Francesco Romani

Software Engineer

IN THE BEGINNING IT WAS ABOUT THE METAL NOW IS ABOUT THE CLOUD: EXPLORING WEBASSEMBLY IN CLOUD ENVIRONMENTS

Francesco Romani

Software Engineer

WALL OF TEXT TITLE

Seriously, it was a pretty long talk title

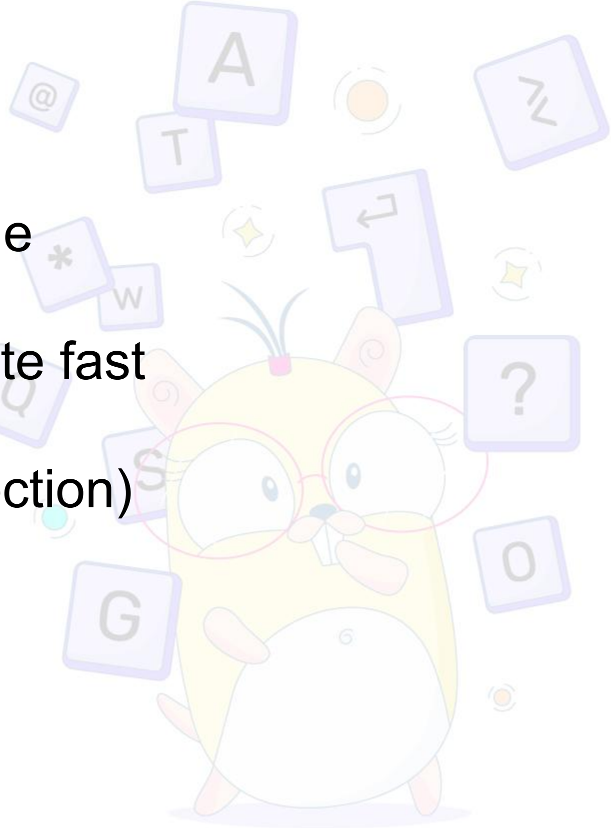
SETTING EXPECTATIONS

Exploring from the system/platform angle

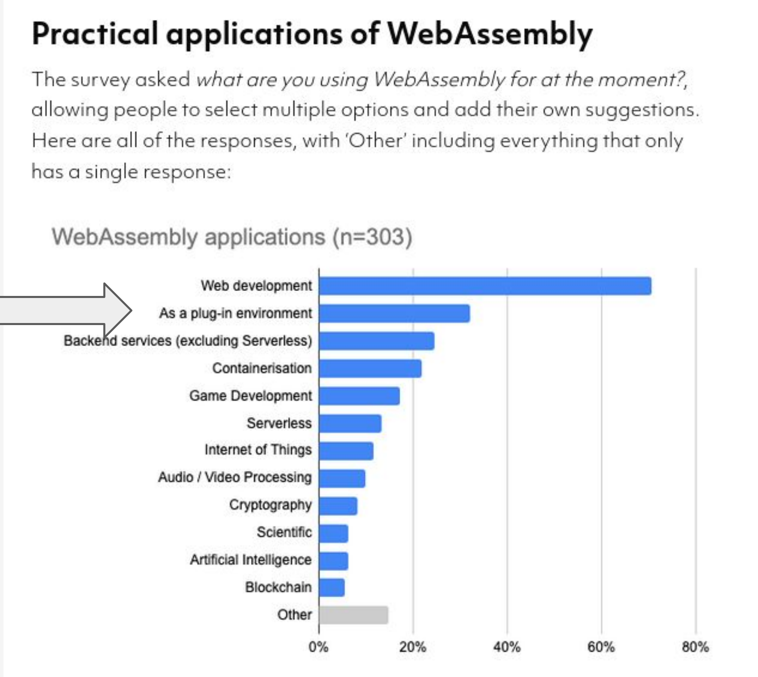
Fast moving target - solutions get obsolete fast

WASI pulling WASM (kinda opposite direction)

Fixes and suggestions welcome! :)



WASM in the cloud: is it just me?



Practical applications of WebAssembly

The survey asked *what are you using WebAssembly for at the moment?*, allowing people to select multiple options and add their own suggestions. Here are all of the responses, with 'Other' including everything that only has a single response:

the state of wasm 2023

WHOAMI?

Software Engineer @ Red Hat

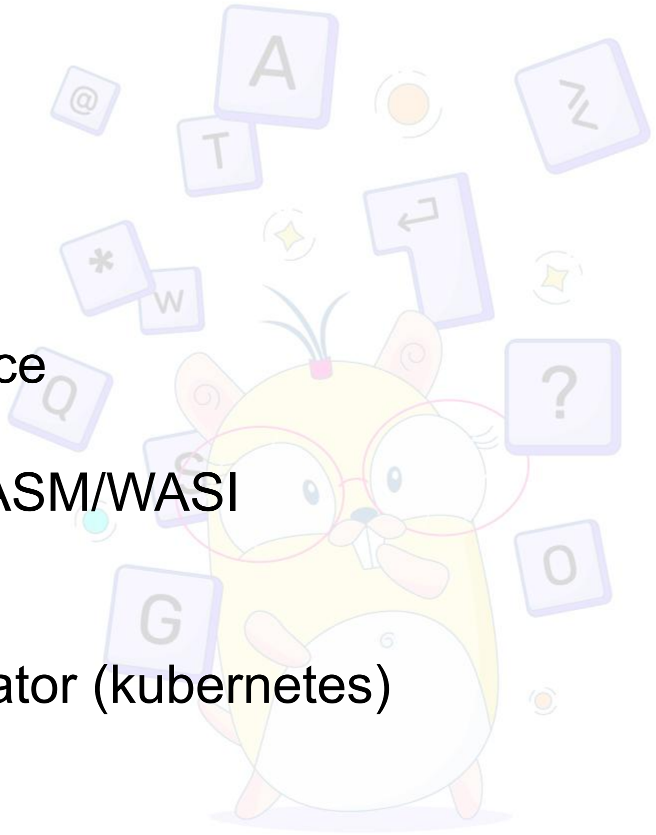
Kubernetes/Openshift contributor

Mostly kubelet/runtime - optimizations for low-latency workloads

Thoughts, opinions and mistakes are mine only!

Outside computing: running, climbing, tabletop gaming

TALK OUTLINE



1. golang, meet WASM/WASI
   a. **W**eb **AS**se**M**bly
   b. **W**eb **A**ssembly **S**ystem **I**nterface

2. extending a golang project with WASM/WASI

3. WASM/WASI in the cloud orchestrator (kubernetes)

Title

golang, meet WASM/WASI

Our journey begins getting to know WASM and WASI, the toolchain, and how to run WASM workloads inside containers and not.

We’re in for a fun start.

▶ TALK OUTLINE

1. **golang, meet WASM/WASI**

2. extending a golang project with WASM/WASI

3. WASM/WASI in the cloud

Toolchain: just set GOOS and GOARCH

```
# new in golang >= 1.21
# see: https://go.dev/blog/wasi

$ GOOS=wasip1 GOARCH=wasm go build -o main.wasm main.go

$ file main.wasm
main.wasm: WebAssembly (wasm) binary module version 0x1
 (MVP)
```

Running WASM files

We need a runtime

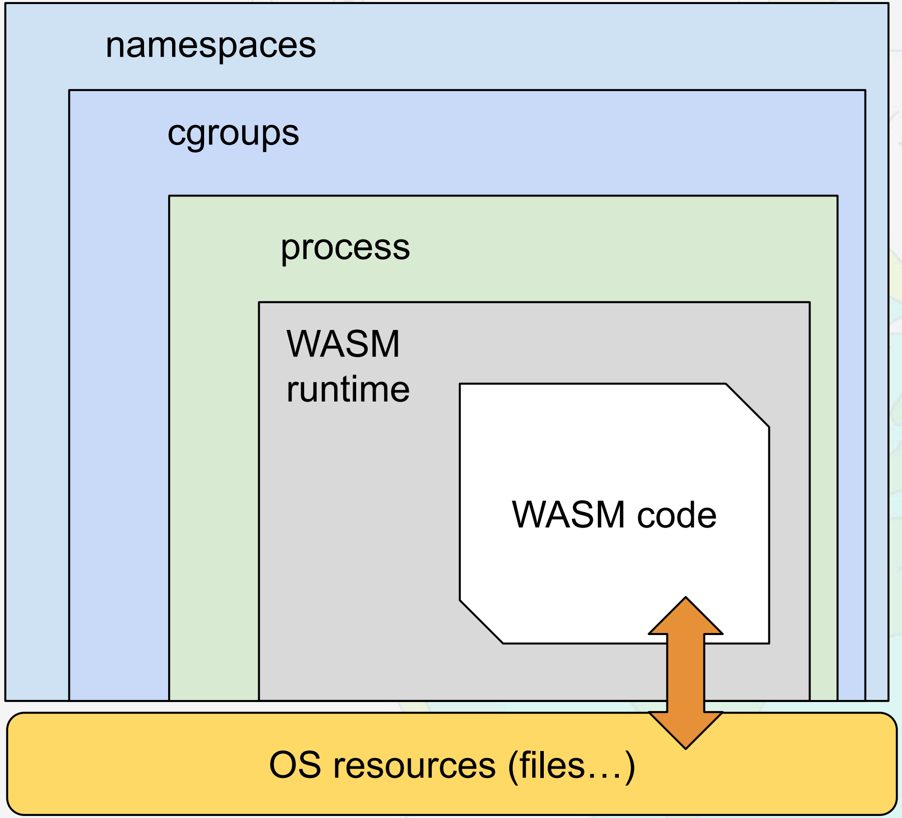
- wasmtime
- wasmer
- **wasmedge** **# !!**
- **wasirun (wazero)** **# !!**
- many others

N67 Cill Íomaí KILLIMER
Gleninagh House
ALL ROOMS EN SUITE
Lios Dúin Bhearna 16 LISDOONVARNA
Fán Óir FANORE 14
MONK'S PUB
ON THE PIER 200 YDS.
Corr Finne 21 COROFIN
An Féar Sorca
Gregans Castle Hotel & Restaurant
Logues Lodge
B&B
AILLWEE CAVE
RUSHEEN LODGE GUESTHOUSE 4 STAR
THE BURREN PERFUMERY & FLORAL CENTRE
P
Cappabhaile House Bed & Breakfast
Aran Islands Doolin Ferries
HEALTH & MEDICAL SHOP
NOW OPEN
opposite the Church BALLYVAUGHAN
NEWTOWN CASTLE & TRAIL

"Far, far too much choice", Trish Steel, CC BY-SA 2.0, via Wikimedia Commons

Runtime == VM

sandboxed environment



Building a WASM container

```
$ podman build \
--annotation "run.oci.handler=wasm" \
--annotation "module.wasm.image/variant=compat" \
-t quay.io/fromani/hello-wasi-go:latest
-f Dockerfile.wasm .
```

Running a WASM container

```
$ podman run --annotation "module.wasm.image/variant=compat"
quay.io/fromani/example-wasi-go:latest

Random number: -963697005
Random bytes: [32 120 210 127 207 206 192 144 247 122 87 155 144 27 204 127 7 53 106 185
134 66 242 46 12 18 42 40 155 113 92 215 24 167 123 158 16 232 216 154 229 211 48 30 139
57 54 2 167 95 104 185 72 40 34 17 37 190 246 246 183 6 116 234 190 247 174 125 241 166
1 252 108 52 169 253 42 55 85 254 230 58 245 60 227 118 83 39 9 163 129 38 66 103 103 14
125 108 164 0 91 79 200 236 99 30 113 145 248 14 240 87 28 178 25 204 112 73 120 214 85
65 38 220 100 118 128 150]
Printed from wasi: This is from a main function
This is from a main function
The env vars are as follows.
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
container=podman
HOME=
HOSTNAME=a1487872fe9a
The args are as follows.
/example-wasi-go.wasm
Working directory is "/"
File content is This is in a file
```

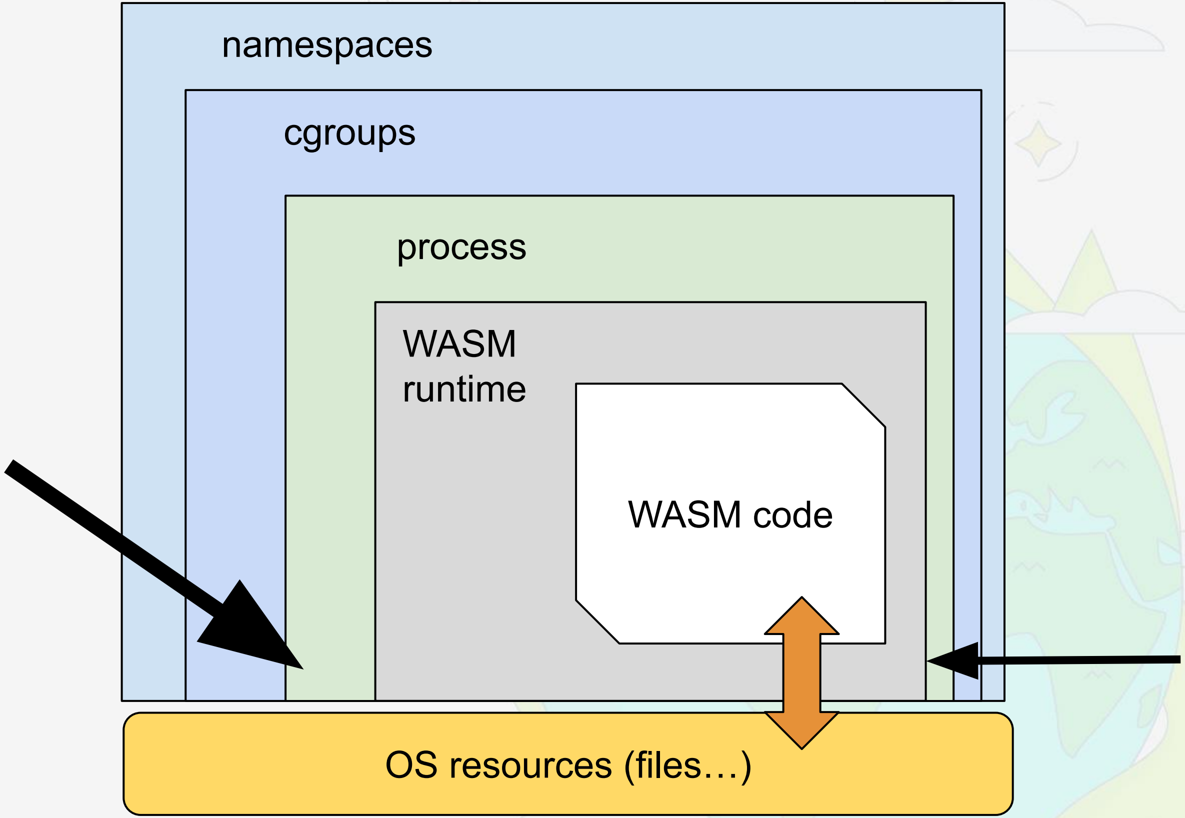
Access control

capabilities-based access control

sandbox needs to be granted access to filesystem

require **some** language support

no changes expected in application code



“C” in “WASI Preview 1” stands for “Complete”

https://go.dev/blog/wasi - limitations

no parallelism (more on this later)

no wasm export (wasm import OK!)

language support (preview1 !)

"Widok Towers, Warsaw under construction", Wistula, CC BY-SA 4.0, via Wikimedia Commons

Title

extending with WASM/WASI

Extending our golang application with WASM/WASI plugins is an appealing prospect. Let’s see what we can do.

A good story is never without challenges.

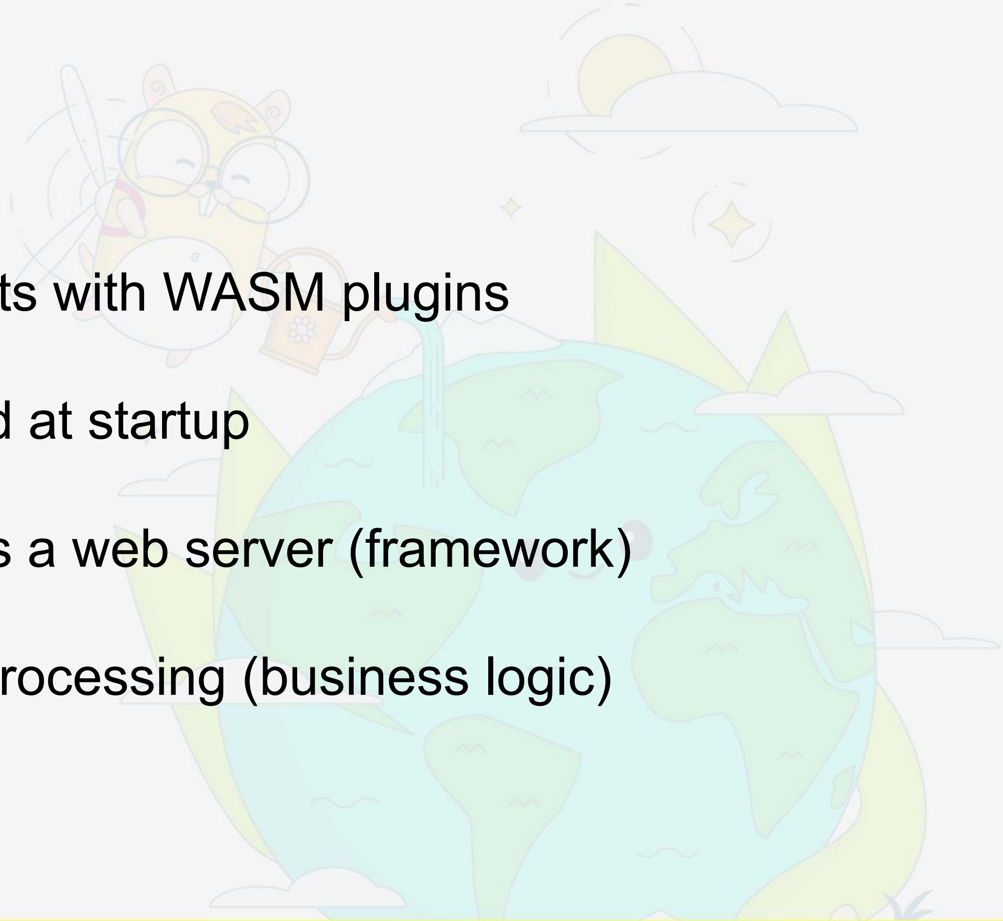
TALK OUTLINE

1. golang, meet WASM/WASI

2. **extending a golang project with WASM/WASI**

3. WASM/WASI in the cloud

The example application



Processing HTTP requests with WASM plugins

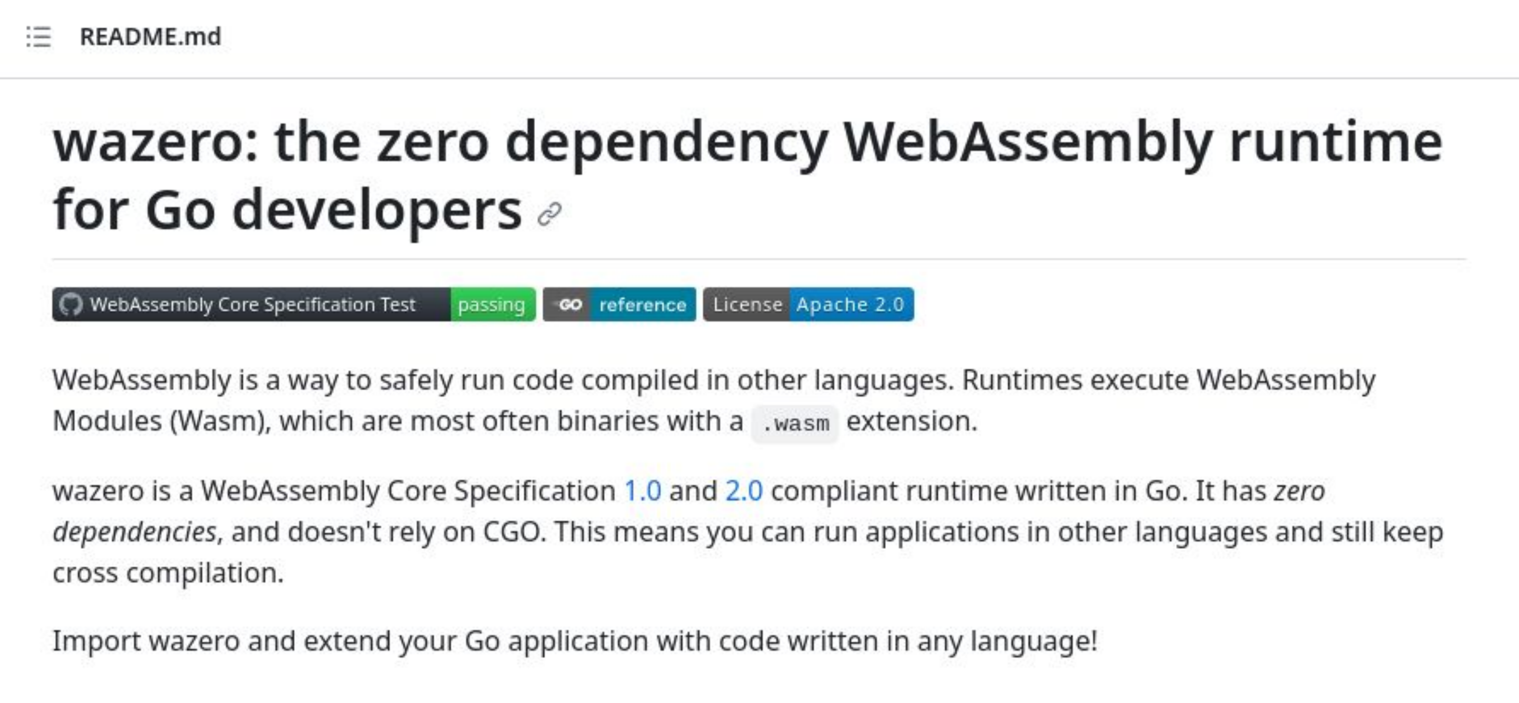
WASM plugins configured at startup

Golang host application is a web server (framework)

WASM plugin do all the processing (business logic)

extending with WASM/WASI

GO wasm runtime



README.md

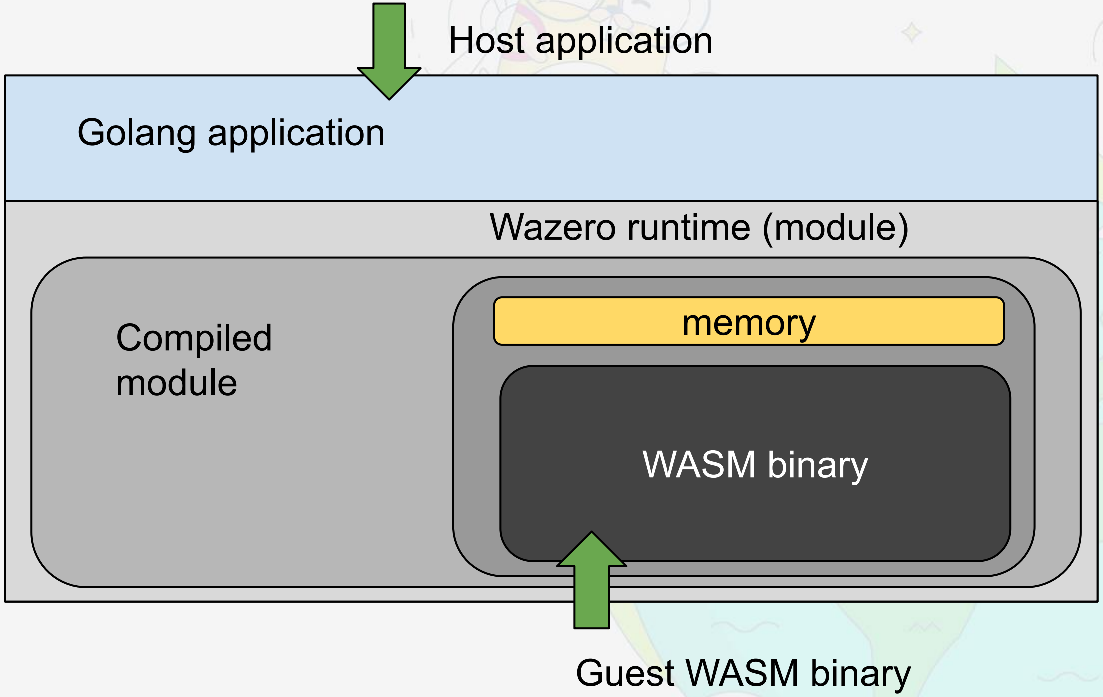
wazero: the zero dependency WebAssembly runtime for Go developers

WebAssembly is a way to safely run code compiled in other languages. Runtimes execute WebAssembly Modules (Wasm), which are most often binaries with a `.wasm` extension.

wazero is a WebAssembly Core Specification 1.0 and 2.0 compliant runtime written in Go. It has *zero dependencies*, and doesn't rely on CGO. This means you can run applications in other languages and still keep cross compilation.

Import wazero and extend your Go application with code written in any language!

Host vs Guest



Running WASM files

```
func (wh *wasmHandler) ServeHTTP(
    w http.ResponseWriter,
    r *http.Request
) {
    // create runtime

    // load module

    // instantiate module - run entry point

    // process request - process by side effect
}
```

Running WASM files: create runtime

```
ctx := context.Background()

rt := wazero.NewRuntime(ctx)
defer rt.Close(ctx)

wasi_snapshot_preview1.MustInstantiate(ctx, rt)
```

Running WASM files: load module

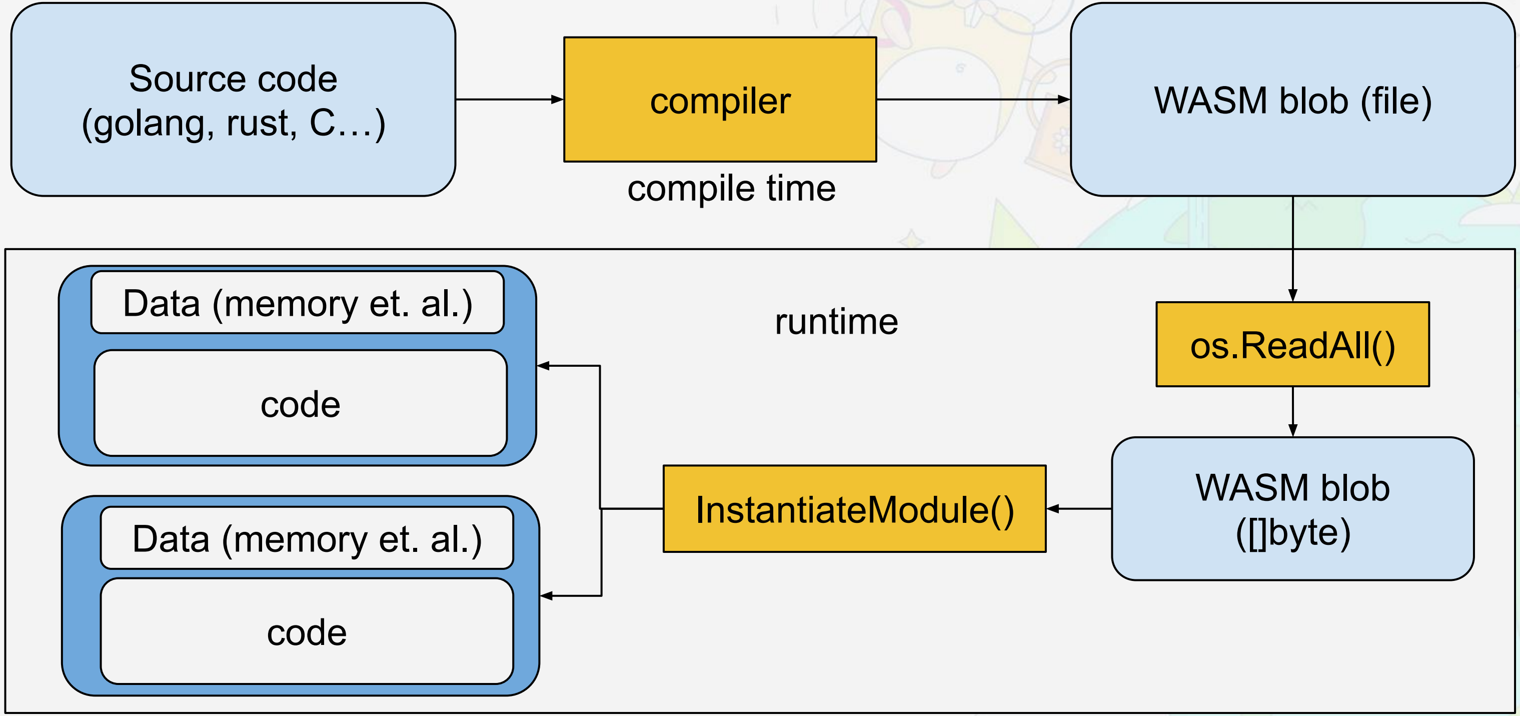
```
// a smarter io.ReadAll
wasmObj, err := wh.loadModule(wh.moduleName)

var stdout bytes.Buffer
cfg := wazero.NewModuleConfig()
    .WithName(wh.moduleName)
    .WithStdout(&stdout)
    .WithStderr(os.Stderr)
```

Running WASM files: instantiate module

```
mod, err := rt.InstantiateWithConfig(ctx, wasmObj, cfg)
if err != nil {
    ...
}

mod.Close(ctx)
```

Running WASM files: instantiate module (take 2)

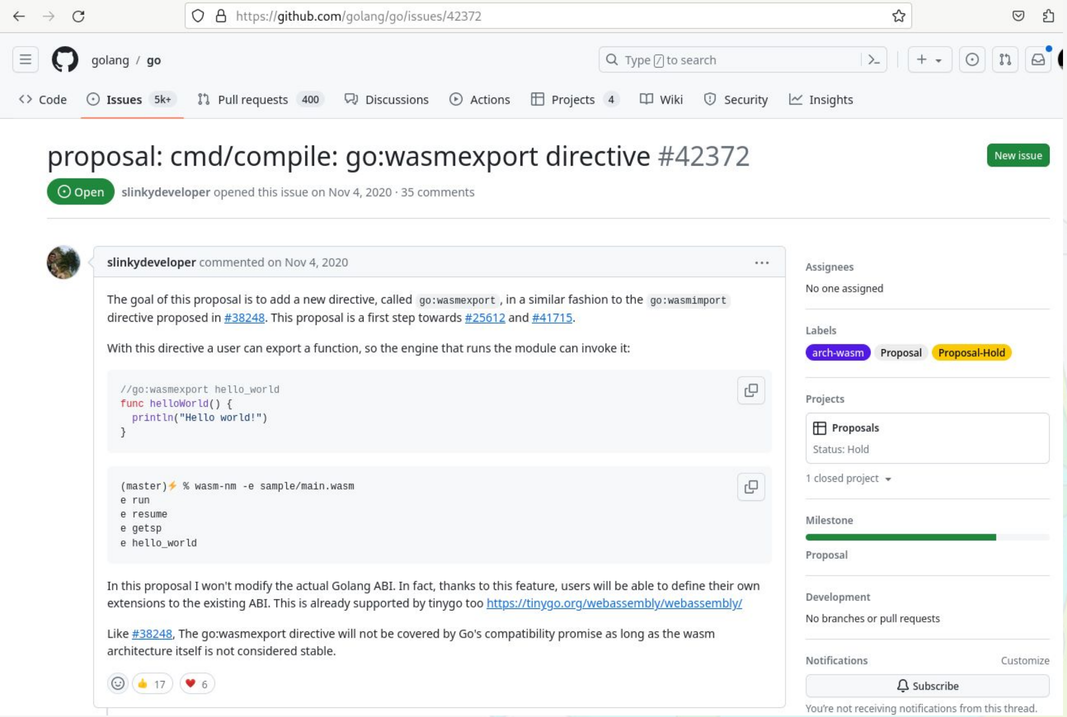
for more details: wazero docs



Exposing functions

process explicitly!

From the other side of the screen it looks so easy



proposal: cmd/compile: go:wasmexport directive #42372

Open slinkydeveloper opened this issue on Nov 4, 2020 · 35 comments

slinkydeveloper commented on Nov 4, 2020

The goal of this proposal is to add a new directive, called `go:wasmexport`, in a similar fashion to the `go:wasmimport` directive proposed in #38248. This proposal is a first step towards #25612 and #41715.

With this directive a user can export a function, so the engine that runs the module can invoke it:

```
//go:wasmexport hello_world
func helloWorld() {
  println("Hello world!")
}
```

```
(master)⚡ % wasm-nm -e sample/main.wasm
e run
e resume
e getsp
e hello_world
```

In this proposal I won't modify the actual Golang ABI. In fact, thanks to this feature, users will be able to define their own extensions to the existing ABI. This is already supported by tinygo too https://tinygo.org/webassembly/webassembly/

Like #38248, The go:wasmexport directive will not be covered by Go's compatibility promise as long as the wasm architecture itself is not considered stable.

👍 17 ❤️ 6

Assignees
No one assigned

Labels
arch-wasm Proposal Proposal-Hold

Projects
Proposals
Status: Hold
1 closed project

Milestone
Proposal

Development
No branches or pull requests

Notifications Customize
Subscribe
You're not receiving notifications from this thread.

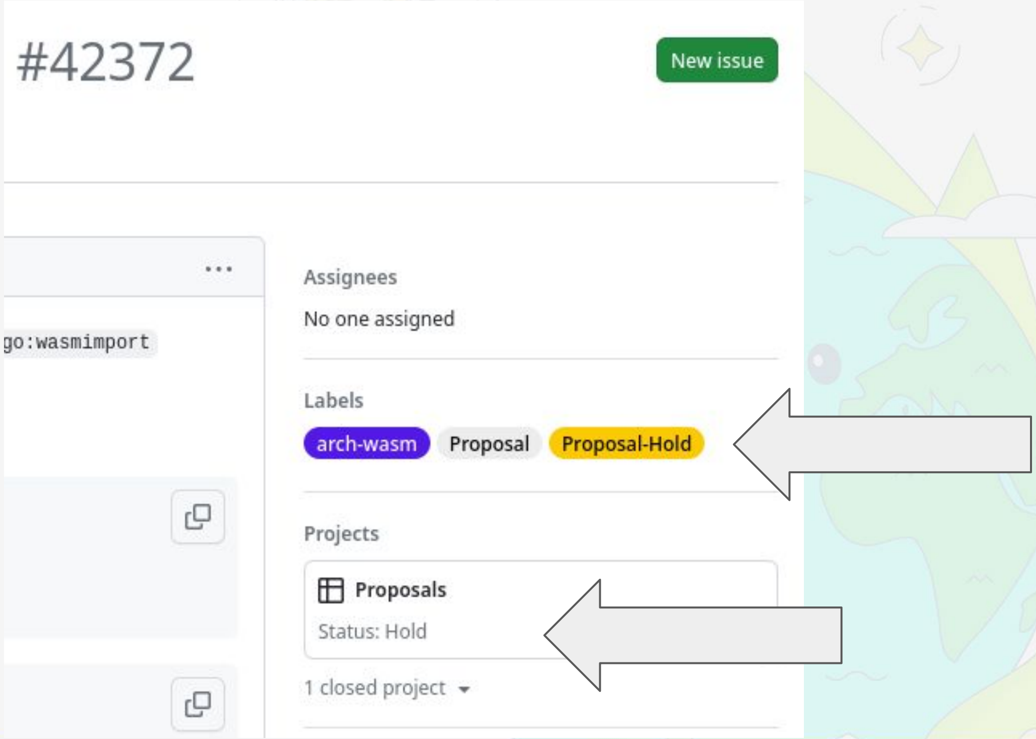
Exposing functions

golang 1.21 (fetched: 20231022)

Pass data

Retrieve data

(multi-value?)



#42372

Assignees

No one assigned

Labels

arch-wasm Proposal Proposal-Hold

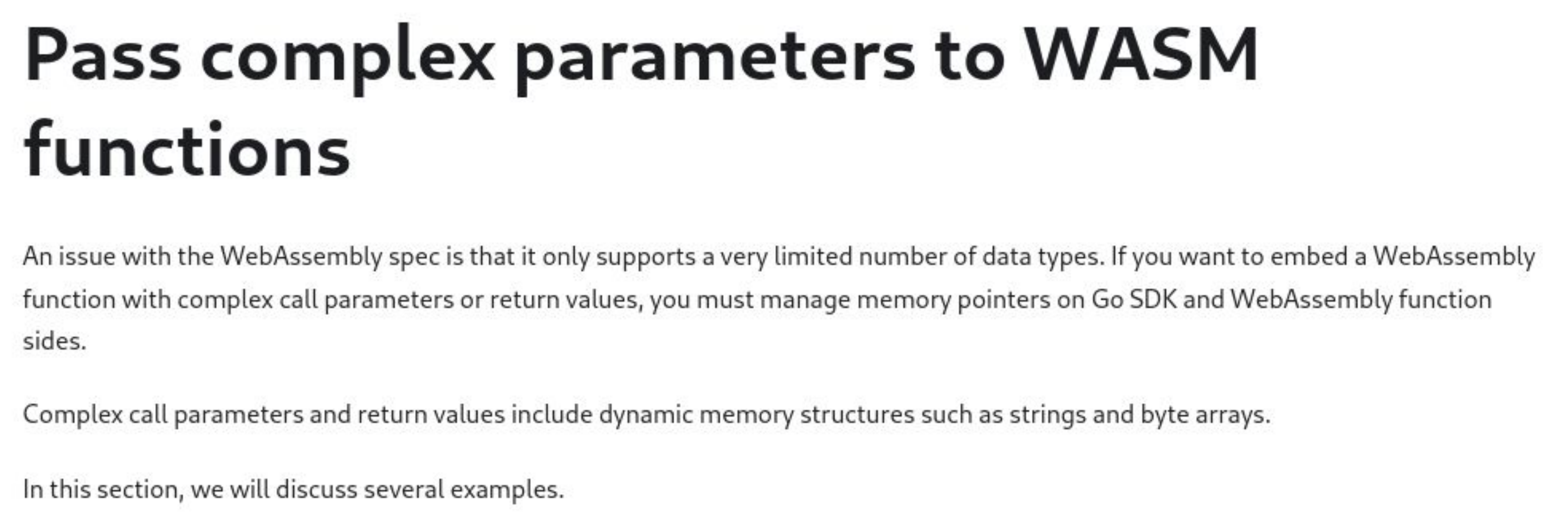
Projects

Proposals

Status: Hold

1 closed project

Complex parameters, complex solutions



Pass complex parameters to WASM functions

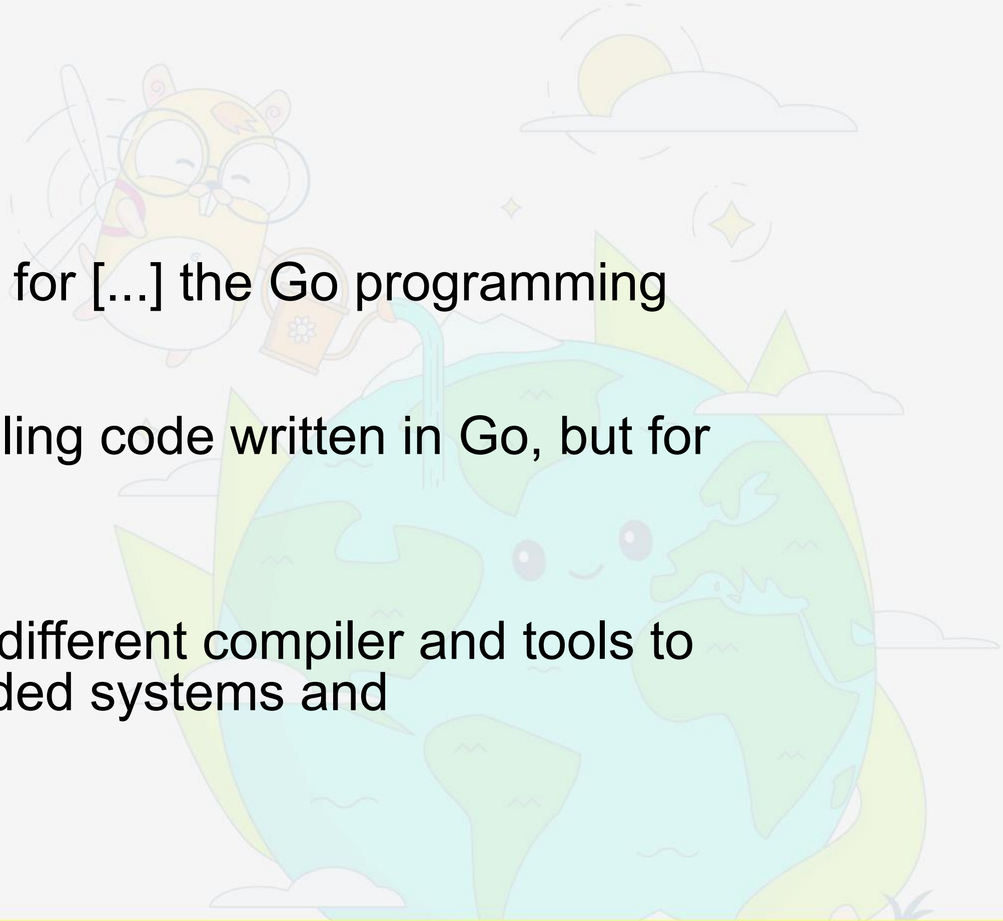
An issue with the WebAssembly spec is that it only supports a very limited number of data types. If you want to embed a WebAssembly function with complex call parameters or return values, you must manage memory pointers on Go SDK and WebAssembly function sides.

Complex call parameters and return values include dynamic memory structures such as strings and byte arrays.

In this section, we will discuss several examples.

source

Meet TinyGO



TinyGo is a new compiler for [...] the Go programming language.

TinyGo focuses on compiling code written in Go, but for smaller kinds of systems:

[...]
However, TinyGo uses a different compiler and tools to make it suited for embedded systems and WebAssembly.

Caveats

https://tinygo.org/docs/reference/lang-support/

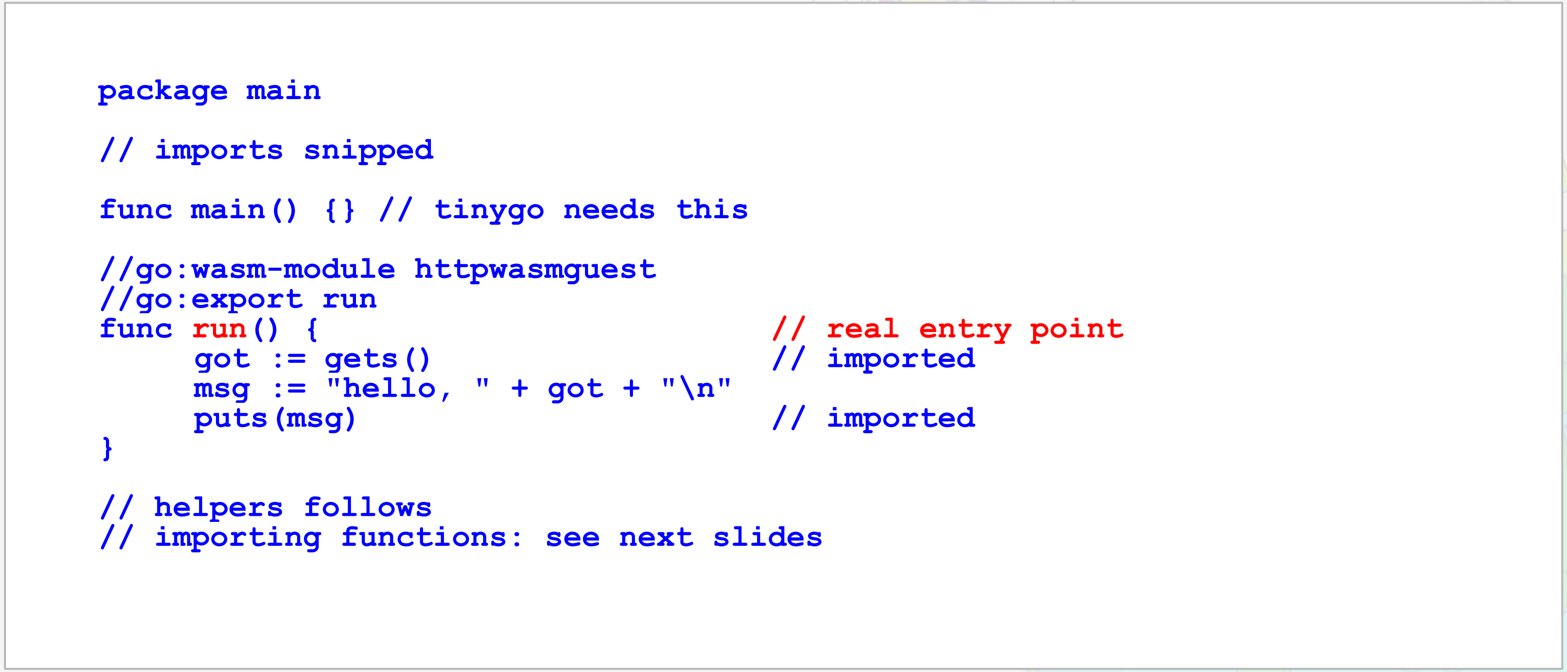
- limited reflection support (thus)
- limited stdlib

tinygo + WASM caveats

- no parallelism (yet)
- GC tuning required

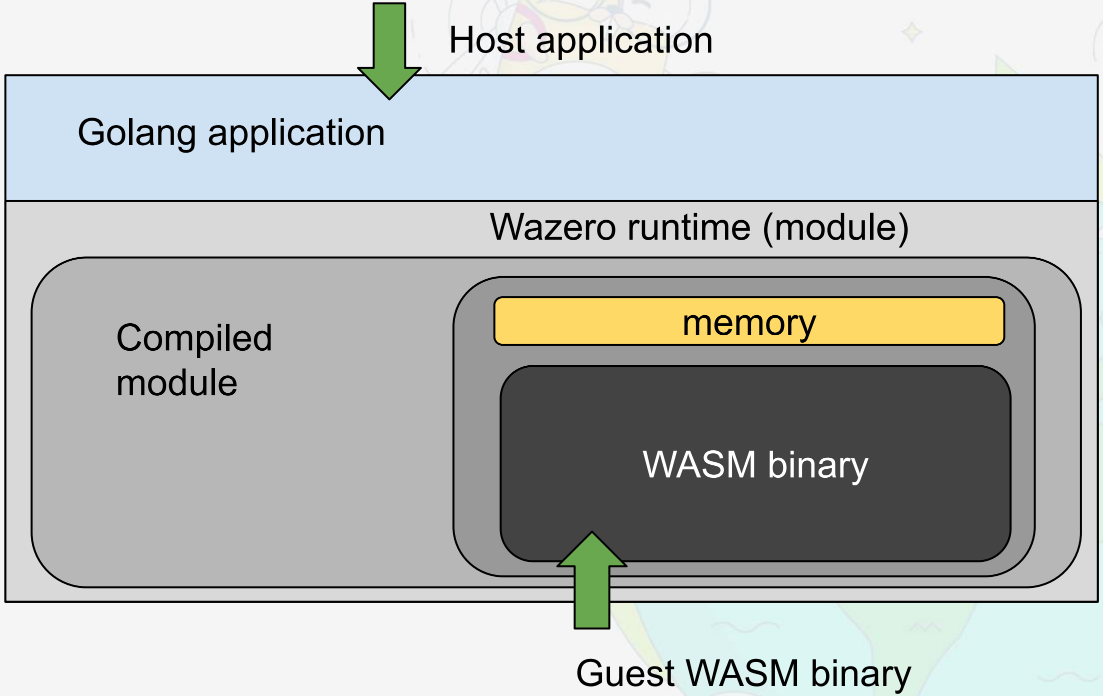
extending with WASM/WASI

(TinyGO) guest module layout



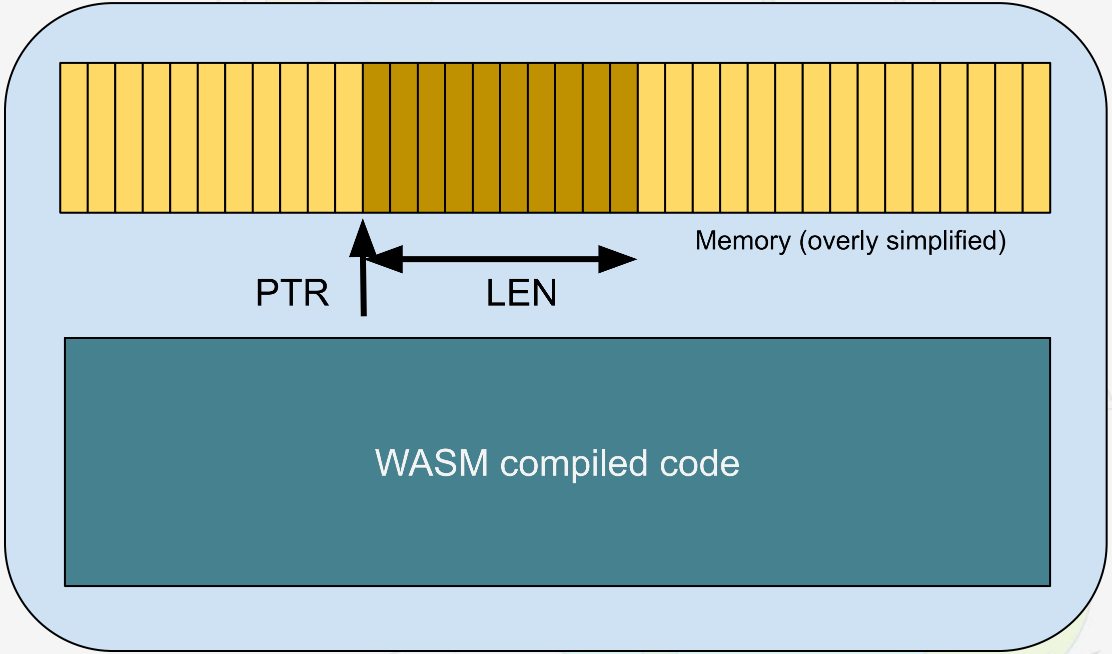
```
package main

// imports snipped

func main() {} // tinygo needs this

//go:wasm-module httpwasmguest
//go:export run
func run() {                          // real entry point
     got := gets()                    // imported
     msg := "hello, " + got + "\n"
     puts(msg)                        // imported
}

// helpers follows
// importing functions: see next slides
```

Host vs Guest



WASM memory layout overview

- address space: 32 bits
- pages of 64 KiB
- guest memory managed automatically
- host/guest interaction



rolling our own I/O: output

```
//go:wasmimport httpwasm oputs
func putStringStdout(bufPtr, bufLen uint32)

func puts(s string) {
    ptr, size := stringToPtr(s) // determine ptr + len
    putStringStdout(ptr, size)  // call host function
    runtime.KeepAlive(s)        // need this
}

func stringToPtr(s string) (uint32, uint32) {
    ptr := unsafe.Pointer(unsafe.StringData(s))
    return uint32(uintptr(ptr)), uint32(len(s))
}
```

rolling our own I/O: input

```go
//go:wasmimport httpwasm igets
func getStringStdin() uint64

func gets() string {
    ret := getStringStdin()     // call host function
    ptr := uint32(ret >> 32)    // decode ptr..
    size := uint32(ret)         // ...and len
    data := ptrToBytes(ptr, size) // convert to []byte
    return string(data)         // usable at last
}
```

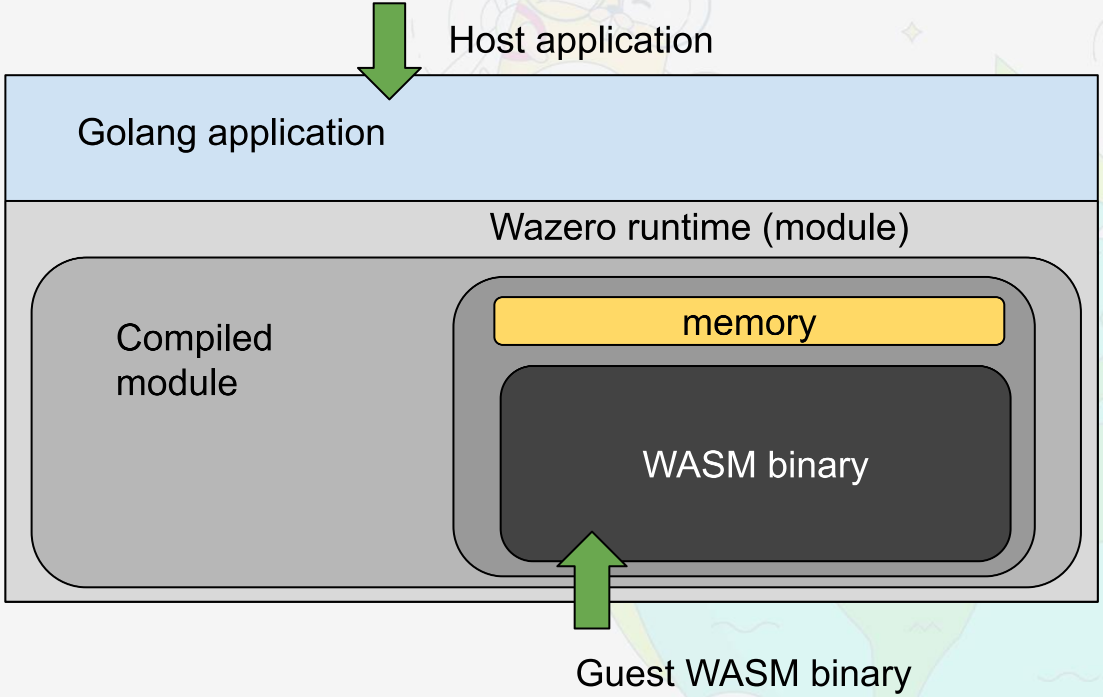
rolling our own I/O: input pt 2

```
func ptrToBytes(ptr, size uint32) []byte {
    var b []byte
    s := (*reflect.SliceHeader)(unsafe.Pointer(&b))
    s.Len = uintptr(size)
    s.Cap = uintptr(size)
    s.Data = uintptr(ptr)
    return b
}

// TL;DR: unsafe-ly create a []byte from ptr, size
```

The host side

The Rose and Crown pub by JThomas, CC BY-SA 2.0

Host vs Guest



new references needed

```
const moduleEntryPointName = "run"


type wasmEngine struct {
      code     wazero.CompiledModule
      rt       wazero.Runtime
      hostMod  api.Module // closed when we close the runtime
      guestMod api.Module // closed when we close the runtime
      stack    []uint64
      mallocFn api.Function
      freeFn   api.Function
      runFn    api.Function
}

// need to store module function references (post-instantiate)
```

extending with WASM/WASI

register host functions

```
hostMod, err := rt.NewHostModuleBuilder("httpwasm").
    NewFunctionBuilder().WithFunc(igets).Export("igets").
    NewFunctionBuilder().WithFunc(eputs).Export("eputs").
    NewFunctionBuilder().WithFunc(oputs).Export("oputs").
        Instantiate(ctx)

// will peek at the code later
```

instantiate and lookup guest functions

```
cfg := wazero.NewModuleConfig().WithName("httpwasmguest")
// also invokes the _start function, empty now

guestMod, err := rt.InstantiateModule(ctx, code, cfg)

runFn := guestMod.ExportedFunction(moduleEntryPointName)
// omitted: check if == nil - error

// rinse and repeat for malloc and free
```

before to run: callData

```
// pass data through context to host functions

type callData struct {
	stdin    bytes.Buffer
	stdout   bytes.Buffer
	stderr   bytes.Buffer
	mallocFn api.Function
	freeFn   api.Function
	allocs   []uint32
}

type callDataKey struct{}
```

before to run: callData /2

```
func putCallData(ctx context.Context, we *wasmEngine)
(context.Context, *callData) {
    cdata := callData{
        mallocFn: we.mallocFn,
        freeFn:   we.freeFn,
    }
    ctx = context.WithValue(ctx, callDataKey{}, &cdata)
    return ctx, &cdata
}


func getCallData(ctx context.Context) *callData {
    return ctx.Value(callDataKey{}).(*callData)
}
```

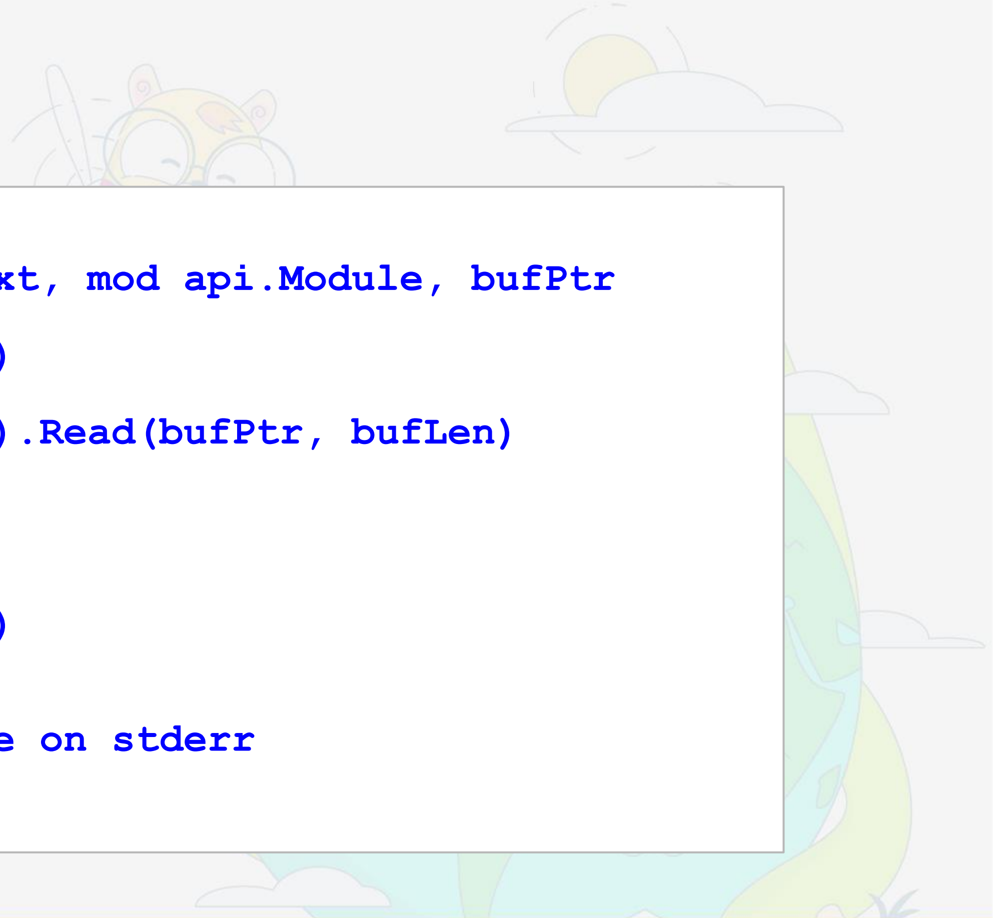
let it run

```
func (we *wasmEngine) Run(ctx context.Context, name string, stdin
io.Reader, env map[string]string) (string, string, error) {

     guestCtx, cdata := putCallData(ctx, we)

     stdinData, err := io.ReadAll(stdin)

     stdinData = append(stdinData, byte('\n'))

     _, err = cdata.stdin.Write(stdinData)

     err = we.runFn.CallWithStack(guestCtx, we.stack) // reuse stack

     dealloc(cdata) // manual memory management, keepalive

     return cdata.stdout.String(), cdata.stderr.String(), err
}
```

guest module refresher

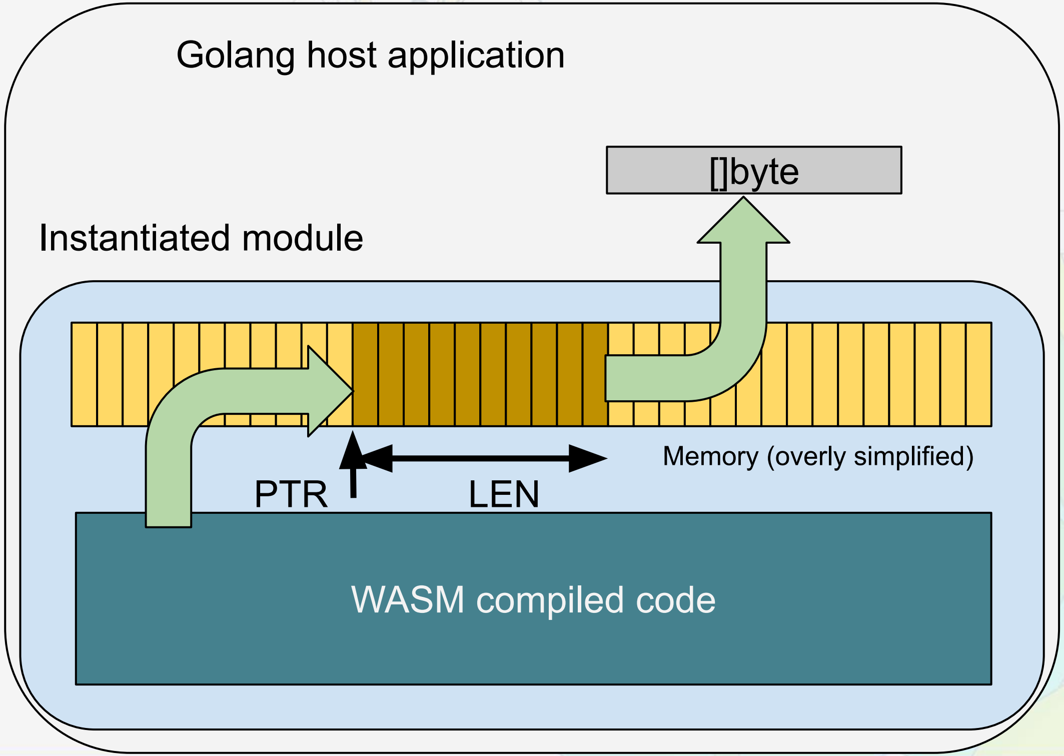
```
func run() {
    got := gets()
    msg := "hello, " + got + "\n"
    puts(msg)
}
```

host output is easy (is it?)



```
func oputs(ctx context.Context, mod api.Module, bufPtr
uint32, bufLen uint32) {
    cdata := getCallData(ctx)

    bytes, ok := mod.Memory().Read(bufPtr, bufLen)
    if !ok {
        // TODO
    }

    cdata.stdout.Write(bytes)
}

// eputs is (almost) the same on stderr
```

how does it fit together?

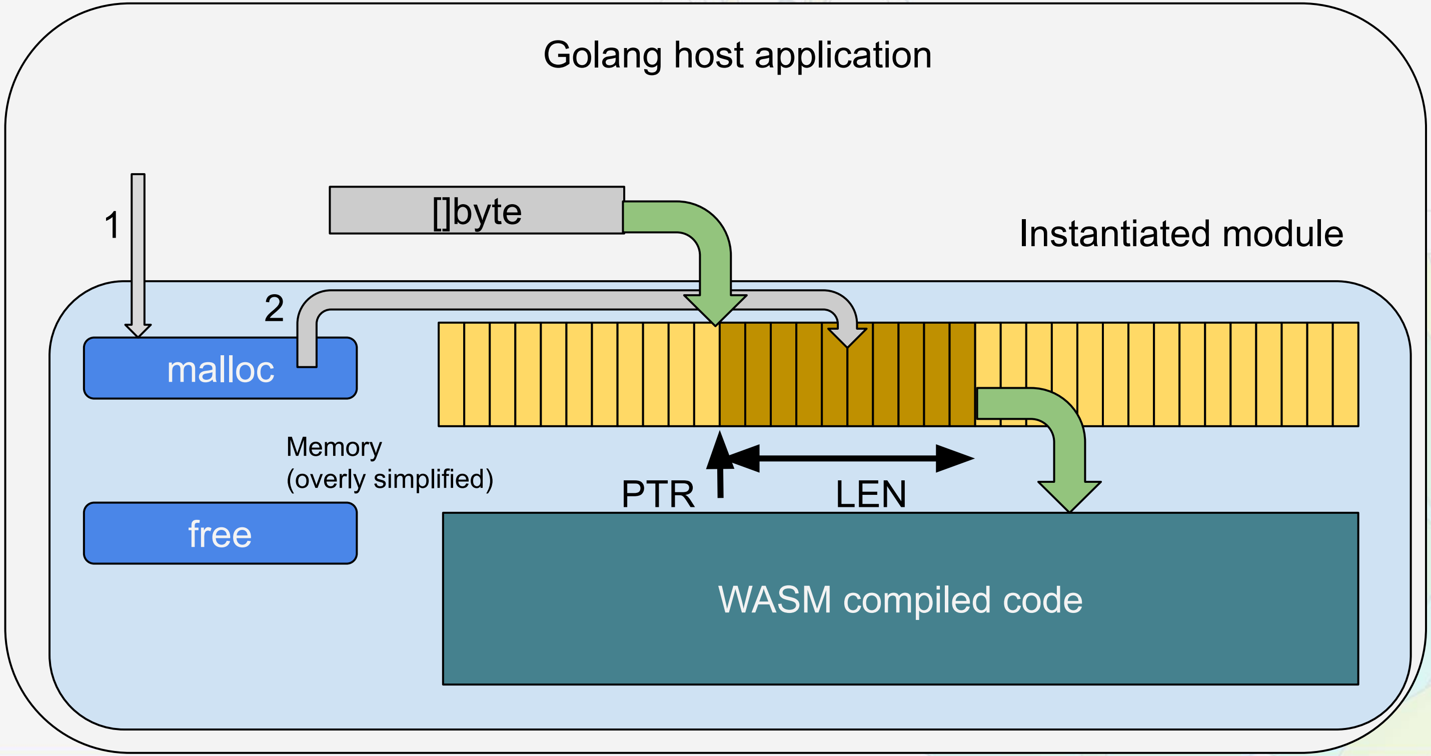


extending with WASM/WASI

host input, and the joys of manual memory management

```
func igets(ctx context.Context, mod api.Module) uint64 {
	cdata := getCallData(ctx)

	stdinData, err := cdata.stdin.ReadBytes('\n')

	// when to free() ?
	results, err := cdata.mallocFn.Call(ctx, uint64(len(stdinData)))

	ptr := results[0]
	size := uint64(len(stdinData))

	mod.Memory().Write(uint32(ptr), stdinData)

	return (uint64(ptr) << uint64(32)) | uint64(size) // encode block
}
```

how does it fit together?



extending with WASM/WASI

when to free? and how?

```
func dealloc(cdata *callData) error {
    ctx := context.TODO()
    for len(cdata.allocs) > 0 {
        ptr := cdata.allocs[0]
        cdata.allocs = cdata.allocs[1:]

         , err := cdata.freeFn.Call(ctx, uint64(ptr))
        if err != nil {
            return err
        }
    }
    return nil
}
```

avoiding self inflicted pain

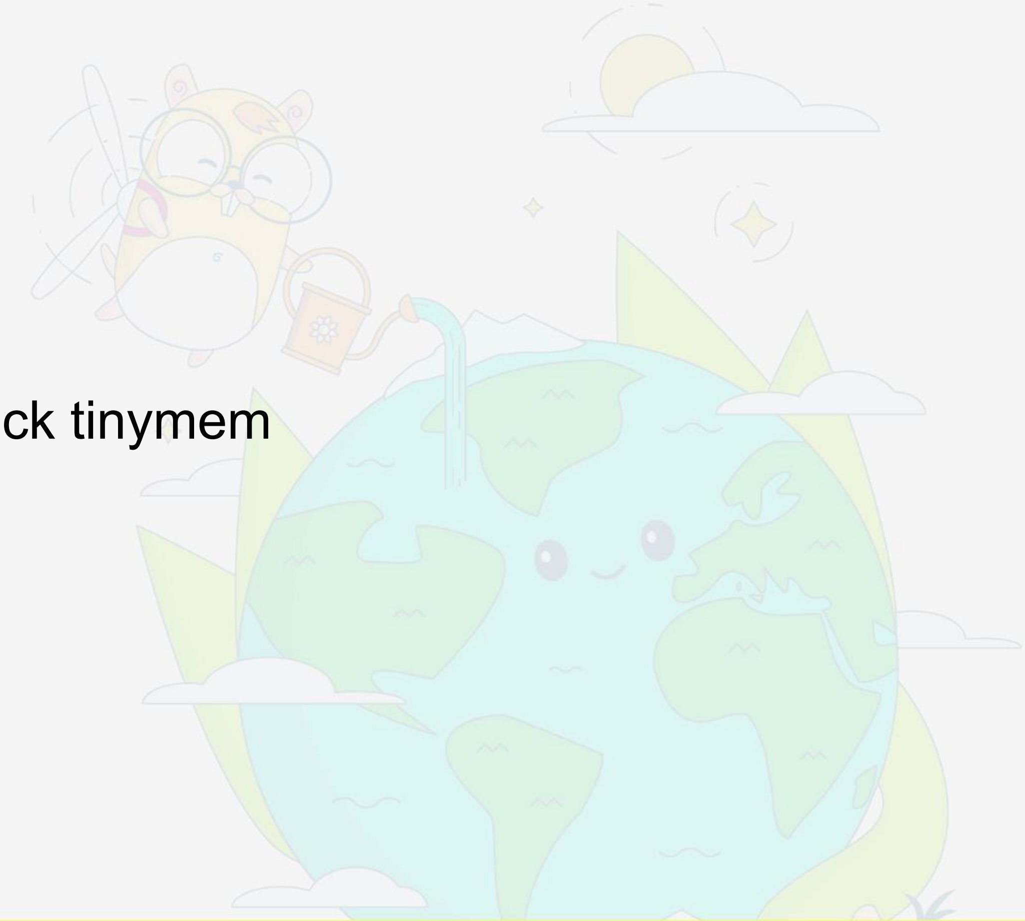
```
// out gets turns out to be a pretty weird case
// so let's just avoid it?

// host uses malloc to prepare the message,
// so it becomes trivial to free. And then:

func run(ptr uintpr, len uint32) {
    got := tinymem.PtrToString(ptr, len)
    msg := "hello, " + got + "\n"
    puts(msg)
}
```

wrapping up

tinygo for the guest side

pass complex data - check tinymem

review API



manual memory management essentials

READ THIS FIRST! https://wazero.io/languages/tinygo/

memory ownership

https://github.com/tetratelabs/tinymem

Title

WASM/WASI in the clouds

We’re not alone trying out WASM. Cloud infra projects are actively planning or experimenting about extending their components with WASM/WASI. Let’s see what and why

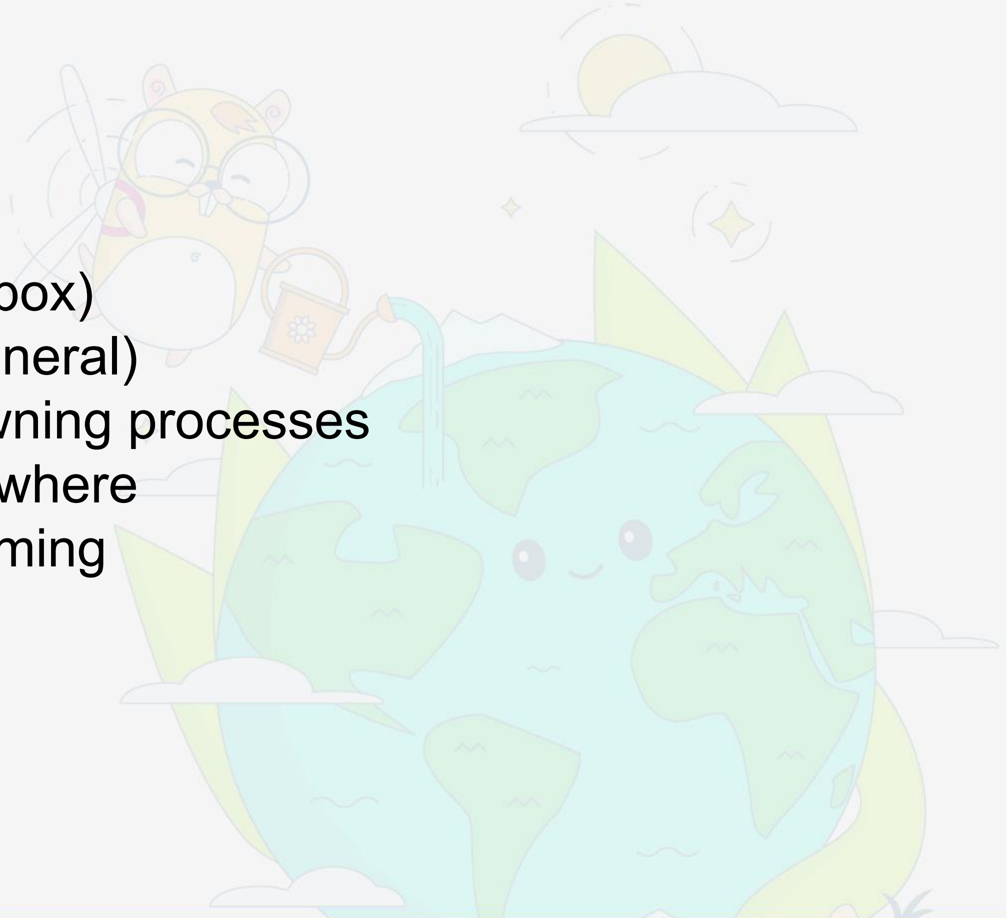
When the least expected, cloud connected!

▶ TALK OUTLINE

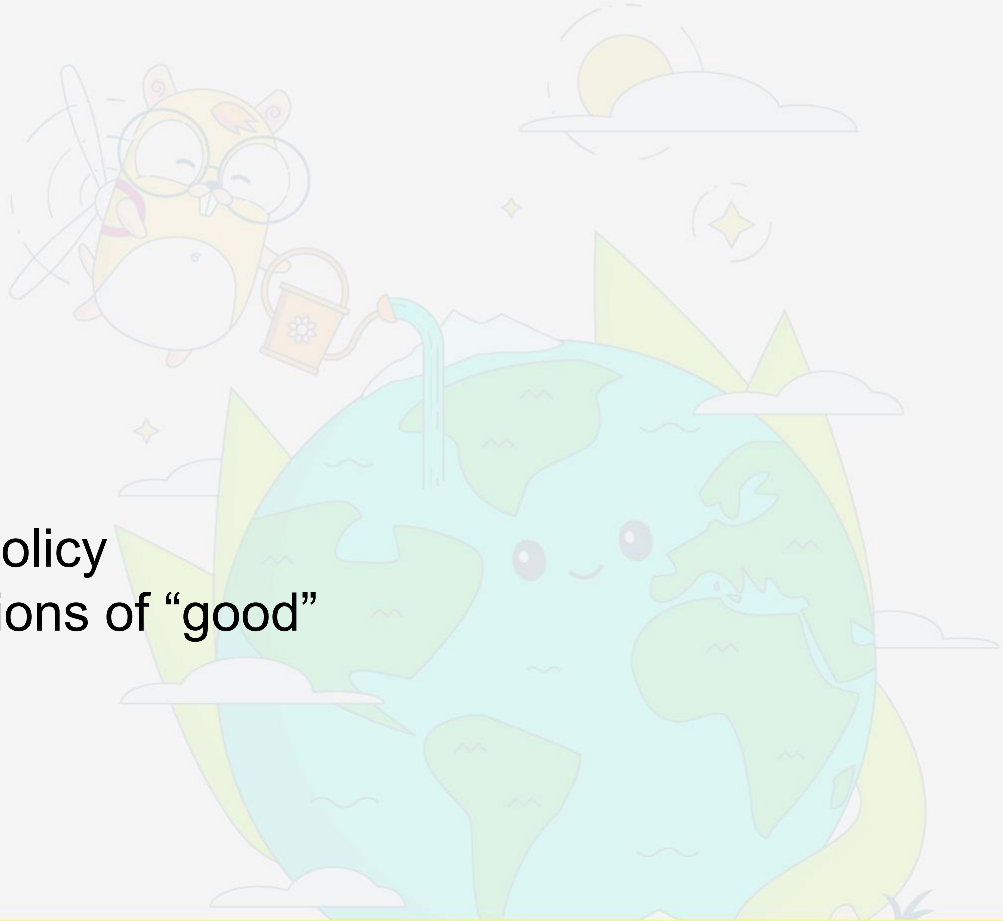
1. golang, meet WASM/WASI
2. extending a golang project with WASM/WASI
3. **WASM/WASI in the cloud**

WASI in the cloud: selling points

- safe execution (sandbox)
- fast (enough, or in general)
  - benchmark: spawning processes
- write once, run everywhere
  - polyglot programming



the kubernetes scheduler



assign pods to nodes

resource allocation

high demand of custom policy

- wildly different definitions of “good”

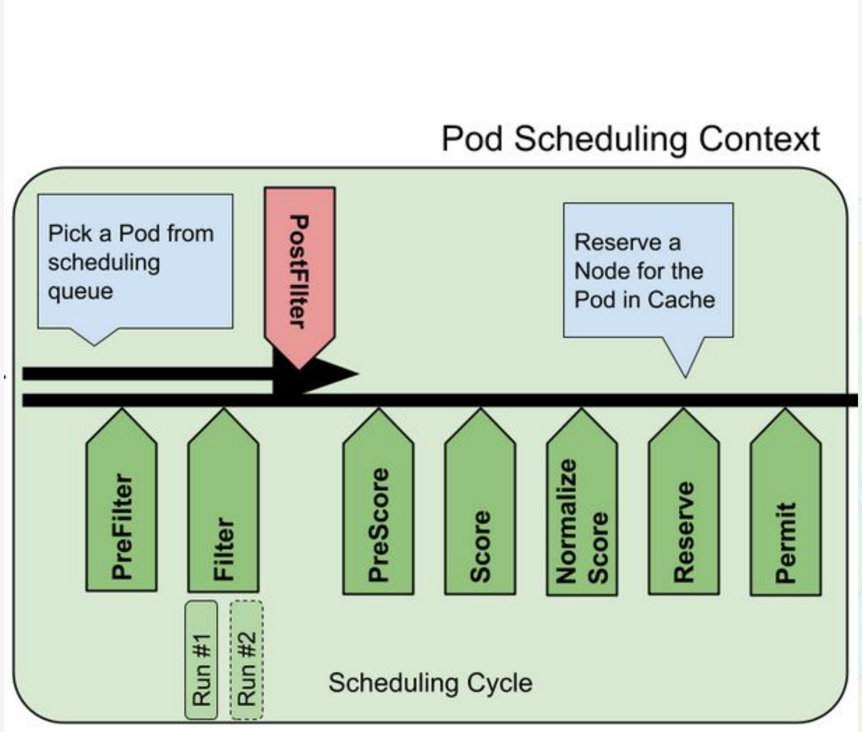
the kubernetes scheduler extensions

scheduler extender

- webhook

scheduler framework

- rich set of extension points
- “builtin” plugins



WASM in kubernetes scheduler: sharing state (explained)

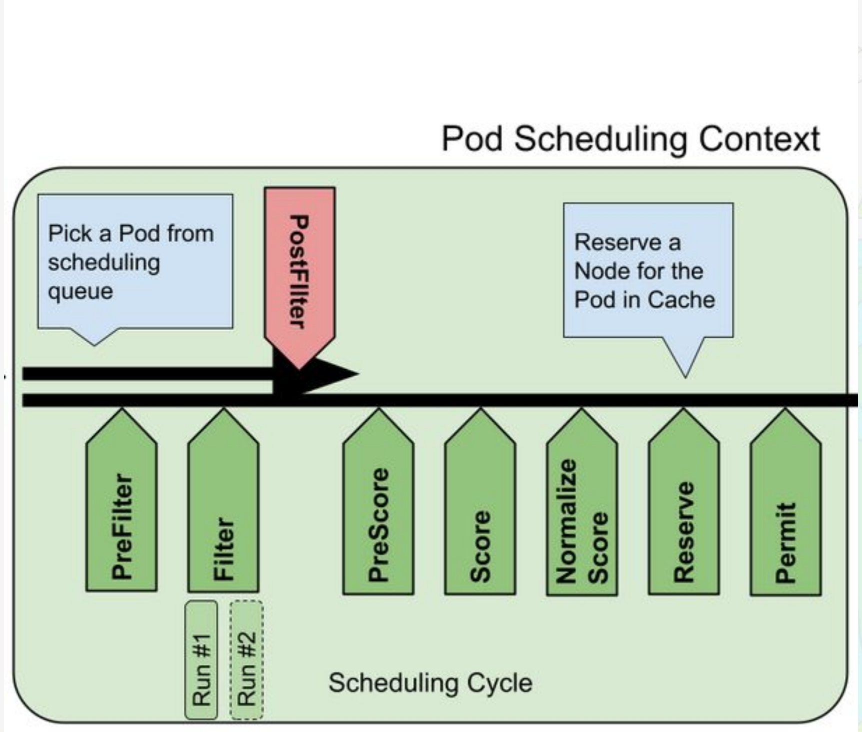
the kubernetes scheduler provides CycleState to share state across a scheduling cycle

in some cases checking a condition requires computing which can be reused later (**overly simplified example**):

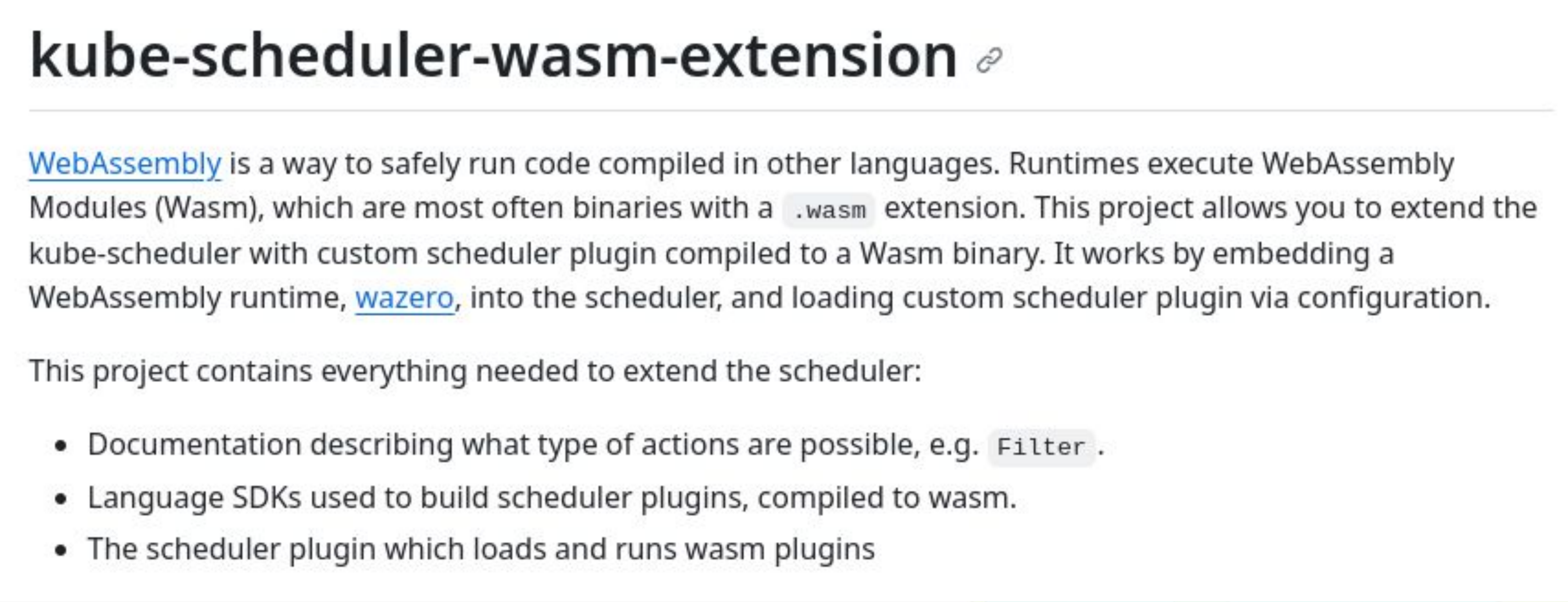
PreFilter: do the affinity rules make sense?
vs
Filter: do the affinity rules match any node?

WASM in kubernetes: sharing state (refresh)

kubernetes scheduler plugins WANT to share state across extension points



kubernetes (scheduler) meet WASI



kube-scheduler-wasm-extension

WebAssembly is a way to safely run code compiled in other languages. Runtimes execute WebAssembly Modules (Wasm), which are most often binaries with a `.wasm` extension. This project allows you to extend the kube-scheduler with custom scheduler plugin compiled to a Wasm binary. It works by embedding a WebAssembly runtime, wazero, into the scheduler, and loading custom scheduler plugin via configuration.

This project contains everything needed to extend the scheduler:

- Documentation describing what type of actions are possible, e.g. `Filter`.
- Language SDKs used to build scheduler plugins, compiled to wasm.
- The scheduler plugin which loads and runs wasm plugins

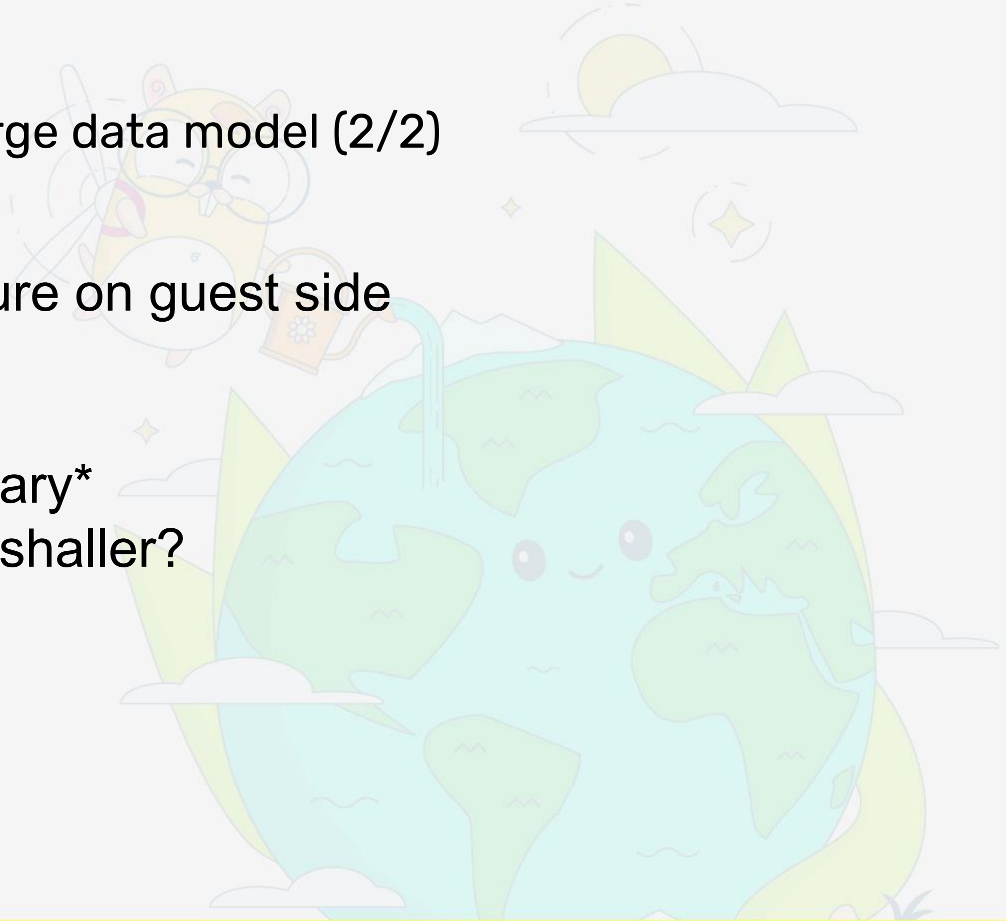
WASM in kubernetes scheduler: large data model (1/2)

kubernetes objects (nodes, pods) are big and nested

how to pass the sandbox boundary?

- per-field accessors
  - gets out of hand too easily
- host marshals the full object, pass as blob, guest unmarshals
  - path of least resistance
  - lots of garbage on guest side

WASM in kubernetes scheduler: large data model (2/2)



mitigating memory pressure on guest side

- lazy decoding
- update only if necessary*
- just use better unmarshaller?

changing APIs to reduce friction

```
func NewStatus(code Code, reasons ...string) *Status {
  s := &Status{
    code:    code,
    reasons: reasons,
  }
  if code == Error {
    s.err = errors.New(s.Message())
  }
  return s
}
```

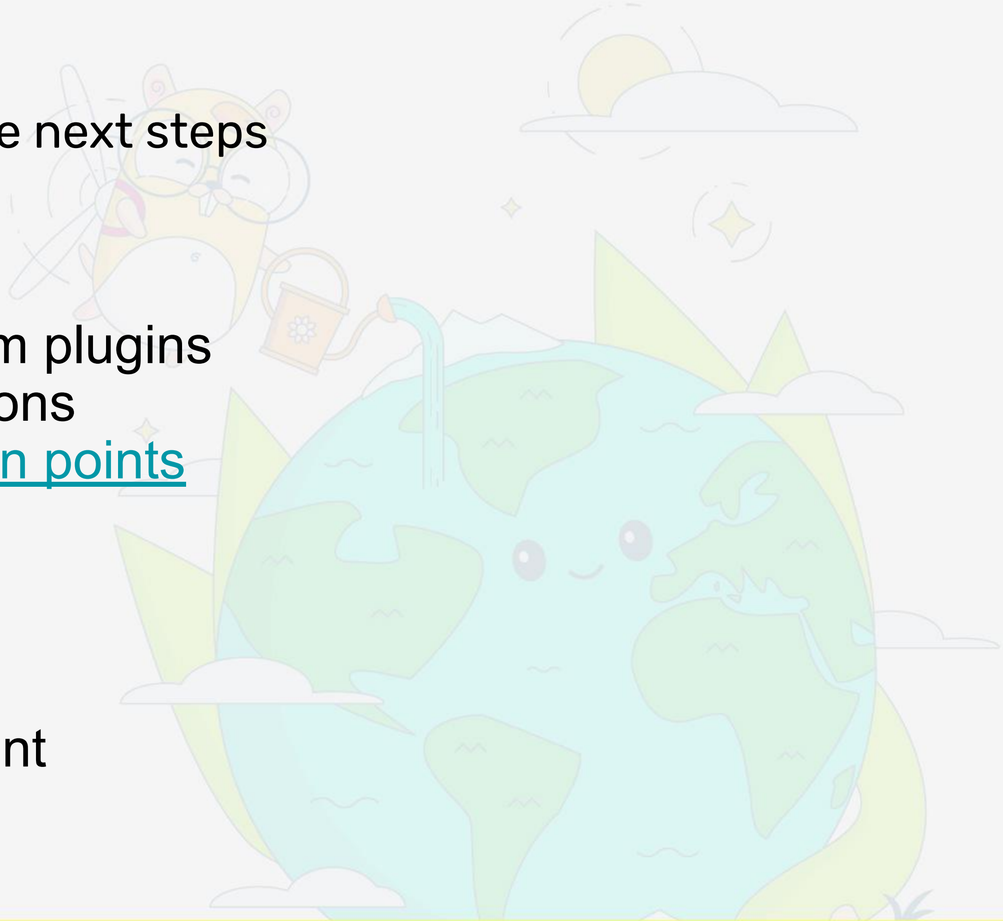
changing APIs to reduce friction (/2)

```
func (tm *TopologyMatch) Filter(...) *framework.Status {
    if nodeInfo.Node() == nil {
        return framework.NewStatus(
            framework.Error,
            "node not found",
         )
    }
    // ...
}
```

changing APIs to reduce friction (/3)

```
func StatusToCode(s *api.Status) uint32 {
    if s == nil || s.Code == api.StatusCodeSuccess {
     return uint32(api.StatusCodeSuccess)
    }

    if reason := s.Reason; reason != "" {
         setStatusReason(reason)
    }
    return uint32(s.Code)
}
```

WASM in kubernetes scheduler: the next steps



Some highlights

- support multiple wasm plugins
- better event notifications
- cover all the extension points

open points:

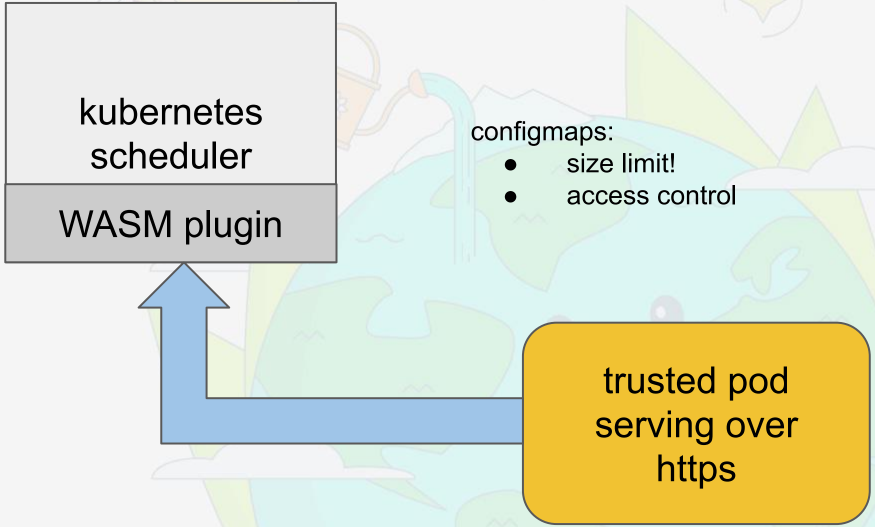
- toolchain maturity
- distribution/deployment

WASM in kubernetes scheduler: operations

deploying the scheduler

shipping the WASM plugins

securing the flow: “just” fetch the data from a trusted source?



configmaps:

- size limit!
- access control

Title

Using WASM/WASI in golang in 2023 and beyond

TL;DR: promising, but rough edges

- WASI **PREVIEW** 1
- golang 1.21 support
  - lack of `go:wasmexport` [1][2]
- sharing data between host and guest
- tinygo
  - incomplete reflection
  - hit/miss package support

THE END

fromani@redhat.com
https://github.com/ffromani

fromani@gmail.com

Thanks for attending! Questions?

Reading non-owned files

```
func main() {
        fmt.Printf("running as uid=%v gid=%v\n", os.Getuid(), os.Getgid())

        fi, err := os.Stat("/proc/cpuinfo")
        fmt.Printf("cpuinfo stat err = %v\n", err)
        if err == nil {
                // TODO
        }

        stat, ok := fi.Sys().(*syscall.Stat_t)
        if !ok {
                // TODO
        }

        fmt.Printf("stat: uid=%v gid=%v\n", stat.Uid, stat.Gid)


         , err = os.ReadFile("/proc/cpuinfo")
        fmt.Printf("cpuinfo read = %v\n", err)
}
```

Reading non-owned files

```
$ wasmedge --dir /:/ --dir .:. cpuinfo-wasi-go.wasm
running as uid=1 gid=1
cpuinfo stat err = <nil>
stat: mode=-rw-------
stat: uid=0 gid=0
cpuinfo read = open /proc/cpuinfo: Permission denied #
!!!!!!
cpuinfo data = 0
```

Reading non-owned files

```
$ wasirun --dir / cpuinfo-wasi-go.wasm
running as uid=1 gid=1
cpuinfo stat err = <nil>
stat: mode=-rw-------
stat: uid=0 gid=0
cpuinfo read = <nil>
cpuinfo data = 25159
```

Reading non-owned files in a container

```
$ podman run \
--annotation "module.wasm.image/variant=compat" \
-v /proc:/proc quay.io/fromani/cpuinfo-wasi-go:latest
running as uid=1 gid=1
cpuinfo stat err = <nil>
stat: mode=-rw-------
stat: uid=0 gid=0
cpuinfo read = open /proc/cpuinfo: Permission denied #!!!

no wasirun bindings :(
```

TinyGO & WASI

```
$ tinygo build -o main.wasm -target=wasi main.go


$ stat -c '%n %s' example-wasi-*.wasm
example-wasi-go.wasm     2375217
example-wasi-tinygo.wasm  740301
```

extending with WASM/WASI

how to improve?

```
func (wh *wasmHandler) ServeHTTP(
    w http.ResponseWriter,
    r *http.Request
) {
    // load module          -> extract: loader

    // create runtime       -> extract: engine

    // instantiate module   -> extract: engine

    // process request      -> orchestrate
}
```

extending with WASM/WASI

the engine

```go
// xref: https://github.com/tetratelabs/wazero/issues/985
type wasmEngine struct {
        code wazero.CompiledModule
        rt   wazero.Runtime
}


func newWasmEngine(ctx context.Context, wasmObj []byte) (*wasmEngine, error) {
        rt := wazero.NewRuntime(ctx)

        wasi_snapshot_preview1.MustInstantiate(ctx, rt)

        code, err := rt.CompileModule(ctx, wasmObj)
        if err != nil {
                return nil, err
        }

        return &wasmEngine{
                rt:   rt,
                code: code,
        }, nil
}

func (we *wasmEngine) Close(ctx context.Context) error {  return we.rt.Close(ctx); }
```

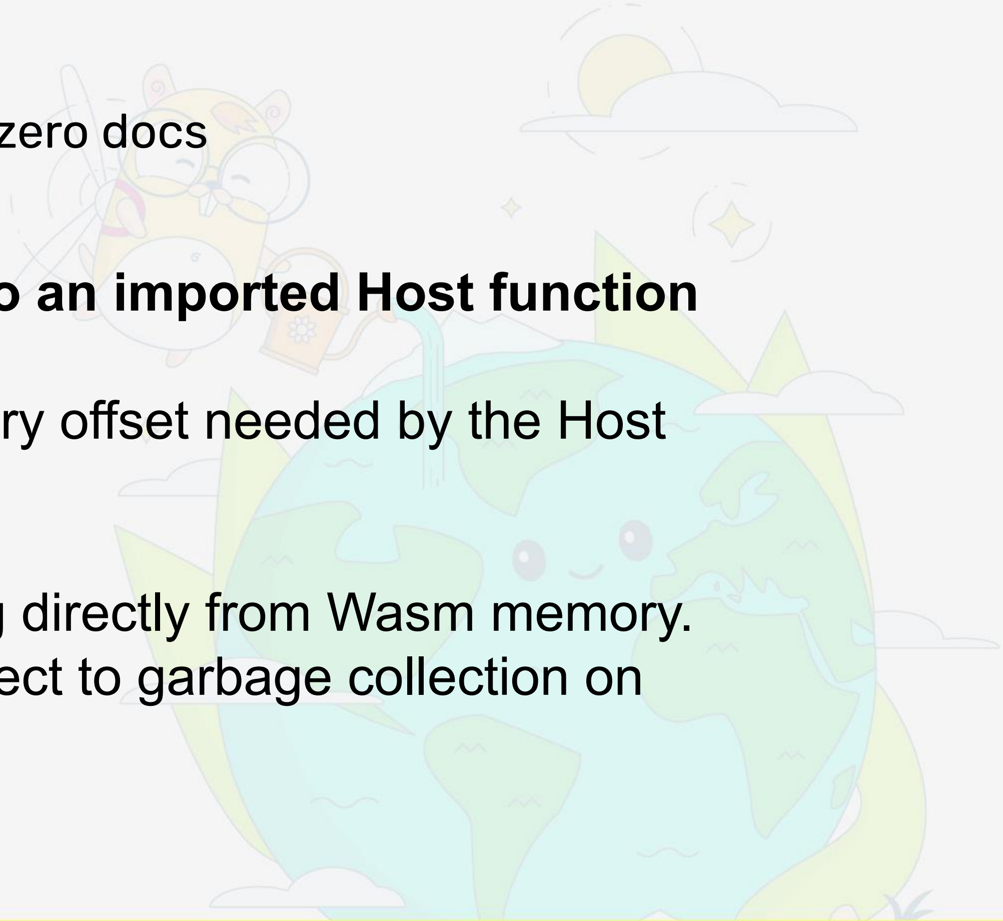
the engine - part 2

```
func (we *wasmEngine) Run(ctx context.Context, name string, …)
{
    // ...

    conf := wazero.NewModuleConfig().WithName(name) // ...

    mod, err := we.rt.InstantiateModule(ctx, we.code, config)
    if err != nil {
        // ...
    }

    mod.Close(ctx)

    return stdout.String(), stderr.String(), nil
}
```

the handler, revisited

```go
type wasmHandler struct {
    engine *wasmEngine
    name   string
}

func (wh *wasmHandler) ServeHTTP(w http.ResponseWriter, r *http.Request) {
    ctx := context.Background()

    stdout, stderr, err := wh.engine.Run(
        ctx,
        wh.name,
        r.Body,
        wh.makeEnviron(r),
    )

    // ...

    fmt.Fprint(w, stdout)
}
```

manual memory management: wazero docs



Guest passes a string to an imported Host function

Guest [...] gets the memory offset needed by the Host function.

The host reads that string directly from Wasm memory. The original string is subject to garbage collection on the Guest [...].

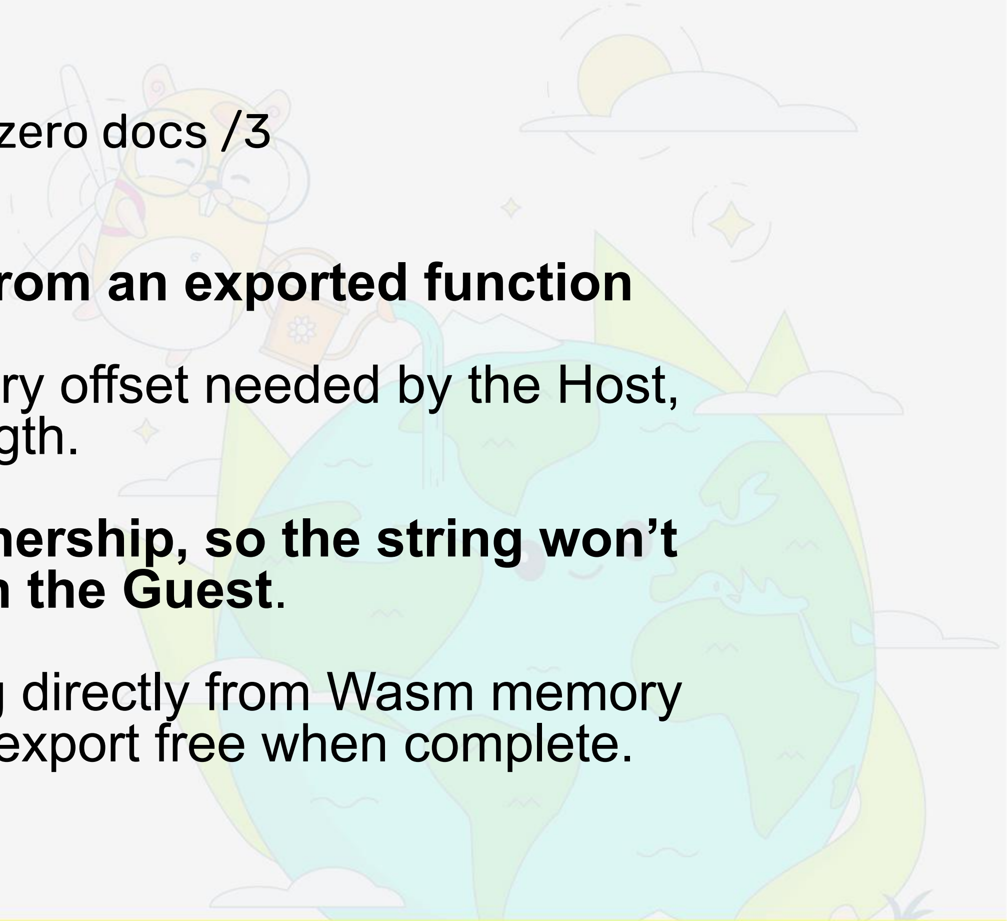
manual memory management: wazero docs /2

Host allocates a string to call an exported Guest function

Host **calls the built-in export malloc** [...].
The host owns that allocation, **so must call the built-in export free when done**.

The Guest [...] retrieves the string from the Wasm parameters.

manual memory management: wazero docs /3



Guest returns a string from an exported function

Guest [...] gets the memory offset needed by the Host, and returns it and the length.

This is a transfer of ownership, so the string won’t be garbage collected on the Guest.

The host reads that string directly from Wasm memory and must call the built-in export free when complete.