In this lab, we will learn how to create a basic Wasm plugin and deploy it to workloads running in the Kubernetes cluster.

We’ll be using [TinyGo](https://tinygo.org/) and [Proxy Wasm Go SDK](https://github.com/tetratelabs/proxy-wasm-go-sdk) to build the Wasm plugin.

**Installing TinyGo**

TinyGo powers the SDK we’ll use as Wasm isn’t yet supported by the official Go compiler.

Let’s download and install the TinyGo:

wget https://github.com/tinygo-org/tinygo/releases/download/v0.27.0/tinygo\_0.27.0\_amd64.deb

sudo dpkg -i tinygo\_0.27.0\_amd64.deb

Run tinygo version to verify that the installation succeeded:

$ tinygo version

tinygo version 0.27.0 linux/amd64 (using go version go1.20.4 and LLVM version 15.0.0)

**Scaffolding the Wasm plugin**

We’ll start by creating a new folder for our extension, initializing the Go module, and downloading the SDK dependency:

mkdir wasm-extension && cd wasm-extension

go mod init wasm-extension

Next, let’s create the main.go file where the code for the Wasm plugin will live. The plugin reads the additional response headers (key/value pairs) we’ll provide through the configuration in the WasmPlugin resource. Any values set in the configuration will then be added to the response.

Here’s what the code looks like:

package main

import (

"github.com/valyala/fastjson"

"github.com/tetratelabs/proxy-wasm-go-sdk/proxywasm"

"github.com/tetratelabs/proxy-wasm-go-sdk/proxywasm/types"

)

func main() {

proxywasm.SetVMContext(&vmContext{})

}

// Override types.DefaultPluginContext.

func (ctx pluginContext) OnPluginStart(pluginConfigurationSize int) types.OnPluginStartStatus {

data, err := proxywasm.GetPluginConfiguration()

if err != nil {

proxywasm.LogCriticalf("error reading plugin configuration: %v", err)

}

var p fastjson.Parser

v, err := p.ParseBytes(data)

if err != nil {

proxywasm.LogCriticalf("error parsing configuration: %v", err)

}

obj, err := v.Object()

if err != nil {

proxywasm.LogCriticalf("error getting object from json value: %v", err)

}

obj.Visit(func(k []byte, v \*fastjson.Value) {

ctx.additionalHeaders[string(k)] = string(v.GetStringBytes())

})

return types.OnPluginStartStatusOK

}

type vmContext struct {

// Embed the default VM context here,

// so that we don't need to reimplement all the methods.

types.DefaultVMContext

}

// Override types.DefaultVMContext.

func (\*vmContext) NewPluginContext(contextID uint32) types.PluginContext {

return &pluginContext{contextID: contextID, additionalHeaders: map[string]string{}}

}

type pluginContext struct {

// Embed the default plugin context here,

// so that we don't need to reimplement all the methods.

types.DefaultPluginContext

additionalHeaders map[string]string

contextID uint32

}

// Override types.DefaultPluginContext.

func (ctx \*pluginContext) NewHttpContext(contextID uint32) types.HttpContext {

proxywasm.LogInfo("NewHttpContext")

return &httpContext{contextID: contextID, additionalHeaders: ctx.additionalHeaders}

}

type httpContext struct {

// Embed the default http context here,

// so that we don't need to reimplement all the methods.

types.DefaultHttpContext

contextID uint32

additionalHeaders map[string]string

}

func (ctx \*httpContext) OnHttpResponseHeaders(numHeaders int, endOfStream bool) types.Action {

proxywasm.LogInfo("OnHttpResponseHeaders")

for key, value := range ctx.additionalHeaders {

if err := proxywasm.AddHttpResponseHeader(key, value); err != nil {

proxywasm.LogCriticalf("failed to add header: %v", err)

return types.ActionPause

}

proxywasm.LogInfof("header set: %s=%s", key, value)

}

return types.ActionContinue

}

Save the above to main.go, then use TinyGo to build the plugin:

# Download the dependencies

go mod tidy

# Build the wasm file

tinygo build -o main.wasm -scheduler=none -target=wasi main.go

The next step is creating the Dockerfile, building the Wasm plugin image, and pushing it to the registry. First, let’s create the Dockerfile with the following contents:

FROM scratch

COPY main.wasm ./plugin.wasm

Since we’ve already built the main.wasm file, we can now use Docker to build and push the Wasm plugin to the registry:

docker build -t $REPOSITORY/wasm:v1 . --push

**Note:** $REPOSITORY is the name of the repository you’ve created in the registry.

With the Wasm plugin in the registry, we can now craft the WasmPlugin resource:

apiVersion: extensions.istio.io/v1alpha1

kind: WasmPlugin

metadata:

name: wasm-example

namespace: default

spec:

selector:

matchLabels:

app: httpbin

url: oci://$REPOSITORY/wasm:v1

pluginConfig:

header\_1: first\_header

header\_2: second\_header

Before saving the above YAML, replace the [REPOSITORY] with the name of your repository. Once replaced, save the YAML to wasm-plugin.yaml and then use the kubectl command to create the WasmPlugin resource:

kubectl apply -f wasm-plugin.yaml

We’ll deploy a sample workload to try out the Wasm plugin. We’ll use the httpbin sample from the Istio distribution. Make sure the default namespace is labeled for Istio sidecar injection (kubectl label ns default istio-injection=enabled) and then deploy the httpbin workload.

kubectl apply -f https://raw.githubusercontent.com/istio/istio/master/samples/httpbin/httpbin.yaml

Before continuing, check that the httpbin Pod is up and running:

$ kubectl get pod

NAME READY STATUS RESTARTS AGE

httpbin-66cdbdb6c5-4pv44 2/2 Running 1 11m

You can look at the logs from the istio-proxy container to see if something went wrong with downloading the Wasm plugin.

Let’s try out the deployed Wasm plugin!

We will use the Istio sleep sample as the client. Deploy it:

kubectl apply -f https://raw.githubusercontent.com/istio/istio/master/samples/sleep/sleep.yaml

We can now make request from sleep to one of the httpbin endpoints, say http://httpbin:8000/get and inspect the Http response headers:

SLEEP\_POD=$(kubectl get pod -l app=sleep -ojsonpath='{.items[0].metadata.name}')

kubectl exec $SLEEP\_POD -- curl -s --head httpbin:8000/get

HTTP/1.1 200 OK

server: envoy

content-type: application/json

content-length: 604

header\_1: first\_header

header\_2: second\_header

...

In the output, you can see that the Wasm plugin added the two headers set in the configuration to the response.

* **Dockerfile**41 BYTES

**[DOWNLOADOPENS IN A NEW WINDOW](https://import.cdn.thinkific.com/415508/courses/1078293/-230608-120141.Dockerfile" \t "_blank)**

* **wasm-plugin.yaml**268 BYTES

**[DOWNLOADOPENS IN A NEW WINDOW](https://import.cdn.thinkific.com/415508/courses/1078293/wasmplugin-230608-120141.yaml" \t "_blank)**

* **httpbin.yaml**707 BYTES

**[DOWNLOADOPENS IN A NEW WINDOW](https://import.cdn.thinkific.com/415508/courses/1078293/httpbin-230608-120141.yaml" \t "_blank)**

* **main.go**2.33 KB

**[DOWNLOADOPENS IN A NEW WINDOW](https://import.cdn.thinkific.com/415508/courses/1078293/main-230608-120141.go" \t "_blank)**

COMPLETE &