

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 v1.9.1`

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

Then, add the [Extism.Sdk NuGet package](#) to your project: `nuget v1.2.1`

```
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#:

```
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);
```

F#:

```
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#:

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

F#:

```
let output = plugin.Call("count_vowels", "Hello, World!")
printfn "%s" output
// => {"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.

Precompiling plugins

If you're going to create more than one instance of the same plugin, we recommend pre-compiling the plugin and instantiate them:

C#:

```
var manifest = new Manifest(new PathWasmSource("/path/to/plugin.wasm"), "main");

// pre-compile the wasm file
using var compiledPlugin = new CompiledPlugin(_manifest, [], withWasi: true);

// instantiate plugins
using var plugin = compiledPlugin.Instantiate();
```

F#:

```
// Create manifest
let manifest = Manifest(PathWasmSource("/path/to/plugin.wasm"))

// Pre-compile the wasm file
use compiledPlugin = new CompiledPlugin(manifest, Array.empty<HostFunction>,
withWasi = true)

// Instantiate plugins
use plugin = compiledPlugin.Instantiate()
```

This can have a dramatic effect on performance*:

```
// * Summary *

BenchmarkDotNet v0.14.0, Windows 11 (10.0.22631.4460/23H2/2023Update/SunValley3)
13th Gen Intel Core i7-1365U, 1 CPU, 12 logical and 10 physical cores
.NET SDK 9.0.100
[Host] : .NET 9.0.0 (9.0.24.52809), X64 RyuJIT AVX2
DefaultJob : .NET 9.0.0 (9.0.24.52809), X64 RyuJIT AVX2
```

Method	Mean	Error	StdDev
CompiledPluginInstantiate	266.2 ms	6.66 ms	19.11 ms
PluginInstantiate	27,592.4 ms	635.90 ms	1,783.12 ms

*: See [the complete benchmark](#)

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!")
printfn "%s" output1
// => {"count": 3, "total": 6, "vowels": "aeiouAEIOU"}

let output2 = plugin.Call("count_vowels", "Hello, World!")
printfn "%s" output2
// => {"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!")
printfn "%s" 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_vowels_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", null, (CurrentPlugin plugin, long
    keyOffset) =>
```

```

{
    var key = plugin.ReadString(keyOffset);
    if (!kvStore.TryGetValue(key, out var value))
    {
        value = new byte[] { 0, 0, 0, 0 };
    }

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

HostFunction.FromMethod("kv_write", null, (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#:

```

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

let functions =
[|
    HostFunction.FromMethod("kv_read", null, 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", null, 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"}
```

Passing context to host functions

Extism provides two ways to pass context to host functions:

UserData

UserData allows you to associate persistent state with a host function that remains available across all calls to that function. This is useful for maintaining configuration or state that should be available throughout the lifetime of the host function.

C#:

```
var hostFunc = new HostFunction(
    "hello_world",
    new[] { ExtismValType.PTR },
    new[] { ExtismValType.PTR },
    "Hello again!", // <= userData, this can be any .NET object
    (CurrentPlugin plugin, Span<ExtismVal> inputs, Span<ExtismVal> outputs) => {
        var text = plugin.GetUserData<string>(); // <= We're retrieving the
data back
        // Use text...
    });
```

F#:

```
// Create host function with userData
let hostFunc = new HostFunction(
    "hello_world",
    [| ExtismValType.PTR |],
    [| ExtismValType.PTR |],
    "Hello again!", // userData can be any .NET object
    (fun (plugin: CurrentPlugin) (inputs: Span<ExtismVal>) (outputs:
Span<ExtismVal>) ->
        // Retrieve the userData
        let text = plugin.GetUserData<string>()
        printfn "%s" text // Prints: "Hello again!"
        // Rest of function implementation...
    ))
```

The `userData` object is preserved for the lifetime of the host function and can be retrieved in any call using `CurrentPlugin.GetUserData<T>()`. If no `userData` was provided, `GetUserData<T>()` will return the default value for type `T`.

Call Host Context

Call Host Context provides a way to pass per-call context data when invoking a plugin function. This is useful when you need to provide data specific to a particular function call rather than data that persists across all calls.

C#:

```
// Pass context for specific call
var context = new Dictionary<string, object> { { "requestId", 42 } };
var result = plugin.CallWithHostContext("function_name", inputData, context);

// Access in host function
void HostFunction(CurrentPlugin plugin, Span<ExtismVal> inputs,
Span<ExtismVal> outputs)
{
    var context = plugin.GetCallHostContext<Dictionary<string, object>>();
    // Use context...
}
```

F#:

```
// Create context for specific call
let context = dict [ "requestId", box 42 ]

// Call plugin with context
let result = plugin.CallWithHostContext("function_name", inputData, context)

// Access context in host function
let hostFunction (plugin: CurrentPlugin) (inputs: Span<ExtismVal>) (outputs:
Span<ExtismVal>) =
    match plugin.GetCallHostContext<IDictionary<string, obj>>() with
    | null -> printfn "No context available"
    | context ->
        let requestId = context.[ "requestId" ] :?> int
        printfn "Request ID: %d" requestId
```

Host context is only available for the duration of the specific function call and can be retrieved using `CurrentPlugin.GetHostContext<T>()`. If no context was provided for the call, `GetHostContext<T>()` will return the default value for type `T`.

Fuel limit

The fuel limit feature allows you to constrain plugin execution by limiting the number of instructions it can execute. This provides a safeguard against infinite loops or excessive resource consumption.

Setting a fuel limit

Set the fuel limit when initializing a plugin:

C#:

```
var manifest = new Manifest(...);
var options = new PluginInitializationOptions {
    FuelLimit = 1000, // plugin can execute 1000 instructions
    WithWasi = true
};

var plugin = new Plugin(manifest, functions, options);
```

F#:

```
let manifest = Manifest(PathWasmSource("/path/to/plugin.wasm"))
let options = PluginInitializationOptions(
    FuelLimit = Nullable<int64>(1000L), // plugin can execute 1000 instructions
    WithWasi = true
)

use plugin = new Plugin(manifest, Array.empty<HostFunction>, options)
```

When the fuel limit is exceeded, the plugin execution is terminated and an `ExtismException` is thrown containing "fuel" in the error message.

API Docs

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

Namespace Extism.Sdk

Classes

[ByteArrayWasmSource](#)

Wasm Source represented by raw bytes.

[CompiledPlugin](#)

A pre-compiled plugin ready to be instantiated.

[CurrentPlugin](#)

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

[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.

[PluginIntializationOptions](#)

Options for initializing a 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.