

Using C # Code in your Botlet Logic

Introduction

This document will guide you through the steps of how to consume an Azure App Service in a service botlet. You can write your botlet logic in any language of your choice (e.g., C#, JavaScript, Python, etc.). You can use any integrated development environment that you prefer. When your work is done, you can have the service botlet hosted up in Microsoft Azure.

To host a service botlet in Azure requires that you have a subscription. If you do not have a Microsoft Azure subscription and you need to obtain one, click [Start Free](#).

It is not always necessary that you must write your own botlet components. You may use items that you discover in the [Store](#) and reuse them in your own flow. Items in the Store handle simple and complex tasks such as chaining botlets together to help create a unique user experience.

To illustrate this example, we will use C# code and we will be:

- Using Visual Studio to create an ASP .NET Web Application
- Creating a C# Class
- Deploying the new service to Microsoft Azure
- Creating a service botlet in the Workspace to consume the functionality of the new service
- Testing the consumption and interaction of the new service botlet with a test botlet using the Test Chat Tool

The objects BotRequest and BotResponse are part of the communication protocol used by botlets and services. To see the details of these objects, see [BotRequest and BotResponse Objects](#).

Create an ASP .NET Web Application

To consume an Azure App Service in a service botlet, create an ASP .NET Web Application in the integrated development environment of your choice. In the steps provided below, we are using Microsoft Visual Studio and have selected an empty Visual C# project template.

If you require detailed step-by-step instructions for adding a web application in Visual Studio, see: [Getting Started With ASP .NET Web API 2 \(C#\)](#).

Create New Project

1. Create a new project in Microsoft Visual Studio and give the web application a name.
2. Select a project template type (i.e., **ASP .NET Web application - .NET Framework**), name it **KnowledgeStoreService**, and then click **OK**.
3. Select the **Empty** project template, click **Web API**, and then click **OK**.

Now that we've created a new project and named the ASP .NET Web application "KnowledgeStoreService," we are now ready to create a C# class.

Create C# Class

1. In your project, select the **Visual C# class** in the list, give it a name (e.g., HelloWorldService.cs), and then click **Add**.
2. Add the following attributes to the **HelloWorldService.cs** project file.

```
[KnowledgeStoreService]
```

3. Enter the string name. For example:

```
[KnowledgeStoreService ("msd.valeriy.hello_world_services")]
```

Note: Ensure that the service name matches the input parameter of the Action that is created in the Workspace.

Example:

```
namespace KnowledgeStoreService.Services
{
    using Microsoft.KnowledgeStore.Service.Attributes;

    [KnowledgeStoreService("msd.valeriy.hello_world_service")]
    public class HelloWorldService
    {
    }
}
```

4. Add the action parameter string name to call the service botlet.

Example:

```
[KnowledgeStoreAction("action")]
public string Action(string name)
{
    return "Hello, " + name;
}
```

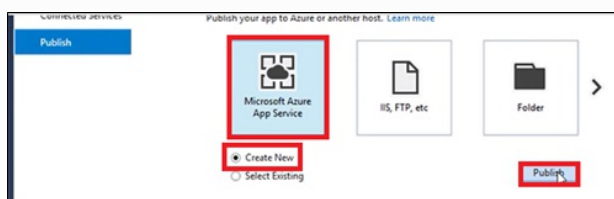
Note: KnowledgeStoreAction is the function that calls the action specified in the service botlet.

It at this point that you are ready to deploy your service to Azure.

Deploy the Service to Azure

This section describes the steps for deploying a service to the Microsoft Azure App Service.

1. In Visual Studio, go to the **Solution Explorer**, right-click the **Project**, and then click **Publish**.
2. Click **Microsoft Azure App Service**, click **Create new**, and then click **Publish**.

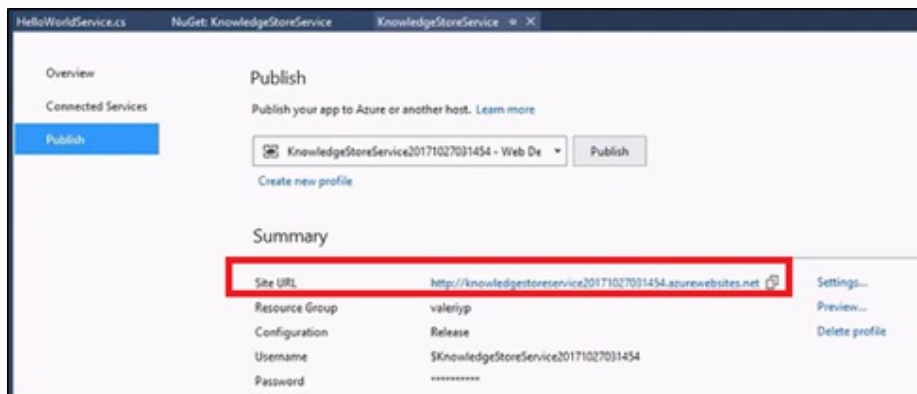


For more details about the deployment workflow steps, see: [Create an ASP .NET web app in Azure..](#)

3. In the **Create App Service** window, select the following Azure resources under the **Hosting** tab, and click **Create**. Ensure to enter an **API App Name**. Click the drop-down menu to select a **Subscription**, **Resource**

Group, and App Service Plan.

To locate and copy the **Site URL**, click the **Publish** tab. As shown in the image below, the Site URL is under the **Summary** section.

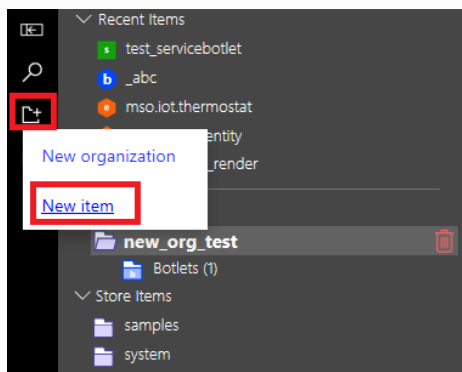


This completes the steps to deploying a service to Azure. To ensure that a deployment to Azure is functioning properly, you can test it out using test tools such as Postman or Fiddler.

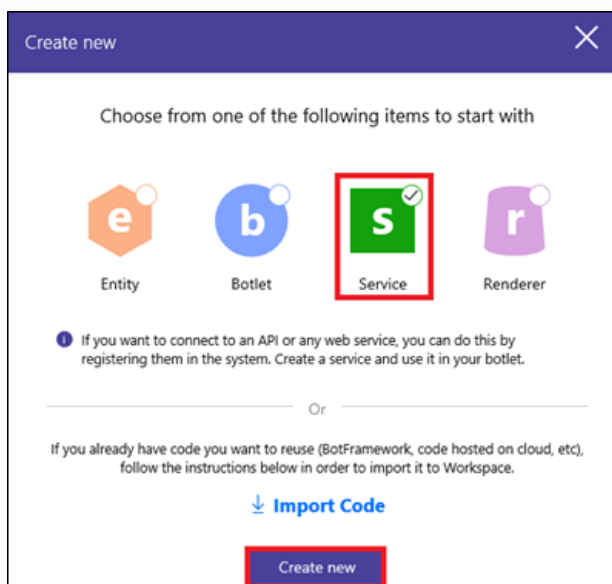
Test the Web API in a Service Botlet

The high-level steps described below describe what you must do to create a service botlet.

1. In the **Items Tree** click **Create New**, and then click **New item**.



2. Click **Service**, and then click **Create new**.



3. Enter an **organization** name, **Service** name, click a **Category**, and then click **Next**.

New service

Step 1 of 3

Organization ⓘ

Type an existing organization name

Give your service a name ⓘ

service_unique_id

Service ID ⓘ

Category ⓘ

Primary Category

Next

Note: This service botlet is a wrapper around the new service that was just created.

4. Enter a description for the service and click **Next**.

Follow the steps to create your service.

Step 2 of 3

Service ID:
my_first_organization.test3

Category:
Action & Adventure

Description ⓘ

Service Image ⓘ

Browse...

A default image will be used if no image is selected

Previous Next

5. Select the option **I have a Web Service**, enter the **Azure Webservice URL**, and then click **Done**.

Follow the steps to create your service.

×

Step 3 of 3

Choose Service Type:

☒ I have an API and document

In the next steps, you will be guided to fill out the information for actions and input / output parameters with the API and documentations.

☐ I have a swagger specification

Upload your swagger specification file or copy/paste the raw json to the text editor.

☐ I have a web service

A web service provides you the inputs and outputs for the actions of your service. You can simply enter the webservice URL to get started.

Previous

Done

Note: The Azure Webservice URL is obtained from the deployment performed in the previous section.

In addition, the Webservice URL (i.e., Webhook) should be formatted like this:

http(s)://<host>/knowledgestore_service/<service>

Where

“Host” comes from the Azure deployment.

“Service” is the one specified in the `[KnowledgeStoreService(<service>)]` attribute.

Adding an Action to the Azure Webservice

The following steps describe how to add an action and to the new Azure webservice.

1. Click the **Organization** in the **Workspace Items Tree** that contains the service botlet that you want to add an action.
2. In the **Control Panel**, click the **Actions** menu tab.
3. Enter an **Action name** and **Description**, and then click **Add**.

Overview >

Discoverability >

Actions >

Permissions >

Welcome to your new service! Let's create an action to get started.
An action allows your service to perform its primary task.

Add a new action

new_action

This is an action for my new test service.

Add



Note: The action name (i.e., action) is obtained from the source code as shown in the example below.

Example:

```
[KnowledgeStoreAction("action")]
public string Action(string name)
{
    return "Hello, " + name;
}
```

- Click **Add parameters**.

+ Add a new action

Action Name *	Description *	Intent	Status	Edit	Delete
> test_123	This is only a test.			Add parameters >	 

- Click **+ New input parameter**, and enter the **Parameter name**, **Entity Type** (e.g., string), provide a **Description**, then click **Add**.

Input parameters:

X New input parameter

Input

Parameter name *

Entity type *

Input

Enter a parameter name (e.g

Enter entity type id

Description *

Add

Required Parameter name *	Entity type *	Description *	Edit	Delete
---------------------------	---------------	---------------	------	--------

Note: The input parameter "name" must match the name specified in the class of the C# code as shown in

the example below.

Example:

```
[KnowledgeStoreService("msd.valeriyp.test_service")]
public class TestService
{
    [KnowledgeStoreAction("action")]
    public string Action(string name)
    {
        return "Hello, " + name;
    }
}
```

7. Click **New output parameter** and enter an **Output parameter name**, **Entity type** (e.g., string), provide a **Description**, and then click **Add**.

Output	Parameter name*	Entity type*	Description*	Edit	Delete
--------	-----------------	--------------	--------------	------	--------

Note: The output parameter name used in this example is called “result” and the entity type is a string.

8. Click **Save** to save the newly added Input and Output parameters.

It is at this point that a new service botlet has been created. The next thing to do is to test this service botlet by calling a test botlet.

Executing the New Service Using C

The steps below describe how to call the test botlet from the service botlet, using C# code.

1. In the **Items Tree**, select the service botlet and click the **Overview** menu tab.
2. Copy the service botlet’s action parameters by selecting the SLC code displayed in the Code Viewer and press **CTR-C** to copy it.

Example SCL Code:

```
SAY "What is your name"
GET_INPUT
CALL "msd.valeriyp.hello_world_service", "action", name=USER_INPUT STORE text
SAY text.result
```

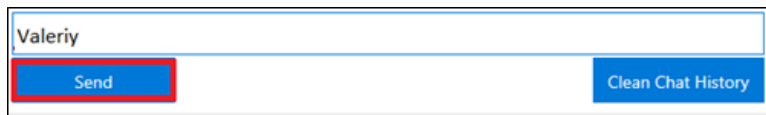
3. Select the test botlet in the **Items Tree** and click the **Overview** tab in the Control Panel.
4. Copy the SCL code from the service botlet to the test botlet. Each line of the SCL code is described below in terms of what it does.
5. Click **Save**.

It is at this point that you can test the interaction of the service botlet with the test botlet using the Chat Tool.

Using the Chat Tool for Testing Purposes

To determine if the test botlet is properly calling the service botlet, follow the steps below.

1. Select the test botlet in the Workspace **Items Tree**.
2. In the Control Panel, click the **Chat** menu tab.
3. In the **Chat** window, enter your name in the text field, and then click **Send** or press **Enter**.



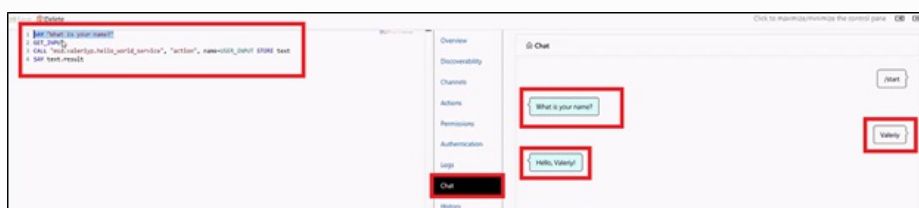
As shown in the following image, the results of the test should produce two text lines. One is from the SAY command, and the second is from the response type of the C# code.



Note: If the test chat does not implement what is specified in the C# code (e.g., “Hello Valeriy”), it will fail. If the test chat fails, review the **Logs** to troubleshoot the failed call.

Analyzing the SCL

The image below shows a side-by-side comparison of the SCL that gets displayed inside the Code Viewer and the interactions that it has testing the service using the Chat tool.



In line 1, **SAY “What is your name?”** This is what gets displayed when the Chat tool starts for this test botlet.

In line 2, **Get_Input** This displays the user’s name (e.g., Valeriy!) in the Chat tool after the user enters their name in the text field and clicks **Send** or presses **Return**.

In line 3, **CALL “msd.valeriyp.hello_world_service”** This calls the service to get the input text string.

In line 4, **SAY text.result** This sends back the response string (i.e., Hello, Valeriy!).

Analyzing the C# Code

The code sample shown below is from the HelloWorldService.cs project file. It is here that the C# code takes the name and it returns a string “Hello {name}!”; to the SCL (i.e., SAY text.result).


```

namespace TestKnowledgeStoreService.Services
{
    using Microsoft.KnowledgeStore.Service.Attributes;

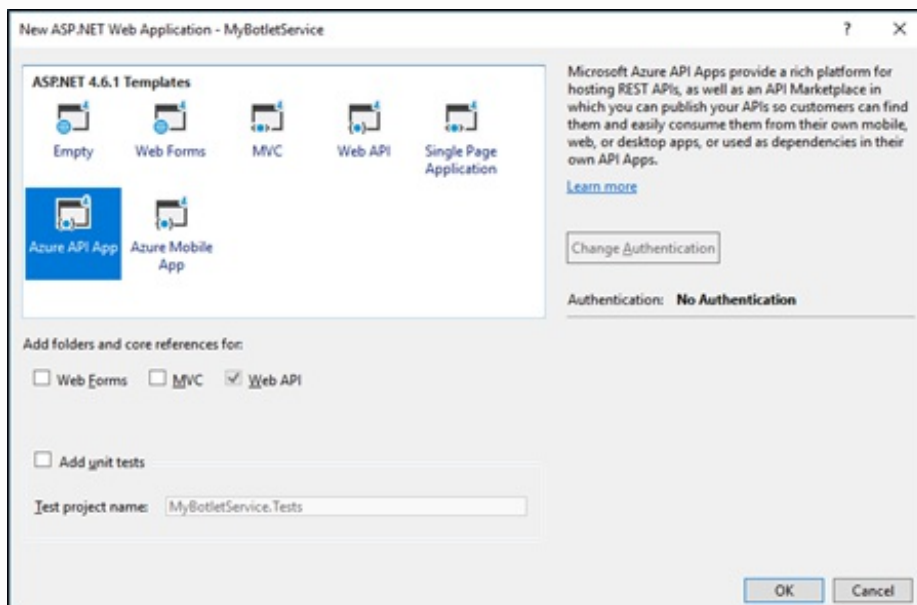
    [KnowledgeStoreService("msd.valeriy.test_service")]
    public class TestService
    {
        [KnowledgeStoreAction("action")]
        public string Action(string name)
        {
            return "Hello, " + name;
        }
    }
}

```

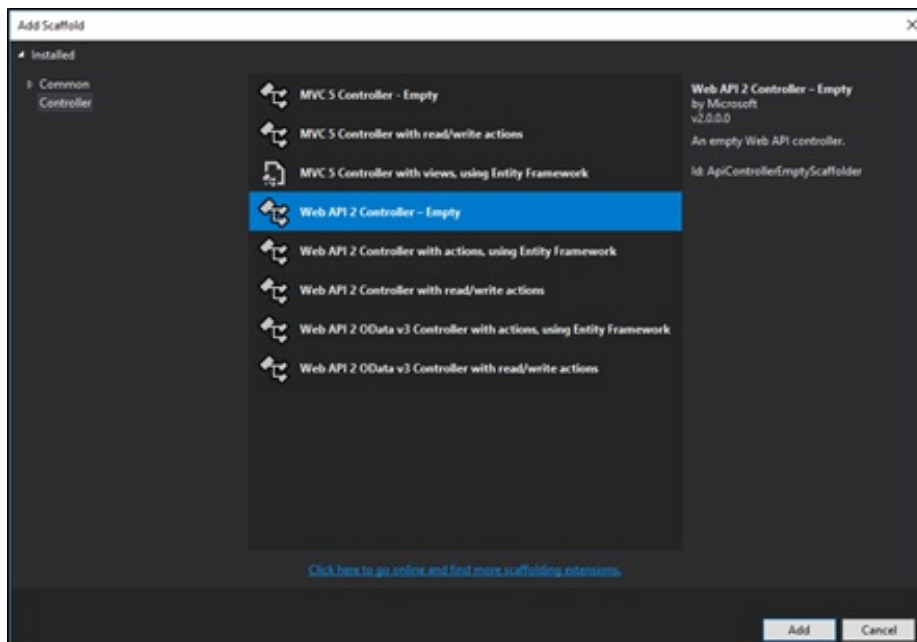
Using Web API Controller

If you are not using a NuGet package to consume an Azure App Service in a service botlet, refer to the high-level steps below that describe how to use the Web API Controller instead.

1. In Visual Studio, create a new web application.
2. Select the **Azure API App** and enter the Test project name, and then click **OK**.



3. Select the **Web API 2 Controller - Empty**, and then click **Add**.



4. Refer to the code sample provided in the [Default Controller Code Sample](#) project file.
5. Deploy your service to Azure.
6. Create a service botlet in the Workspace.
7. For the service type, select **I have a Web Service**, enter the **Azure Webservice URL**, and then click **Done**.

An example of the webhook URL is: `[your web app name].azurewebsite.net/api/sample`

8. Create two actions for your service. The parameter to enter are:

First action is `say_hi`

Add a non-required parameter of type `mso.person` named **person**.

Then add a second action named "fail". This one won't need any parameters.

9. Create a new test botlet.

Key Points

The interaction between the "service botlet" and your service is done following a JSON schema that represents objects running under the covers of our framework.

The two most important ones are called `mst.bot.response` and `mst.bot.request`.

Using the example provided, this is what gets generated.

Request:

```
CALL "msd.alfremen.sample_service" , "say_hi"

{
  "@": "mst.bot.request",
  "method": "say_hi"
}
```

Response:

```
{
  "@": "mst.bot.response",
  "result": {
    "greetings": "Hello world"
  }
}
```

Which can be generated by the following command in SCL:

Request:

```
SAY CALL_RESULT.greetings

The parameters for the following line
CALL "msd.alfremen.sample_service", "say_hi", name = "Alfredo"
Sends the following JSON (simplified)
{
  "@": "mst.bot.request",
  "method": "say_hi",
  "data": {
    "@": "mst.map",
    "name": "Alfredo"
  }
}
```

Response:

```
{
  "@": "mst.bot.response",
  "result": {
    "greetings": "Hello Alfredo"
  }
}
```

When there is an error, the output is shown below.

Using our “fail” method CALL “msd.alfremen.sample_service”, “fail” SAY “This will not be said”
ON_ERROR SAY CALL_ERROR.text

CALL is sending the following JSON

Request:

```
{
  "@": "mst.bot.request",
  "method": "fail",
  "data": {
    "@": "mst.map",
    "name": "Alfredo"
  }
}
```

Response:

```
{
  "@": "mst.bot.response",
  "error": {
    "@": "mst.error",
    "text": "This is a fail message from the service"
  }
}
```

The specific error can be caught using the `ON_ERROR` syntax, and the object of the error will be contained within `CALL_ERROR`, which is why when we do `CALL_ERROR.text`. The value produced is “This is a fail message from the service.”

In addition to atomic types, you can also include other more complex entity types. When deserializing the JSON from `mst.bot.request` and serializing into `mst.bot.response`, ensure to remember that all entity types need to include a property whose key is “@” and which the value is the name of the entity as declared in the portal.

For example, if you are sending a location as a parameter, it may look something like the code example shown below.

```
{
  "@": "mst.bot.request",
  "method": "say_hi",
  "data": {
    "@": "mst.map",
    "person": {
      "@": "mso.person",
      "name": "Alfredo"
    }
  }
}
```

Default Controller Code Sample

```
using System;
using System.Collections.Generic;
using System.Web.Http;
using System.Web.Http.Description;
using Newtonsoft.Json;
using System.Threading.Tasks;
using Newtonsoft.Json.Linq;

namespace MyBotletService.Controllers
{
    public class DefaultController : ApiController
    {
        [HttpPost]
        [Route("api/sample")]
        [ResponseType(typeof(BotResponse))]
        public async Task<IHttpActionResult> HandleQuery()
        {
            BotResponse response = new BotResponse();
            try
            {
                // Read JSON contained as body in the POST request
                string jsonString = await Request.Content.ReadAsStringAsync();

                // Deserialize into a C# object
                BotRequest botRequest = JsonConvert.DeserializeObject<BotRequest>(jsonString);

                switch (botRequest.method)
                {
                    case "say_hi":
                        response.result = SayHi(botRequest.data);
                        break;
                    case "fail":
                        response.result = AlwaysFail();
                        break;
                    default:
                        throw new NotImplementedException($"{botRequest.method} is not implemented");
                        break;
                }
            }
            catch (Exception ex)
            {
                //TODO: Log Exception
            }
        }
    }
}
```

```

        response.error = new Error()
        {
            text = ex.Message,
        };
    }

    // send back as JSON
    return Json(response, new JsonSerializerSettings() { NullValueHandling = NullValueHandling.Ignore });
}

private Map AlwaysFail()
{
    throw new Exception("This is a fail message from the service");
}

private Map SayHi(Map requestData)
{
    Map results = new Map();

    if (requestData != null)
    {
        if (requestData.TryGetValue("name", out object nameObject))
        {
            //we know we expect the object in name to be o type string
            string name = (string)nameObject;
            results.Add("greetings", $"Hello {name}");
        }
        else if (requestData.TryGetValue("person", out object personObject))
        {
            string name = (string)((JObject)personObject)["name"];
            results.Add("greetings", $"Hello person named {name}");
        }
    }

    if(results.Count == 0)
    {
        results.Add("greetings", "Hello world");
    }

    return results;
}

}

public class BotRequest
{
    [JsonProperty(PropertyName = "@")]
    public static readonly string entityName = "mst.bot.request";

    public string integration_type { get; set; }
    public string integration_key { get; set; }
    public string user_id { get; set; }
    public object integration_data { get; set; }
    public string session { get; set; }
    public string method { get; set; }
    public Map data { get; set; }
    public Map context { get; set; }
    public string id { get; set; }
    public string trace_id { get; set; }
}

public class BotResponse
{
    [JsonProperty(PropertyName = "@")]
    public static readonly string entityName = "mst.bot.response";

    public Error error { get; set; }
    public Map result { get; set; }
    public string id { get; set; }
    public string trace_id { get; set; }
}

//The value is an object because it can be any entity type
public class Map : Dictionary<string, object>
{
    [JsonProperty(PropertyName = "@")]
    public static readonly string entityName = "mst.map";
}

```

```
        public static readonly string entityName = "mst.error";  
    }  
  
    public class Error  
    {  
        [JsonProperty(PropertyName = "@")]  
        public static readonly string entityName = "mst.error";  
  
        public string code { get; set; }  
        public string text { get; set; }  
        public string debug { get; set; }  
        public IList<string> trace { get; set; }  
    }  
}
```

[← Previous](#)

© Copyright 2018, Microsoft Corporation.

Built with [Sphinx](#) using a [theme](#) provided by [Read the Docs](#).