check whether a picture exists before we try to display it.

- ☐ Just after the second `await Task.Delay(200)` you added above (right before the end of that foreach loop), add the following:

```
;Show picture, if applicable
if (%atrim(emp_rec.image) .nes. "")
begin

end
```

So far, we've had the bot send responses by calling `context.PostAsync` with a string argument. That won't work when we want to show an image. Instead, we can take advantage of some functionality provided by the bot framework to make a custom message to send back to the user.

- ☐ Inside the begin/end block you created above, add the following code:

```
data message1 = context.MakeMessage()
message1.Attachments.Add(new Attachment() {
    &    ContentUrl = %atrim(emp_rec.image),
    &    ContentType = "image/png",
    &    Name = %atrim(emp_rec.firstname)})
await context.PostAsync(message1)
```

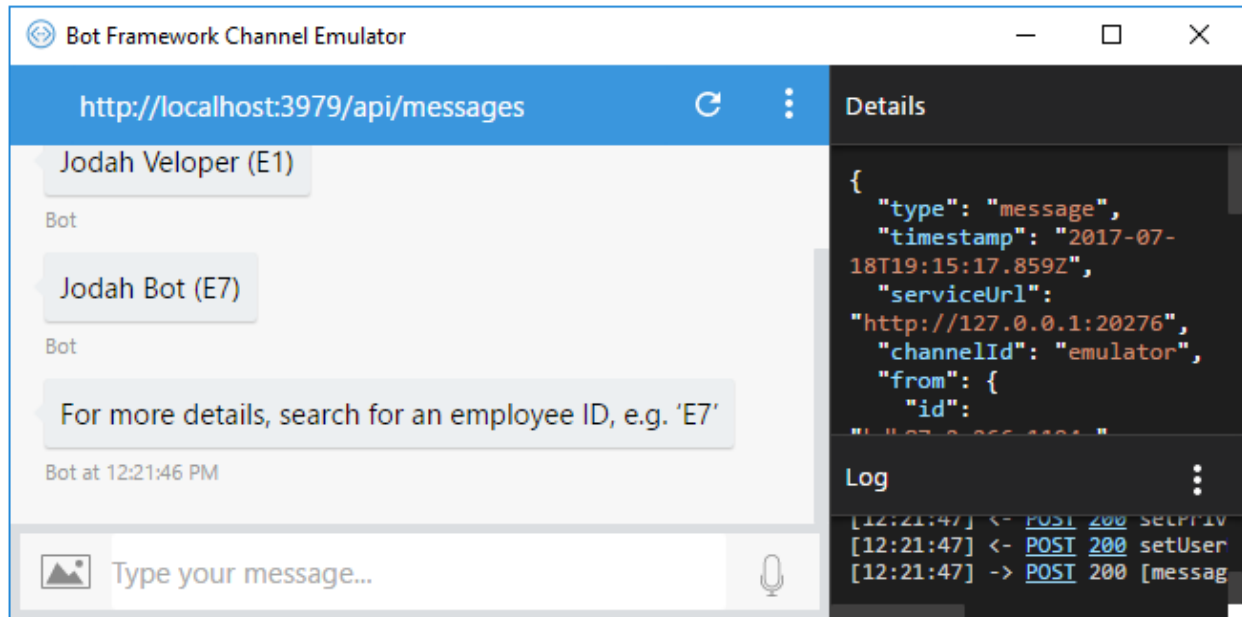
That code will create a message with an image attachment, using the employee's first name as the caption.

Testing

- ☐ Build the solution and start debugging.
- ☐ In the emulator, type "Jodah".

The bot will show a list of the two results, and prompt you to search by ID number:

Building a Bot with Synergy .NET



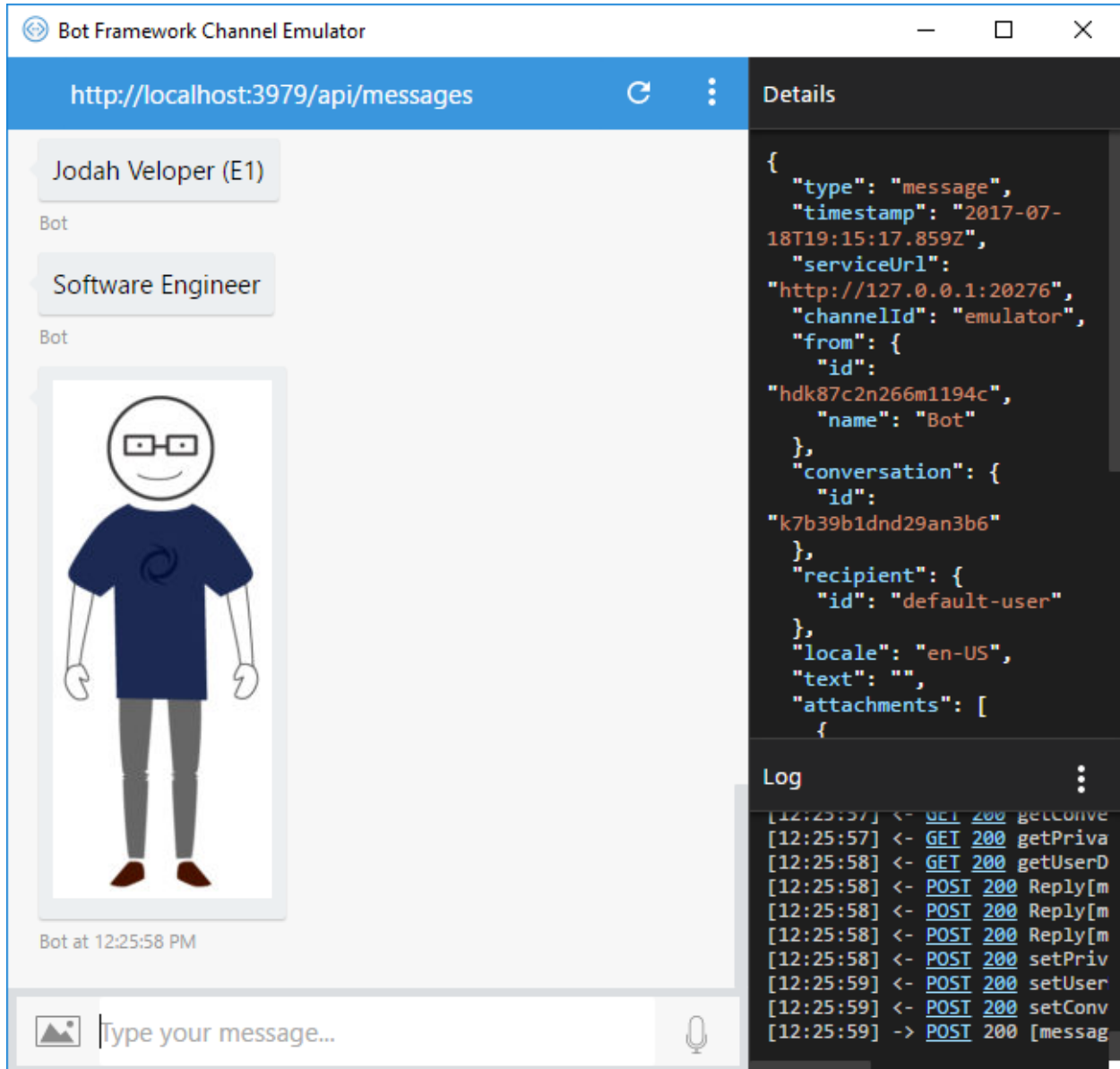
☐ Type in "E7".

The bot will give details about Jodah Bot, but won't try to show a picture because the image field in the record is blank.

☐ Enter "E1".

The bot will show details and a picture for Jodah Veloper:

Building a Bot with Synergy .NET



The screenshot shows the Bot Framework Channel Emulator window. The title bar reads "Bot Framework Channel Emulator". The address bar shows "http://localhost:3979/api/messages". The chat interface on the left shows a conversation with "Jodah Veloper (E1)" and "Software Engineer". A bot profile card is displayed with a cartoon character wearing glasses and a blue t-shirt with a Synergy logo. The details panel on the right shows a JSON message object. The log panel at the bottom right shows a series of HTTP requests and responses.

Bot Framework Channel Emulator

http://localhost:3979/api/messages

Jodah Veloper (E1)

Bot

Software Engineer

Bot

Bot at 12:25:58 PM

Type your message...

Details

```
{
  "type": "message",
  "timestamp": "2017-07-18T19:15:17.859Z",
  "serviceUrl": "http://127.0.0.1:20276",
  "channelId": "emulator",
  "from": {
    "id": "hdk87c2n266m1194c",
    "name": "Bot"
  },
  "conversation": {
    "id": "k7b39b1dnd29an3b6"
  },
  "recipient": {
    "id": "default-user"
  },
  "locale": "en-US",
  "text": "",
  "attachments": [
  ]
}
```

Log

```
[12:25:57] <- GET 200 getConve
[12:25:57] <- GET 200 getPriva
[12:25:58] <- GET 200 getUserD
[12:25:58] <- POST 200 Reply[m
[12:25:58] <- POST 200 Reply[m
[12:25:58] <- POST 200 Reply[m
[12:25:58] <- POST 200 setPriv
[12:25:59] <- POST 200 setUser
[12:25:59] <- POST 200 setConv
[12:25:59] -> POST 200 [messag
```

- ☐ Close the emulator.
- ☐ In Visual Studio, stop debugging.

Conclusion

In this tutorial, we have covered the steps to make a basic bot that can retrieve data from a Synergy ISAM file. We have seen how regular expressions and .NET string methods can help parse input. And we've seen how to use attachments to send multimedia output to the user, such as images.

This tutorial has only scratched the surface on the topic of bots. It hasn't covered all of the steps required to host a bot as an online web service. This process would include the same steps as hosting any non-Synergy bot built in Visual Studio with Microsoft.BotBuilder. However, it would also require making sure that the program can access Synergy licenses, and that the environment is set up so it can access the data files.

Additionally, there are many more advanced techniques that can be used in building bots, such as using cognitive services to parse user input, or taking advantage of channel-specific communication features. Fortunately, there are other materials available online to explain these techniques, and they most likely can be adapted to work with Synergy. If you're interested in this subject, I'd recommend that you explore further and see what you can find.

This tutorial is just a starting point. It shows that it is possible, and not too difficult, to build a working bot using Synergy. I hope you have found it useful.

Appendix: Source Files

This Appendix contains the completed Synergy source files from the tutorial. It only includes files that were created or modified manually, and does not include files that are unmodified from the template, such as `MessagesController.dbl`. It also doesn't include the two-line definition file `DataDefinitions.def`, which is explained on page 14.

Feel free to refer to these files to double-check code from the tutorial, or copy code from here into the Visual Studio solution. In some cases, such as the `CreateDataFiles` project, the instructions specifically ask you to copy code from here, as the code is repetitive and would take up too much space in the body of the tutorial.

Note that some source files, such as `ProcessQuery.dbl` and `SendResponse.dbl`, are updated multiple times throughout the tutorial. So code copied from the appendix may not work properly if used in an intermediate part of the tutorial.

The first source file, named `Program.dbl`, is from the `CreateDataFiles` project, and is identified as such. The last file is from the `Repository` project, and is identified as such. The remaining files are from the `EmployeeBot.Bot` project, specifically from the `Dialogs` folder.

Building a Bot with Synergy .NET

CreateDataFiles\Program.dbf

```
import System
import System.Collections.Generic
import System.Text

namespace CreateDataFiles

;;; <summary>
;;; The main entry point for the application.
;;; </summary>
main
.include "EMPLOYEE" repository, record="employee", END
proc
data filename, string, "DAT:employees.ism"
data filechan, int

isamc(filename, 212, 3,
& "START=1, LENGTH=8, NAME=EMPLOYEEID, TYPE=SEQUENCE",
& "START=9, LENGTH=20, NAME=FIRSTNAME, TYPE=ALPHA, DUPS",
& "START=29, LENGTH=20, NAME=LASTNAME, TYPE=ALPHA, DUPS",
& "START=49, LENGTH=64, NAME=TITLE, TYPE=ALPHA, DUPS")

open(filechan, "U:I", filename)

employee.firstname = "Jodah"
employee.lastname = "Veloper"
employee.title = "Software Engineer"
employee.image = "http://www.jodahveloper.com/wp-content/uploads/2017/07/Jodah.png"
store(filechan, employee)

employee.firstname = "Bigbah"
employee.lastname = "Smann"
employee.title = "CEO"
employee.image = "http://www.jodahveloper.com/wp-content/uploads/2017/07/Bigbah.png"
store(filechan, employee)

employee.firstname = "Manny"
employee.lastname = "Jurr"
employee.title = "Product Development Manager"
employee.image = "http://www.jodahveloper.com/wp-content/uploads/2017/08/Manny.png"
store(filechan, employee)

employee.firstname = "Mark"
employee.lastname = "Etting"
employee.title = "Director of Communications"
employee.image = "http://www.jodahveloper.com/wp-content/uploads/2017/07/Mark.png"
store(filechan, employee)

employee.firstname = "Vi"
employee.lastname = "Sprezz"
employee.title = "Senior Executive"
employee.image = "http://www.jodahveloper.com/wp-content/uploads/2017/07/Vi.png"
store(filechan, employee)

employee.firstname = "Connie"
employee.lastname = "Sultant"
employee.title = "Professional Services Specialist"
```

Building a Bot with Synergy .NET

```
employee.image = "http://www.jodahveloper.com/wp-content/uploads/2017/07/Connie.png"
store(filechan,employee)

employee.firstname = "Jodah"
employee.lastname = "Bot"
employee.title = "Artificial Intelligence"
employee.image = ""
store(filechan,employee)

close filechan

endmain

endnamespace
```

Building a Bot with Synergy .NET

GetEmployeeByFirstName.dbl

```
import System
import System.Collections.Generic
import System.Text
import System.Collections
import Synergex.SynergyDE.Select

namespace EmployeeBot.Bot.Dialogs

    .include "DEF:DataDefinitions.def"

    public partial class RetrieveData
        public static method GetEmployeeByFirstName, @ArrayList
            employeeName, string
            endparams
        proc
            data arraylist = new ArrayList()
            data emp_rec, emp_struct
            disposable data fobj, @From, new From(DATAFILE,emp_rec)
            disposable data sobj, @Select, new Select(fobj,
                &      (NoCaseWhere)(emp_rec.firstname .eqs. employeeName))

            foreach emp_rec in sobj
                begin
                    arraylist.Add(emp_rec)
                end
            mreturn arraylist
        endmethod
    endclass
endnamespace
```

Building a Bot with Synergy .NET

GetEmployeeByID.dbl

```
import System
import System.Collections.Generic
import System.Text
import System.Collections
import Synergex.SynergyDE.Select

namespace EmployeeBot.Bot.Dialogs

    .include "DEF:DataDefinitions.def"

    public partial class RetrieveData
        public static method GetEmployeeByID, @ArrayList
            employeeID, int
        endparams
        proc
            data arraylist = new ArrayList()
            data emp_rec, emp_struct
            disposable data fobj, @From, new From(DATAFILE,emp_rec)
            disposable data sobj, @Select, new Select(fobj,
                &      (Where)(emp_rec.employeeid .eq. employeeID))

            foreach emp_rec in sobj
                begin
                    arraylist.Add(emp_rec)
                end
            end

            mreturn arraylist
        endmethod
    endclass
endnamespace
```

Building a Bot with Synergy .NET

GetEmployeeByLastName.dbl

```
import System
import System.Collections.Generic
import System.Text
import System.Collections
import Synergex.SynergyDE.Select

namespace EmployeeBot.Bot.Dialogs

    .include "DEF:DataDefinitions.def"

    public partial class RetrieveData
        public static method GetEmployeeByLastName, @ArrayList
            employeeName, string
            endparams
        proc
            data arraylist = new ArrayList()
            data emp_rec, emp_struct
            disposable data fobj, @From, new From(DATAFILE,emp_rec)
            disposable data sobj, @Select, new Select(fobj,
                &      (NoCaseWhere)(emp_rec.lastname .eqs. employeeName))

            foreach emp_rec in sobj
                begin
                    arraylist.Add(emp_rec)
                end
            mreturn arraylist
        endmethod
    endclass
endnamespace
```


Building a Bot with Synergy .NET

GetEmployeeByTitle.dbl

```
import System
import System.Collections.Generic
import System.Text
import System.Collections
import Synergex.SynergyDE.Select

namespace EmployeeBot.Bot.Dialogs

    .include "DEF:DataDefinitions.def"

    public partial class RetrieveData
        public static method GetEmployeeByTitle, @ArrayList
            employeeTitle,      string
            endparams
        proc
            data arraylist = new ArrayList()
            data emp_rec, emp_struct
            disposable data fobj, @From, new From(DATAFILE,emp_rec)
            disposable data sobj, @Select, new Select(fobj,
                &      NoCaseWhere.Contains(emp_rec.title,employeeTitle))

            foreach emp_rec in sobj
                begin
                    arraylist.Add(emp_rec)
                end
            end

            mreturn arraylist
        endmethod
    endclass

endnamespace
```

MessagesDialog.dbl

```
import System
import System.Collections
import System.Collections.Generic
import System.Text
import System.Text.RegularExpressions
import System.Threading.Tasks
import Microsoft.Bot.Builder.Dialogs
import Microsoft.Bot.Connector

namespace EmployeeBot.Bot.Dialogs

    {Serializable}
    public partial class MessagesDialog implements IDialog<object>

        public async method StartAsync,    @Task
            context, @IDialogContext
        proc
            context.Wait(MessageReceivedAsync)
        endmethod

        public async method MessageReceivedAsync, @Task
            context, @IDialogContext
            argument, @IAwaitable<IMessageActivity>
        proc
            data message, @IMessageActivity, await argument
            data msgtext = (message.Text == ^null) ? "" : message.Text
            await ProcessQueryAsync(context,msgtext)
            context.Wait(MessageReceivedAsync)
        endmethod

    endclass

endnamespace
```

Building a Bot with Synergy .NET

ProcessQuery.dbl

```
import System.Text.RegularExpressions
import System.Threading.Tasks
import Microsoft.Bot.Builder.Dialogs
import System.Collections

namespace EmployeeBot.Bot.Dialogs

    public partial class MessagesDialog implements IDialog<object>

        public async method ProcessQueryAsync, @Task
            context,      @IDialogContext
            msgtext,      string

        proc
            data EmployeeList, @ArrayList, new ArrayList()
            data isSearchOver, boolean, false

            ;If a user asks for help, send information about the Bot's abilities
            if (Regex.IsMatch(msgtext, "help", RegexOptions.IgnoreCase))
                begin
                    await context.PostAsync("Greetings from EmployeeBot.")
                    await Task.Delay(200)
                    await context.PostAsync("I'm here to tell you about employees.")
                    await Task.Delay(200)
                    await context.PostAsync("If you know the ID of an employee, type it in: e.g
'E2'.")

                    await Task.Delay(200)
                    await context.PostAsync("If you know the name of an employee, mention it and
I'll look it up, e.g. 'tell me about Bigbah'.")
                    await Task.Delay(200)
                    await context.PostAsync("I'll do my best to find the person you want to know
about.")

                    mreturn
                end

            ;Search employees by ID [match E#]
            if (Regex.IsMatch(msgtext, "E\d+", RegexOptions.IgnoreCase))
                begin
                    data eid, int, ExtractEmployeeID(msgtext)
                    EmployeeList = RetrieveData.getEmployeeByID(eid)
                    if (EmployeeList.Count > 0) isSearchOver = true
                end

            ;Search by name or title
            if (!isSearchOver)
                begin
                    data tokenList, @ArrayList, new ArrayList()
                    tokenList = tokenizeQuery(msgtext)
                    data token, string
                    foreach token in tokenList
                        begin
                            EmployeeList.AddRange(RetrieveData.GetEmployeeByFirstName(token))
                            EmployeeList.AddRange(RetrieveData.GetEmployeeByLastName(token))
                            EmployeeList.AddRange(RetrieveData.GetEmployeeByTitle(token))
                        end
                    if (EmployeeList.Count > 0) isSearchOver = true
                end
            end
        end
    end
```

Building a Bot with Synergy .NET

```
        await SendResponseAsync(context, msgtext, EmployeeList)

    mreturn
endmethod

;Extract employee ID from messagetext
private method ExtractEmployeeID, int
    inputstring, string
endparams
proc
    data messagetext = inputstring
    messagetext = Regex.Match(messagetext, "E\d+", RegexOptions.IgnoreCase).Value
    messagetext = Regex.Replace(messagetext, "E", "", RegexOptions.IgnoreCase)
    data eid, int
    eid = messagetext
    mreturn eid
endmethod

;Split the message text into tokens
private method tokenizeQuery, @ArrayList
    msgtext, string
endparams
proc
    data token, string
    data tokens, [#]string
    data tokenlist, @ArrayList, new ArrayList()

    data separators, [#]char, new char[#] {" ", ",", ".", "?"}
    tokens = msgtext.Split(separators)

    foreach token in tokens
    begin
        if (!Regex.IsMatch(token, "E\d+", RegexOptions.IgnoreCase))
        begin
            tokenlist.Add(token)
        end
    end

    mreturn tokenlist
endmethod

endclass

endnamespace
```

Building a Bot with Synergy .NET

SendResponse.dbl

```
import System.Collections
import System.Threading.Tasks
import Microsoft.Bot.Builder.Dialogs
import Microsoft.Bot.Connector

namespace EmployeeBot.Bot.Dialogs

.include "DEF:DataDefinitions.def"

public partial class MessagesDialog implements IDialog<object>
    public async method SendResponseAsync, @Task
        context, @IDialogContext
        msgtext, string
        EmployeeList, @ArrayList
    endparams
    proc
        if (EmployeeList.count == 0)
        begin
            await context.PostAsync("Sorry, I couldn't find anything matching '" +
                & msgtext + "'")
            mreturn
        end

        data emp_rec, emp_struct
        data generalEmployeeList, @ArrayList, new ArrayList()
        data generalIDList, @ArrayList, new ArrayList()
        data relevantEmployeeList, @ArrayList, new ArrayList()
        data relevantIDList, @ArrayList, new ArrayList()

        ;filter employee list to remove duplicates
        foreach emp_rec in EmployeeList
        begin
            if (generalIDList.Contains(emp_rec.employeeid)) then
            begin
                ;If the current employee record is already in the general list,
                ;then it's a duplicate that matched multiple search terms.
                ;Add it to the relevant list, unless it's there already.
                if (relevantIDList.Contains(emp_rec.employeeid) == false)
                begin
                    relevantEmployeeList.Add(emp_rec)
                    relevantIDList.Add(emp_rec.employeeid)
                end
            end
            else
            begin
                ;If it isn't in the general list already, add it.
                generalEmployeeList.Add(emp_rec)
                generalIDList.Add(emp_rec.employeeid)
            end
        end

        ;if there aren't any records more relevant than others,
        ;treat all records as relevant
        if (relevantIDList.count == 0)
        begin
            relevantIDList = generalIDList
        end
    end
end
```

Building a Bot with Synergy .NET

```
        relevantEmployeeList = generalEmployeeList
    end

    if (relevantEmployeeList.count > 1) then
    begin
        ;there's more than one record, so just show basic information
        ;Loop through employee records and display them
        foreach emp_rec in relevantEmployeeList
        begin
            await context.PostAsync(emp_rec.firstname + " " + emp_rec.lastname +
            &      " (E" + %string(emp_rec.employeeid)+ ")")
            await Task.Delay(200)
        end
        await context.PostAsync(
        &      "For more details, search for an employee ID, e.g. 'E' +
        &      %string(emp_rec.employeeid) + '"")
    end
    else
    begin
        ;there's only one record, so show more details
        foreach emp_rec in relevantEmployeeList
        begin
            await context.PostAsync(emp_rec.firstname + " " + emp_rec.lastname +
            &      " (E" + %string(emp_rec.employeeid)+ ")")
            await Task.Delay(200)
            await context.PostAsync(emp_rec.title)
            await Task.Delay(200)

            ;Show picture, if applicable
            if (%atrim(emp_rec.image) .nes. "")
            begin
                data message1 = context.MakeMessage()
                message1.Attachments.Add(new Attachment() {
                &      ContentUrl = %atrim(emp_rec.image),
                &      ContentType = "image/png",
                &      Name = %atrim(emp_rec.firstname)})
                await context.PostAsync(message1)
            end
        end
    end
    end

    mreturn
endmethod
endclass

endnamespace
```

Building a Bot with Synergy .NET

Repository\repository.scm

```
Structure EMPLOYEE   DBL ISAM
  Description "Employee information"

Field EMPLOYEEID   Type AUTOSEQ   Size 8
  Description "ID"
  Readonly
  Nonnull

Field FIRSTNAME   Type ALPHA   Size 20
  Description "The employee's first name"

Field LASTNAME   Type ALPHA   Size 20
  Description "Employee's last name"

Field TITLE   Type ALPHA   Size 64
  Description "Job title"

Field IMAGE   Type ALPHA   Size 100
  Description "URL for a picture of the employee"

Key EMPLOYEEID   ACCESS   Order ASCENDING   Dups NO
  Description "ID"
  Segment FIELD   EMPLOYEEID   SegType SEQUENCE

Key FIRSTNAME   ACCESS   Order ASCENDING   Dups YES
  Description "The employee's first name"
  Segment FIELD   FIRSTNAME   SegType ALPHA

Key LASTNAME   ACCESS   Order ASCENDING   Dups YES
  Description "Employee's last name"
  Segment FIELD   LASTNAME   SegType ALPHA

Key TITLE   ACCESS   Order ASCENDING   Dups YES
  Description "Employee's job title"
  Segment FIELD   TITLE   SegType ALPHA
```