.NET PDK

This library can be used to write Extism Plug-ins in C# and F#.

> NOTE: This is an experimental PDK. We'd love to hear your feedback.

Prerequisites

1. .NET SDK 8: https://dotnet.microsoft.com/en-us/download/dotnet/8.0
2. WASI Workload:

```
dotnet workload install wasi-experimental
```

3. Extract WASI SDK into local file system and set the `WASI_SDK_PATH` environment variable to point to it

Install

Create a new project and add this nuget package to your project:

```
dotnet new wasiconsole -o MyPlugin
# OR, for F#: dotnet new console -o MyPlugin -lang F#
cd MyPlugin
dotnet add package Extism.Pdk
```

Update your `MyPlugin.csproj`/`MyPlugin.fsproj` as follows:

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <TargetFramework>net8.0</TargetFramework>
    <RuntimeIdentifier>wasi-wasm</RuntimeIdentifier>
    <OutputType>Exe</OutputType>
  </PropertyGroup>
</Project>
```

Getting Started

The goal of writing an Extism plug-in is to compile your C#/F# code to a Wasm module with exported functions that the host application can invoke. The first thing you should understand is creating an export. Let's write a simple program that exports a greet function

which will take a name as a string and return a greeting string. Paste this into your Program.cs/Program.fs:

C#:

```
using System;
using System.Runtime.InteropServices;
using System.Text.Json;
using Extism;

namespace MyPlugin;
public class Functions
{
    public static void Main()
    {
        // Note: a `Main` method is required for the app to compile
    }

    [UnmanagedCallersOnly(EntryPoint = "greet")]
    public static int Greet()
    {
        var name = Pdk.GetInputString();
        var greeting = $"Hello, {name}!";
        Pdk.SetOutput(greeting);

        return 0;
    }
}
```

F#:

```
module MyPlugin

open System
open System.Runtime.InteropServices
open System.Text.Json
open Extism

[<UnmanagedCallersOnly(EntryPoint = "greet")>]
let Greet () : int32 =
    let name = Pdk.GetInputString()
    let greeting = $"Hello, {name}!"
    Pdk.SetOutput(greeting)
    0
```

```
[<EntryPoint>]
let Main args  =
    // Note: an `EntryPoint` function is required for the app to compile
    0
```

Some things to note about this code:

1. The `[UnmanagedCallersOnly(EntryPoint = "greet")]` is required, this marks the `Greet` function as an export with the name `greet` that can be called by the host. `EntryPoint` is optional.
2. We need a `Main` but it's unused. If you do want to use it, it's exported as a function called `_start`.
3. Exports in the .NET PDK are coded to the raw ABI. You get parameters from the host by calling `Pdk.GetInput*` functions and you send returns back with the `Pdk.SetOutput` functions.
4. An Extism export expects an `Int32` return code. `0` is success and `1` is a failure.

Compile with this command:

```
dotnet build
```

This will create a `MyPlugin.wasm` file in `bin/Debug/net8.0/wasi-wasm/AppBundle`. Now, you can try out your plugin by using any of the Extism SDKs or by using Extism CLI's `run` command:

```
extism call .\bin\Debug\net8.0\wasi-wasm\AppBundle\MyPlugin.wasm greet --input
"Benjamin" --wasi
# => Hello, Benjamin!
```

> **Note:** Currently wasi must be provided for all .NET plug-ins even if they don't need system access.

More Exports: Error Handling

Suppose we want to re-write our greeting function to never greet Benjamis. We can use `Pdk.SetError`:

C#:

```
[UnmanagedCallersOnly(EntryPoint = "greet")]
public static int Greet()
```

```
{
    var name = Pdk.GetInputString();
    if (name == "Benjamin")
    {
        Pdk.SetError("Sorry, we don't greet Benjamins!");
        return 1;
    }

    var greeting = $"Hello, {name}!";
    Pdk.SetOutput(greeting);

    return 0;
}
```

F#:

```
[<UnmanagedCallersOnly(EntryPoint = "greet")>]
let Greet () =
    let name = Pdk.GetInputString()
    if name = "Benjamin" then
        Pdk.SetError("Sorry, we don't greet Benjamins!")
        1
    else
        let greeting = $"Hello, {name}!"
        Pdk.SetOutput(greeting)
        0
```

Now when we try again:

```
extism call .\bin\Debug\net8.0\wasi-wasm\AppBundle\MyPlugin.wasm greet --
input="Benjamin" --wasi
# => Error: Sorry, we don't greet Benjamins!
echo $? # print last status code
# => 1
extism call .\bin\Debug\net8.0\wasi-wasm\AppBundle\MyPlugin.wasm greet --
input="Zach" --wasi
# => Hello, Zach!
echo $?
# => 0
```

We can also throw a normal .NET Exception:

```
var name = Pdk.GetInputString();
if (name == "Benjamin")
{
    throw new ArgumentException("Sorry, we don't greet Benjamins!");
}
```

Now when we try again:

```
extism call .\bin\Debug\net8.0\wasi-wasm\AppBundle\MyPlugin.wasm greet --
input="Benjamin" --wasi
# => Error: System.ArgumentException: Sorry, we don't greet Benjamins!
   at MyPlugin.Functions.Greet()
```

Json

Extism export functions simply take bytes in and bytes out. Those can be whatever you want them to be. A common and simple way to get more complex types to and from the host is with json:

C#:

```
[JsonSerializable(typeof(Add))]
[JsonSerializable(typeof(Sum))]
public partial class SourceGenerationContext : JsonSerializerContext {}

public record Add(int a, int b);
public record Sum(int Result);

public static class Functions
{
    [UnmanagedCallersOnly]
    public static int add()
    {
        var parameters = Pdk.GetInputJson(SourceGenerationContext.Default.Add);
        var sum = new Sum(parameters.a + parameters.b);
        Pdk.SetOutputJson(sum, SourceGenerationContext.Default.Sum);
        return 0;
    }
}
```

F#:

```
[<UnmanagedCallersOnly>]
let add () =
    let inputJson = Pdk.GetInputString()
    let jsonData = JsonDocument.Parse(inputJson).RootElement
    let a = jsonData.GetProperty("a").GetInt32()
    let b = jsonData.GetProperty("b").GetInt32()
    let result = a + b
    let outputJson = $"{{ \"Result\": {result} }}"

    Pdk.SetOutput(outputJson)
    0
```

Note: For F#, please make sure the System.Text.Json NuGet package is installed in your project.

```
extism call .\bin\Debug\net8.0\wasi-wasm\AppBundle\MyPlugin.wasm --wasi add --
input='{"a": 20, "b": 21}'
# => {"Result":41}
```

Note: When enabling trimming, make sure you use the source generation as reflection is disabled in that mode.

Configs

Configs are key-value pairs that can be passed in by the host when creating a plug-in. These can be useful to statically configure the plug-in with some data that exists across every function call. Here is a trivial example using Pdk.TryGetConfig:

C#:

```
[UnmanagedCallersOnly(EntryPoint = "greet")]
public static int Greet()
{
    if (!Pdk.TryGetConfig("user", out var user)) {
        throw new InvalidOperationException("This plug-in requires a 'user' key in
the config");
    }

    var greeting = $"Hello, {user}!";
    Pdk.SetOutput(greeting);
```

```
    return 0;
}
```

F#:

```
[<UnmanagedCallersOnly(EntryPoint = "greet")>]
let Greet () =
    match Pdk.TryGetConfig "user" with
    | true, user ->
        let greeting = $"Hello, {user}!"
        Pdk.SetOutput(greeting)
        0
    | false, _ ->
        failwith "This plug-in requires a 'user' key in the config"
```

To test it, the Extism CLI has a --config option that lets you pass in key=value pairs:

```
extism call .\bin\Debug\net8.0\wasi-wasm\AppBundle\MyPlugin.wasm --wasi greet --
config user=Benjamin
# => Hello, Benjamin!
```

Variables

Variables are another key-value mechanism but it's a mutable data store that will persist across function calls. These variables will persist as long as the host has loaded and not freed the plug-in.

C#:

```
[UnmanagedCallersOnly]
public static int count()
{
    int count = 0;
    if (Pdk.TryGetVar("count", out var memoryBlock))
    {
        count = BitConverter.ToInt32(memoryBlock.ReadBytes());
    }
    count += 1;
    Pdk.SetVar("count", BitConverter.GetBytes(count));
    Pdk.SetOutput(count.ToString());
    return 0;
}
```

F#:

```
[<UnmanagedCallersOnly>]
let count () =

    let count =
        match Pdk.TryGetVar "count" with
        | true, buffer ->
            BitConverter.ToInt32(buffer.ReadBytes())
        | false, _ ->
            0

    let count = count + 1

    Pdk.SetVar("count", BitConverter.GetBytes(count))
    Pdk.SetOutput(count.ToString())

    0
```

From Extism CLI:

```
extism call .\bin\Debug\net8.0\wasi-wasm\AppBundle\MyPlugin.wasm --wasi count --
loop 3
1
2
3
```

HTTP

Sometimes it is useful to let a plug-in make HTTP calls:

C#:

```
[UnmanagedCallersOnly]
public static int http_get()
{
    // create an HTTP Request (withuot relying on WASI), set headers as needed
    var request = new HttpRequest("https://jsonplaceholder.typicode.com/todos/1")
    {
        Method = HttpMethod.GET,
    };
    request.Headers.Add("some-name", "some-value");
    request.Headers.Add("another", "again");
    var response = Pdk.SendRequest(request);
```

```
    Pdk.SetOutput(response.Body);
    return 0;
}
```

F#:

```
[<UnmanagedCallersOnly>]
let http_get () =
    let request = HttpRequest("https://jsonplaceholder.typicode.com/todos/1")
    request.Headers.Add("some-name", "some-value")
    request.Headers.Add("another", "again")

    let response = Pdk.SendRequest(request)
    Pdk.SetOutput(response.Body)

    0
```

From Extism CLI:

```
extism call .\bin\Debug\net8.0\wasi-wasm\AppBundle\MyPlugin.wasm --wasi http_get --
allow-host='*.typicode.com'
{
  "userId": 1,
  "id": 1,
  "title": "delectus aut autem",
  "completed": false
}
```

> **NOTE**: `HttpClient` doesn't work in Wasm yet.

Imports (Host Functions)

Like any other code module, Wasm not only let's you export functions to the outside world, you can import them too. Host Functions allow a plug-in to import functions defined in the host. For example, if you host application is written in Go, it can pass a Go function down to your Go plug-in where you can invoke it.

This topic can get fairly complicated and we have not yet fully abstracted the Wasm knowledge you need to do this correctly. So we recommend reading our concept doc on Host Functions before you get started.

A Simple Example

Host functions have a similar interface as exports. You just need to declare them as extern on the top of your `Program.cs`/`Program.fs`. You only declare the interface as it is the host's responsibility to provide the implementation:

C#:

```
[DllImport("extism", EntryPoint = "a_go_func")]
public static extern ulong GoFunc(ulong offset);
[UnmanagedCallersOnly]
public static int hello_from_go()
{
    var message = "An argument to send to Go";
    using var block = Pdk.Allocate(message);
    var ptr = GoFunc(block.Offset);
    var response = MemoryBlock.Find(ptr).ReadString();
    Pdk.SetOutput(response);
    return 0;
}
```

F#:

```
[<DllImport("extism", EntryPoint = "a_go_func")>]
extern uint64 GoFunc(uint64 offset)

[<UnmanagedCallersOnly>]
let hello_from_go () =
    let message = "An argument to send to Go"
    use block = Pdk.Allocate(message)

    let ptr = GoFunc(block.Offset)
    let response = MemoryBlock.Find ptr
    Pdk.SetOutput(response)

    0
```

Testing it out

We can't really test this from the Extism CLI as something must provide the implementation. So let's write out the Go side here. Check out the docs for Host SDKs to implement a host function in a language of your choice.

```
ctx := context.Background()
config := extism.PluginConfig{
```

```
    EnableWasi: true,
}

go_func := extism.NewHostFunctionWithStack(
    "a_go_func",
    func(ctx context.Context, p *extism.CurrentPlugin, stack []uint64) {
        input, err := p.ReadString(stack[0])
        if err != nil {
            panic(err)
        }

        fmt.Println("Hello from Go!")

        offs, err := p.WriteString(input + "!")
        if err != nil {
            panic(err)
        }

        stack[0] = offs
    },
    []api.ValueType{api.ValueTypeI64},
    []api.ValueType{api.ValueTypeI64},
)
```

Now when we load the plug-in we pass the host function:

```
manifest := extism.Manifest{
    Wasm: []extism.Wasm{
        extism.WasmFile{
            Path: "/path/to/plugin.wasm",
        },
    },
}

plugin, err := extism.NewPlugin(ctx, manifest, config,
[]extism.HostFunction{go_func})

if err != nil {
    fmt.Printf("Failed to initialize plugin: %v\n", err)
    os.Exit(1)
}

_, out, err := plugin.Call("hello_from_go", []byte("Hello, World!"))
fmt.Println(string(out))
```

```
go run .
# => Hello from Go!
# => An argument to send to Go!
```

Referenced Assemblies

Methods in referenced assemblies that are decorated with `[DllImport]` and `[UnmanagedCallersOnly]` are imported and exported respectively.

Note: The library imports/exports are ignored if the app doesn't call at least one method from the library.

For example, if we have a library that contains this class:

```
namespace MessagingBot.Pdk;
public class Events
{
    // This function will be imported  by all WASI apps that reference this library
    [DllImport("env", EntryPoint = "send_message")]
    public static extern void SendMessage(ulong offset);

    // You can wrap the imports in your own functions to make them easier to use
    public static void SendMessage(string message)
    {
        using var block = Extism.Pdk.Allocate(message);
        SendMessage(block.Offset);
    }

    // This function will be exported by all WASI apps that reference this library
    [UnmanagedCallersOnly]
    public static void message_received()
    {
        var message = Extism.Pdk.GetInputString();
        // TODO: do stuff with message
    }
}
```

Then, we can reference the library in a WASI app and use the functions:

```
using MessagingBot.Pdk;

Events.SendMessage("Hello World!");
```

This is useful when you want to provide a common set of imports and exports that are specific to your use case.

Optimize Size

Normally, the .NET runtime is very conservative when trimming and includes a lot of metadata for debugging and exception purposes. We have enabled some options in Release mode by default that would make the resulting binary smaller (6mb for a hello world sample vs 20mb in debug mode).

If you have imports in referenced assemblies, make sure you mark them as roots so that they don't get trimmed:

```
<ItemGroup>
    <TrimmerRootAssembly Include="SampleLib" />
</ItemGroup>
```

And then, run:

```
dotnet publish -c Release
```

Now, you'll have a smaller `.wasm` file in `bin\Release\net8.0\wasi-wasm\AppBundle`.

For more details, refer to the official documentation.

Reach Out!

Have a question or just want to drop in and say hi? Hop on the Discord!

API Docs

Please see our API docs for detailed information on each type.

Namespace Extism

Classes

HttpRequest
: An HTTP request

HttpResponse
: Response from an HTTP call

MemoryBlock
: A block of allocated memory.

Pdk
: Provides interop functions for communication between guests and the host.

Enums

HttpMethod
: HTTP Method

LogLevel
: Log level