Azure Functions C# script developer reference

06/07/2017 • 9 minutes to read • Contributors all

In this article

C# script

The C# script experience for Azure Functions is based on the Azure WebJobs SDK. Data flows into your C# function via method arguments. Argument names are specified in `function.json`, and there are predefined names for accessing things like the function logger and cancellation tokens.

This article assumes that you've already read the Azure Functions developer reference.

For information on using C# class libraries, see Using .NET class libraries with Azure Functions.

How .csx works

The `.csx` format allows you to write less "boilerplate" and focus on writing just a C# function. Include any assembly references and namespaces at the beginning of the file as usual. Instead of wrapping everything in a namespace and class, just define a `Run` method. If

you need to include any classes, for instance to define Plain Old CLR Object (POCO) objects, you can include a class inside the same file.

Binding to arguments

The various bindings are bound to a C# function via the `name` property in the *function.json* configuration. Each binding has its own supported types; for instance, a blob trigger can support a string, a POCO, or a CloudBlockBlob. The supported types are documented in the reference for each binding. A POCO object must have a getter and setter defined for each property.

C# Copy

```
public static void Run(string myBlob, out MyClass myQueueItem)
{
    log.Verbose($"C# Blob trigger function processed: {myBlob}");
    myQueueItem = new MyClass() { Id = "myid" };
}

public class MyClass
{
    public string Id { get; set; }
}
```

Tip

If you plan to use the HTTP or WebHook bindings, plan to avoid port exhaustion that can be caused by improper instantiation of `HttpClient`. For more information, review the article Improper Instantiation antipattern.

Using method return value for output binding

You can use a method return value for an output binding, by using the name `$return` in *function.json*:

JSON Copy

```
{
    "type": "queue",
    "direction": "out",
    "name": "$return",
    "queueName": "outqueue",
    "connection": "MyStorageConnectionString",
}
```

C# Copy

```
public static string Run(string input, TraceWriter log)
{
    return input;
}
```

Writing multiple output values

To write multiple values to an output binding, use the `ICollector` or `IAsyncCollector` types. These types are write-only collections that are written to the output binding when the method completes.

This example writes multiple queue messages into the same queue using `ICollector`:

C# Copy

```
public static void Run(ICollector<string> myQueueItem, TraceWriter log)
{
    myQueueItem.Add("Hello");
    myQueueItem.Add("World!");
}
```

Logging

To log output to your streaming logs in C#, include an argument of type `TraceWriter`. We recommend that you name it `log`. Avoid using `Console.Write` in Azure Functions.

`TraceWriter` is defined in the Azure WebJobs SDK. The log level for `TraceWriter` can be configured in host.json.

C# Copy

```
public static void Run(string myBlob, TraceWriter log)
{
    log.Info($"C# Blob trigger function processed: {myBlob}");
}
```

Async

To make a function asynchronous, use the `async` keyword and return a `Task` object.

C# Copy

```
public async static Task ProcessQueueMessageAsync(
        string blobName,
        Stream blobInput,
        Stream blobOutput)
{
    await blobInput.CopyToAsync(blobOutput, 4096, token);
}
```

Cancellation token

Some operations require graceful shutdown. While it's always best to write code that can handle crashing, in cases where you want to handle graceful shutdown requests, you define a `CancellationToken` typed argument. A `CancellationToken` is provided to signal that a host shutdown is triggered.

C# Copy

```
public async static Task ProcessQueueMessageAsyncCancellationToken(
        string blobName,
        Stream blobInput,
        Stream blobOutput,
        CancellationToken token)
    {
        await blobInput.CopyToAsync(blobOutput, 4096, token);
    }
```

Importing namespaces

If you need to import namespaces, you can do so as usual, with the `using` clause.

C# Copy

```
using System.Net;
using System.Threading.Tasks;

public static Task<HttpResponseMessage> Run(HttpRequestMessage req, TraceWriter log)
```

The following namespaces are automatically imported and are therefore optional:

- `System`
- `System.Collections.Generic`
- `System.IO`

- `System.Linq`
- `System.Net.Http`
- `System.Threading.Tasks`
- `Microsoft.Azure.WebJobs`
- `Microsoft.Azure.WebJobs.Host`

Referencing external assemblies

For framework assemblies, add references by using the `#r "AssemblyName"` directive.

C# Copy

```
#r "System.Web.Http"

using System.Net;
using System.Net.Http;
using System.Threading.Tasks;

public static Task<HttpResponseMessage> Run(HttpRequestMessage req, TraceWriter log)
```

The following assemblies are automatically added by the Azure Functions hosting environment:

- `mscorlib`
- `System`
- `System.Core`
- `System.Xml`
- `System.Net.Http`
- `Microsoft.Azure.WebJobs`
- `Microsoft.Azure.WebJobs.Host`
- `Microsoft.Azure.WebJobs.Extensions`
- `System.Web.Http`
- `System.Net.Http.Formatting`

The following assemblies may be referenced by simple-name (for example, `#r "AssemblyName"`):

- `Newtonsoft.Json`
- `Microsoft.WindowsAzure.Storage`
- `Microsoft.ServiceBus`
- `Microsoft.AspNet.WebHooks.Receivers`

- `Microsoft.AspNet.WebHooks.Common`
- `Microsoft.Azure.NotificationHubs`

Referencing custom assemblies

To reference a custom assembly, you can use either a *shared* assembly or a *private* assembly:

- Shared assemblies are shared across all functions within a function app. To reference a custom assembly, upload the assembly to your function app, such as in a `bin` folder in the function app root.
- Private assemblies are part of a given function's context, and support side-loading of different versions. Private assemblies should be uploaded in a `bin` folder in the function directory. Reference using the file name, such as `#r "MyAssembly.dll"`.

For information on how to upload files to your function folder, see the following section on package management.

Watched directories

The directory that contains the function script file is automatically watched for changes to assemblies. To watch for assembly changes in other directories, add them to the `watchDirectories` list in host.json.

Using NuGet packages

To use NuGet packages in a C# function, upload a *project.json* file to the function's folder in the function app's file system. Here is an example *project.json* file that adds a reference to Microsoft.ProjectOxford.Face version 1.1.0:

JSON Copy

```
{
  "frameworks": {
    "net46":{
      "dependencies": {
        "Microsoft.ProjectOxford.Face": "1.1.0"
      }
    }
   }
}
```

Only the .NET Framework 4.6 is supported, so make sure that your *project.json* file specifies `net46` as shown here.

When you upload a *project.json* file, the runtime gets the packages and automatically adds references to the package assemblies. You don't need to add `#r "AssemblyName"` directives. To use the types defined in the NuGet packages, add the required `using` statements to your *run.csx* file.

In the Functions runtime, NuGet restore works by comparing `project.json` and `project.lock.json`. If the date and time stamps of the files **do not** match, a NuGet restore runs and NuGet downloads updated packages. However, if the date and time stamps of the files **do** match, NuGet does not perform a restore. Therefore, `project.lock.json` should not be deployed as it causes NuGet to skip package restore. To avoid deploying the lock file, add the `project.lock.json` to the `.gitignore` file.

To use a custom NuGet feed, specify the feed in a *Nuget.Config* file in the Function App root. For more information, see Configuring NuGet behavior.

Using a project.json file

1. Open the function in the Azure portal. The logs tab displays the package installation output.
2. To upload a project.json file, use one of the methods described in the How to update function app files in the Azure Functions developer reference topic.
3. After the *project.json* file is uploaded, you see output like the following example in your function's streaming log:

Copy

```
2016-04-04T19:02:48.745 Restoring packages.
2016-04-04T19:02:48.745 Starting NuGet restore
2016-04-04T19:02:50.183 MSBuild auto-detection: using msbuild version '14.0' from 'D:\P
2016-04-04T19:02:50.261 Feeds used:
2016-04-04T19:02:50.261 C:\DWASFiles\Sites\facavalfunctest\LocalAppData\NuGet\Cache
2016-04-04T19:02:50.261 https://api.nuget.org/v3/index.json
2016-04-04T19:02:50.261
2016-04-04T19:02:50.511 Restoring packages for D:\home\site\wwwroot\HttpTriggerCSharp1\
2016-04-04T19:02:52.800 Installing Newtonsoft.Json 6.0.8.
2016-04-04T19:02:52.800 Installing Microsoft.ProjectOxford.Face 1.1.0.
2016-04-04T19:02:57.095 All packages are compatible with .NETFramework,Version=v4.6.
2016-04-04T19:02:57.189
2016-04-04T19:02:57.189
2016-04-04T19:02:57.455 Packages restored.
```

Environment variables

To get an environment variable or an app setting value, use `System.Environment.GetEnvironmentVariable`, as shown in the following code example:

C# Copy

```
public static void Run(TimerInfo myTimer, TraceWriter log)
{
    log.Info($"C# Timer trigger function executed at: {DateTime.Now}");
    log.Info(GetEnvironmentVariable("AzureWebJobsStorage"));
    log.Info(GetEnvironmentVariable("WEBSITE_SITE_NAME"));
}

public static string GetEnvironmentVariable(string name)
{
    return name + ": " +
        System.Environment.GetEnvironmentVariable(name, EnvironmentVariableTarget.Proce
}
```

Reusing .csx code

You can use classes and methods defined in other *.csx* files in your *run.csx* file. To do that, use `#load` directives in your *run.csx* file. In the following example, a logging routine named `MyLogger` is shared in *myLogger.csx* and loaded into *run.csx* using the `#load` directive:

Example *run.csx*:

C# Copy

```
#load "mylogger.csx"

public static void Run(TimerInfo myTimer, TraceWriter log)
{
    log.Verbose($"Log by run.csx: {DateTime.Now}");
    MyLogger(log, $"Log by MyLogger: {DateTime.Now}");
}
```

Example *mylogger.csx*:

C# Copy

```
public static void MyLogger(TraceWriter log, string logtext)
{
    log.Verbose(logtext);
}
```

Using a shared *.csx* is a common pattern when you want to strongly type your arguments between functions using a POCO object. In the following simplified example, an HTTP trigger and queue trigger share a POCO object named `Order` to strongly type the order data:

Example *run.csx* for HTTP trigger:

C# Copy

```
#load "..\shared\order.csx"

using System.Net;

public static async Task<HttpResponseMessage> Run(Order req, IAsyncCollector<Order> out
{
    log.Info("C# HTTP trigger function received an order.");
    log.Info(req.ToString());
    log.Info("Submitting to processing queue.");

    if (req.orderId == null)
    {
        return new HttpResponseMessage(HttpStatusCode.BadRequest);
    }
    else
    {
        await outputQueueItem.AddAsync(req);
        return new HttpResponseMessage(HttpStatusCode.OK);
    }
}
```

Example *run.csx* for queue trigger:

C# Copy

```
#load "..\shared\order.csx"

using System;

public static void Run(Order myQueueItem, out Order outputQueueItem,TraceWriter log)
{
    log.Info($"C# Queue trigger function processed order...");
    log.Info(myQueueItem.ToString());

    outputQueueItem = myQueueItem;
}
```

Example *order.csx*:

C# Copy

```
public class Order
{
    public string orderId {get; set; }
    public string custName {get; set;}
    public string custAddress {get; set;}
    public string custEmail {get; set;}
    public string cartId {get; set; }

    public override String ToString()
    {
        return "\n{\n\torderId : " + orderId +
                  "\n\tcustName : " + custName +
                  "\n\tcustAddress : " + custAddress +
                  "\n\tcustEmail : " + custEmail +
                  "\n\tcartId : " + cartId + "\n}";
    }
}
```

You can use a relative path with the `#load` directive:

- `#load "mylogger.csx"` loads a file located in the function folder.
- `#load "loadedfiles\mylogger.csx"` loads a file located in a folder in the function folder.
- `#load "..\shared\mylogger.csx"` loads a file located in a folder at the same level as the function folder, that is, directly under *wwwroot*.

The `#load` directive works only with *.csx* (C# script) files, not with *.cs* files.

Binding at runtime via imperative bindings

In C# and other .NET languages, you can use an imperative binding pattern, as opposed to the *declarative* bindings in *function.json*. Imperative binding is useful when binding parameters need to be computed at runtime rather than design time. With this pattern, you can bind to supported input and output binding on-the-fly in your function code.

Define an imperative binding as follows:

- **Do not** include an entry in *function.json* for your desired imperative bindings.
- Pass in an input parameter `Binder binder` or `IBinder binder`.
- Use the following C# pattern to perform the data binding.

C# Copy

```
using (var output = await binder.BindAsync<T>(new BindingTypeAttribute(...)))
{
```

```
    ...
}
```

`BindingTypeAttribute` is the .NET attribute that defines your binding and `T` is the input or output type that's supported by that binding type. `T` also cannot be an `out` parameter type (such as `out JObject`). For example, the Mobile Apps table output binding supports six output types, but you can only use ICollector or IAsyncCollector for `T`.

The following example code creates a Storage blob output binding with blob path that's defined at run time, then writes a string to the blob.

C# Copy

```
using Microsoft.Azure.WebJobs;
using Microsoft.Azure.WebJobs.Host.Bindings.Runtime;

public static async Task Run(string input, Binder binder)
{
    using (var writer = await binder.BindAsync<TextWriter>(new BlobAttribute("samples-c
    {
        writer.Write("Hello World!!");
    }
}
```

BlobAttribute defines the Storage blob input or output binding, and TextWriter is a supported output binding type. In the previous code sample, the code gets the app setting for the function app's main Storage account connection string (which is `AzureWebJobsStorage`). You can specify a custom app setting to use for the Storage account by adding the StorageAccountAttribute and passing the attribute array into `BindAsync<T>()`. For example,

C# Copy

```
using Microsoft.Azure.WebJobs;
using Microsoft.Azure.WebJobs.Host.Bindings.Runtime;

public static async Task Run(string input, Binder binder)
{
    var attributes = new Attribute[]
    {
        new BlobAttribute("samples-output/path"),
        new StorageAccountAttribute("MyStorageAccount")
    };

    using (var writer = await binder.BindAsync<TextWriter>(attributes))
    {
        writer.Write("Hello World!");
```

```
    }
}
```

The following table lists the .NET attributes for each binding type and the packages in which they are defined.

Binding	Attribute	Add reference
Cosmos DB	`Microsoft.Azure.WebJobs.DocumentDBAttribute`	`#r "Microsoft.Azure.WebJobs.Extensions.DocumentDB"`
Event Hubs	`Microsoft.Azure.WebJobs.ServiceBus.EventHubAttribute`, `Microsoft.Azure.WebJobs.ServiceBusAccountAttribute`	`#r "Microsoft.Azure.Jobs.ServiceBus"`
Mobile Apps	`Microsoft.Azure.WebJobs.MobileTableAttribute`	`#r "Microsoft.Azure.WebJobs.Extensions.MobileApps"`
Notification Hubs	`Microsoft.Azure.WebJobs.NotificationHubAttribute`	`#r "Microsoft.Azure.WebJobs.Extensions.NotificationHubs"`
Service Bus	`Microsoft.Azure.WebJobs.ServiceBusAttribute`, `Microsoft.Azure.WebJobs.ServiceBusAccountAttribute`	`#r "Microsoft.Azure.WebJobs.ServiceBus"`
Storage queue	`Microsoft.Azure.WebJobs.QueueAttribute`, `Microsoft.Azure.WebJobs.StorageAccountAttribute`	
Storage blob	`Microsoft.Azure.WebJobs.BlobAttribute`, `Microsoft.Azure.WebJobs.StorageAccountAttribute`	
Storage table	`Microsoft.Azure.WebJobs.TableAttribute`, `Microsoft.Azure.WebJobs.StorageAccountAttribute`	