Extism .NET Host SDK

This repo houses the .NET SDK for integrating with the [Extism](#)⍈ runtime. Install this library into your host .NET applications to run Extism plugins.

Installation

This library depends on the native Extism runtime, we provide [native runtime packages](#)⍈ for all supported operating systems. You can install with: `nuget` `v1.1.0`

```
dotnet add package Extism.runtime.all
```

Then, add the [Extism.Sdk NuGet package](#)⍈ to your project: `nuget` `v1.0.1`

```
dotnet add package Extism.Sdk
```

Getting Started

This guide should walk you through some of the concepts in Extism and this .NET library.

First you should add a using statement for Extism:

C#:

```
using System;

using Extism.Sdk;
```

F#:

```
open System

open Extism.Sdk
```

Creating A Plug-in

The primary concept in Extism is the [plug-in](#)⍈. You can think of a plug-in as a code module stored in a `.wasm` file.

Since you may not have an Extism plug-in on hand to test, let's load a demo plug-in from the web:

C#:

```csharp
var manifest = new Manifest(new
UrlWasmSource("https://github.com/extism/plugins/releases/latest/download/count_vowe
ls.wasm"));

using var plugin = new Plugin(manifest, new HostFunction[] { }, withWasi: true);
```

F#:

```fsharp
let uri =
Uri("https://github.com/extism/plugins/releases/latest/download/count_vowels.wasm")
let manifest = Manifest(new UrlWasmSource(uri))

let plugin = new Plugin(manifest, Array.Empty<HostFunction>(), withWasi = true)
```

> **Note**: The schema for this manifest can be found here:
> https://extism.org/docs/concepts/manifest/⧉

Calling A Plug-in's Exports

This plug-in was written in Rust and it does one thing, it counts vowels in a string. As such,
it exposes one "export" function: count_vowels. We can call exports using Plugin.Call:

C#:

```csharp
var output = plugin.Call("count_vowels", "Hello, World!");
Console.WriteLine(output);
// => {"count": 3, "total": 3, "vowels": "aeiouAEIOU"}
```

F#:

```fsharp
let output = plugin.Call("count_vowels", "Hello, World!")
// => {"count": 3, "total": 3, "vowels": "aeiouAEIOU"}
```

All exports have a simple interface of optional bytes in, and optional bytes out. This plug-in
happens to take a string and return a JSON encoded string with a report of results.

Plug-in State

Plug-ins may be stateful or stateless. Plug-ins can maintain state b/w calls by the use of variables. Our count vowels plug-in remembers the total number of vowels it's ever counted in the "total" key in the result. You can see this by making subsequent calls to the export:

C#:

```
var output = plugin.Call("count_vowels", "Hello, World!");
Console.WriteLine(output);
// => {"count": 3, "total": 6, "vowels": "aeiouAEIOU"}

output = plugin.Call("count_vowels", "Hello, World!");
Console.WriteLine(output);
// => {"count": 3, "total": 9, "vowels": "aeiouAEIOU"}
```

F#:

```
let output1 = plugin.Call("count_vowels", "Hello, World!")
// => {"count": 3, "total": 6, "vowels": "aeiouAEIOU"}

let output2 = plugin.Call("count_vowels", "Hello, World!")
// => {"count": 3, "total": 9, "vowels": "aeiouAEIOU"}
```

These variables will persist until this plug-in is freed or you initialize a new one.

Configuration

Plug-ins may optionally take a configuration object. This is a static way to configure the plug-in. Our count-vowels plugin takes an optional configuration to change out which characters are considered vowels. Example:

C#:

```
var manifest = new Manifest(new UrlWasmSource("
<https://github.com/extism/plugins/releases/latest/download/count_vowels.wasm>"));

using var plugin = new Plugin(manifest, new HostFunction[] { }, withWasi: true);

var output = plugin.Call("count_vowels", "Yellow, World!");
Console.WriteLine(output);
// => {"count": 3, "total": 3, "vowels": "aeiouAEIOU"}

manifest = new Manifest(new UrlWasmSource("
```

```
     <https://github.com/extism/plugins/releases/latest/download/count_vowels.wasm>"))
    {
        Config = new Dictionary<string, string>
        {
            { "vowels", "aeiouyAEIOUY" }
        },
    };

    using var plugin2 = new Plugin(manifest, new HostFunction[] { }, withWasi: true);

    var output2 = plugin2.Call("count_vowels", "Yellow, World!");
    Console.WriteLine(output2);
    // => {"count": 4, "total": 4, "vowels": "aeiouAEIOUY"}
```

F#:

```
let uri =
Uri("https://github.com/extism/plugins/releases/latest/download/count_vowels.wasm")
let manifest = Manifest(new UrlWasmSource(uri))
manifest.Config <- dict [("vowels", "aeiouAEIOU")]

let plugin = new Plugin(manifest, Array.Empty<HostFunction>(), withWasi = true)

let output = plugin.Call("count_vowels", "Yellow, World!")
Console.WriteLine(output)
// => {"count": 3, "total": 3, "vowels": "aeiouAEIOU"}

let manifest2 =
    Manifest(new
UrlWasmSource(Uri("https://github.com/extism/plugins/releases/latest/download/count_
vowels.wasm")))
manifest2.Config <- dict [("vowels", "aeiouyAEIOUY")]

let plugin2 =
    new Plugin(manifest2, Array.Empty<HostFunction>(), withWasi = true)

let output2 = plugin2.Call("count_vowels", "Yellow, World!")
Console.WriteLine(output2)
// => {"count": 4, "total": 4, "vowels": "aeiouAEIOUY"}
```

Host Functions

Let's extend our count-vowels example a little bit: Instead of storing the `total` in an
ephemeral plug-in var, let's store it in a persistent key-value store!

Wasm can't use our KV store on it's own. This is where `Host Functions` come in.

[Host functions⧉](#) allow us to grant new capabilities to our plug-ins from our application. They are simply some Go functions you write which can be passed down and invoked from any language inside the plug-in.

Let's load the manifest like usual but load up this `count_vowels_kvstore` plug-in:

C#:

```
var manifest = new Manifest(new
UrlWasmSource("https://github.com/extism/plugins/releases/latest/download/count_vowe
ls_kvstore.wasm"));
```

F#:

```
let manifest = Manifest(new
UrlWasmSource(Uri("https://github.com/extism/plugins/releases/latest/download/count_
vowels_kvstore.wasm")))
```

> *Note*: The source code for this is [here⧉](#) and is written in rust, but it could be written in any of our PDK languages.

Unlike our previous plug-in, this plug-in expects you to provide host functions that satisfy our its import interface for a KV store.

We want to expose two functions to our plugin, `void kv_write(key string, value byte[])` which writes a bytes value to a key and `byte[] kv_read(key string)` which reads the bytes at the given `key`.

C#:

```
// pretend this is Redis or something :)
var kvStore = new Dictionary<string, byte[]>();

var functions = new[]
{
    HostFunction.FromMethod("kv_read", IntPtr.Zero, (CurrentPlugin plugin, long
keyOffset) =>
    {
        var key = plugin.ReadString(keyOffset);
        if (!kvStore.TryGetValue(key, out var value))
```

```csharp
        {
            value = new byte[] { 0, 0, 0, 0 };
        }

        Console.WriteLine($"Read {BitConverter.ToUInt32(value)} from key={key}");
        return plugin.WriteBytes(value);
    }),

    HostFunction.FromMethod("kv_write", IntPtr.Zero, (CurrentPlugin plugin, long
keyOffset, long valueOffset) =>
    {
        var key = plugin.ReadString(keyOffset);
        var value = plugin.ReadBytes(valueOffset);

        Console.WriteLine($"Writing value={BitConverter.ToUInt32(value)} from
key={key}");
        kvStore[key] = value.ToArray();
    })
};
```

F#:

```fsharp
let kvStore = new Dictionary<string, byte[]>()

let functions =
    [|
        HostFunction.FromMethod("kv_read", IntPtr.Zero, fun (plugin: CurrentPlugin)
(offs: int64) ->
            let key = plugin.ReadString(offs)
            let value =
                match kvStore.TryGetValue(key) with
                | true, v -> v
                | _ -> [| 0uy; 0uy; 0uy; 0uy |] // Default value if key not found

            Console.WriteLine($"Read {BitConverter.ToUInt32(value, 0)} from
key={key}")
            plugin.WriteBytes(value)
        )

        HostFunction.FromMethod("kv_write", IntPtr.Zero, fun (plugin: CurrentPlugin)
(kOffs: int64) (vOffs: int64) ->
            let key = plugin.ReadString(kOffs)
            let value = plugin.ReadBytes(vOffs).ToArray()

            Console.WriteLine($"Writing value={BitConverter.ToUInt32(value, 0)} from
```

```
key={key}")
            kvStore.[key] <- value
        )
    |]
```

> *Note*: In order to write host functions you should get familiar with the methods on the CurrentPlugin type. The `plugin` parameter is an instance of this type.

We need to pass these imports to the plug-in to create them. All imports of a plug-in must be satisfied for it to be initialized:

C#:

```
using var plugin = new Plugin(manifest, functions, withWasi: true);

var output = plugin.Call("count_vowels", "Hello World!");

Console.WriteLine(output);
// => Read 0 from key=count-vowels"
// => Writing value=3 from key=count-vowels"
// => {"count": 3, "total": 3, "vowels": "aeiouAEIOU"}

output = plugin.Call("count_vowels", "Hello World!");

Console.WriteLine(output);
// => Read 3 from key=count-vowels"
// => Writing value=6 from key=count-vowels"
// => {"count": 3, "total": 6, "vowels": "aeiouAEIOU"}
```

F#:

```
let plugin = new Plugin(manifest, functions, withWasi = true)

let output = plugin.Call("count_vowels", "Hello World!")
printfn "%s" output
// => Read 0 from key=count-vowels
// => Writing value=3 from key=count-vowels
// => {"count": 3, "total": 3, "vowels": "aeiouAEIOU"}

let output2 = plugin.Call("count_vowels", "Hello World!")
printfn "%s" output2
// => Read 3 from key=count-vowels
```

```
 // => Writing value=6 from key=count-vowels
 // => {"count": 3, "total": 6, "vowels": "aeiouAEIOU"}
```

API Docs

Please see our [API docs](#) for detailed information on each type.

Namespace Extism.Sdk

Classes

[ByteArrayWasmSource](#)

Wasm Source represented by raw bytes.

[CurrentPlugin](#)

Represents the current plugin. Can only be used within [HostFunction](#)s.

[ExtismException](#)

Represents errors that occur during calling Extism functions.

[HostFunction](#)

A function provided by the host that plugins can call.

[Manifest](#)

The manifest is a description of your plugin and some of the runtime constraints to apply to it. You can think of it as a blueprint to build your plugin.

[MemoryOptions](#)

Configures memory for the Wasm runtime. Memory is described in units of pages (64KB) and represent contiguous chunks of addressable memory.

[PathWasmSource](#)

Wasm Source represented by a file referenced by a path.

[Plugin](#)

Represents a WASM Extism plugin.

[UrlWasmSource](#)

Wasm Source represented by a file referenced by a path.

[WasmSource](#)

A named Wasm source.

Enums

[HttpMethod](#)

HTTP defines a set of request methods to indicate the desired action to be performed for a given resource.

[LogLevel](#)

Extism Log Levels

Delegates

ExtismFunction

A host function signature.

LoggingSink

Custom logging callback.