Extending Envoy with Wasm from start to finish

Ed Snible / @esnible / IBM Research

Ed Snible
IBM **Research**

IstioCon

Includes
LIVE DEMO

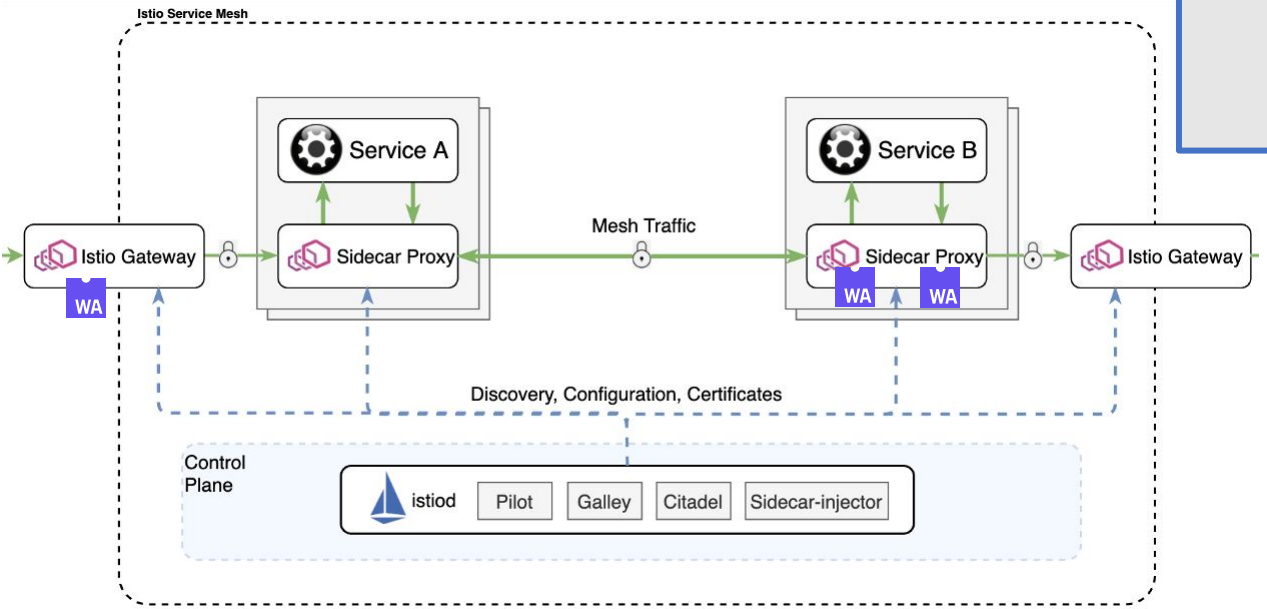
With: **The first dozen places I got stuck writing filters**

#IstioCon

The trouble is that men very often resort to all sorts of devices in order not to think because thinking is such hard work

—Thomas J. Watson

Istio Architecture



Sprinkle a little WA into your mesh

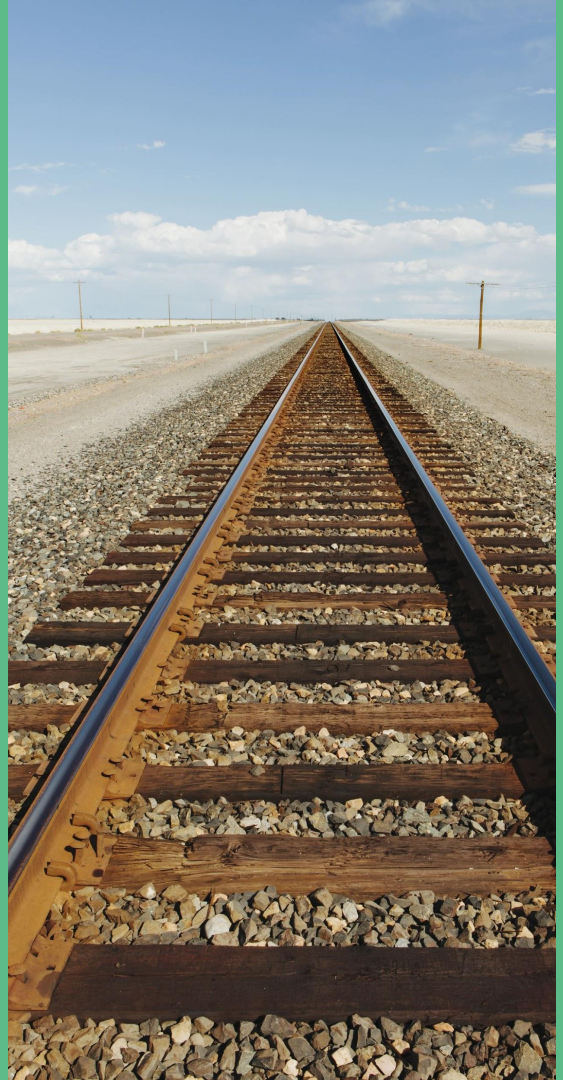
docker.io/mycompany/a:latest

docker.io/mycompany/b:latest

Istio **ServiceEntry**

Istio **VirtualService**

Istio **EnvoyFilter** WA



Why is WebAssembly for Proxies different and better?

The full power of

- a mainstream programming language
- running in an efficient engine
- potentially inspecting every communication between unmodified microservices

Is it safe to let arbitrary code run on the sidecar?

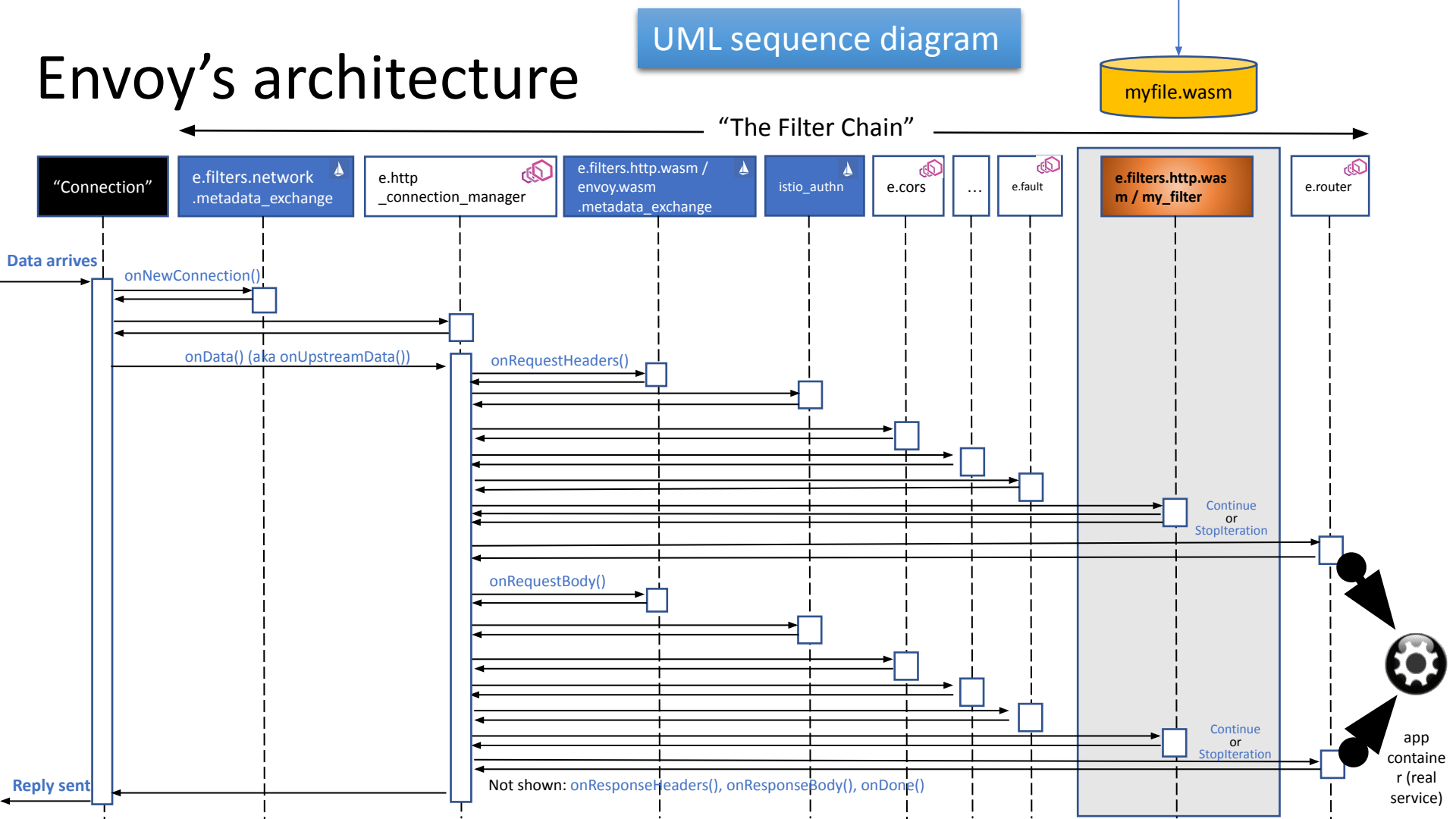
Yes!

- The ability to create EnvoyFilters is controlled by RBAC.
  - Only trusted people should be adjusting mesh behavior.
- Extension code runs in a sandbox. The only networking available is HTTP and GRPC calls
  - When making those calls, only clusters already known to Envoy can be used. No connecting to arbitrary hosts!

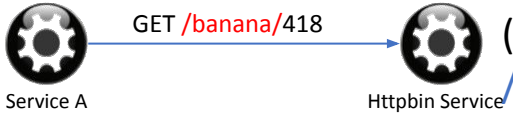
But …

- Developers can cause trouble with unbounded recursion and memory exhaustion
- Admins with access to the pods can point at any storage, admins with access to the storage can replace expected binaries
- Currently no audit tools for .wasm in storage (that I know of).

Envoy's architecture



Example: URL rewrite to match client and server



(expects GET /status/418)

C++

```
FilterHeadersStatus RegexpRepl::onRequestHeaders(
      uint32_t hdr_count, bool eos) {

   std::regex re("banana/([0-9]*)”);
   char const *replaceWith = "status/$1";

   WasmDataPtr wdpGhmv =
getRequestHeader(":path");

   std::string result = std::regex_replace(
      wdpGhmv->toString(), re, replaceWith);
   WasmResult wrRrh = replaceRequestHeader(
      ":path", result);

   return FilterHeadersStatus::Continue;
}
```

AssemblyScript (JavaScript / TypeScript)

```
onRequestHeaders(hd_cnt: u32, eos: bool):
  FilterHeadersStatusValues {

   // (TODO: Current AssemblyScript compiler refuses to compile RegExp)
  let re = new RegExp("banana/([0-9]*)");
  let replaceWith = "status/$1";

  let sValue = stream_context.headers.request.get(":path");

  let result = sValue.replace(re, replaceWith);
  stream_context.headers.request.replace(
    ":path", result);

  return FilterHeadersStatusValues.Continue;
}
```

“convert GET */banana/nnn* to GET */status/nnn*”

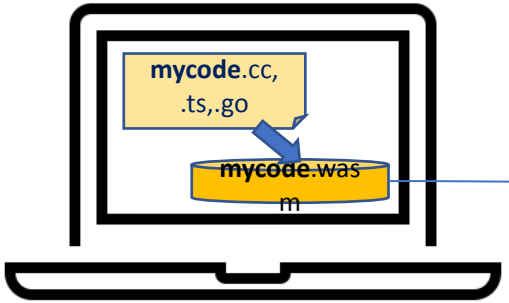
The stuff in blue comes from the SDK
The black stuff comes from your language

Source code repo for this talk:
github.com/esnible/wasm-examples

An Istio contributor's first experiences

It is easy to find the SDKs and follow their tutorial steps to produce .wasm byte code

Compilation takes a few seconds!

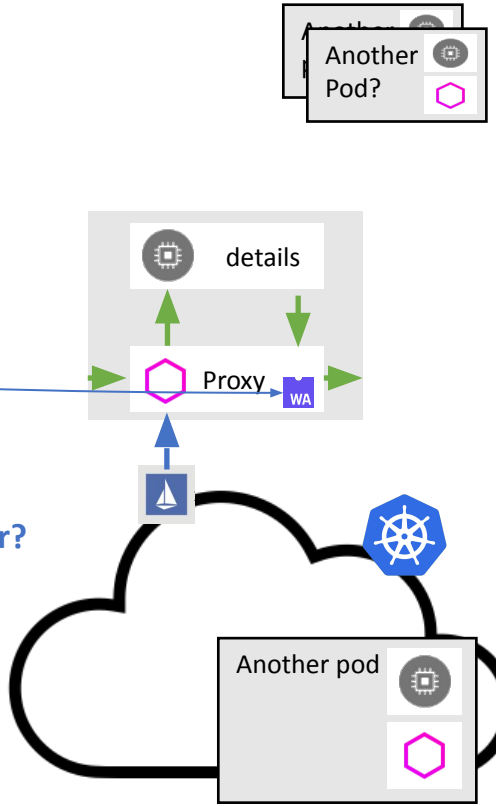


C++: *docker run --workdir … --volume … wasmsdk:v2 /build_wasm.sh*

JavaScript: *npm run asbuild*

Go: *docker run -it –workdir … --volume … tinygo/tinygo:0.16.0 \ tinygo build -o … -scheduler=none -target=wasi …/main.go*

WebAssemblyHub?
EnvoyFilter yaml?
Solo.io Wasme operator?

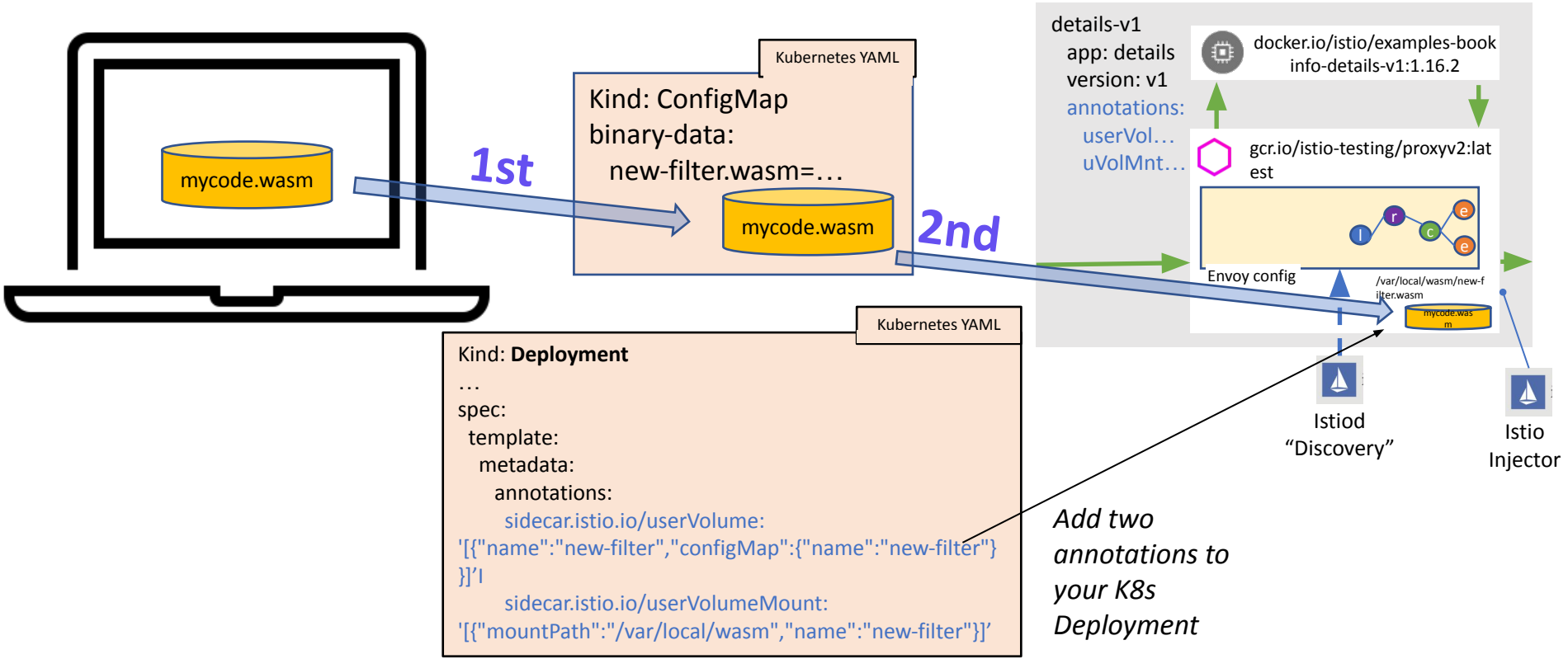


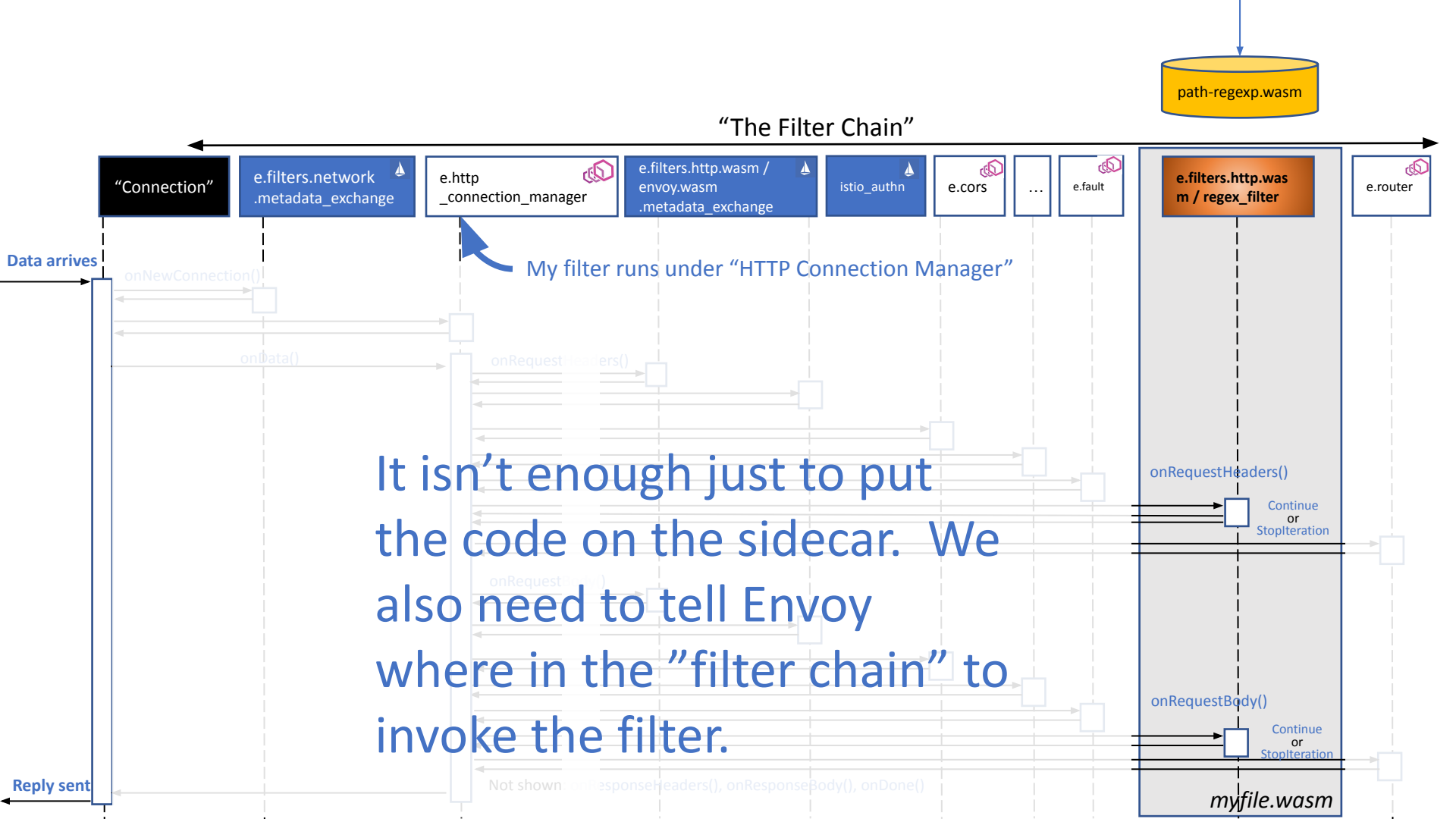
From a laptop to the sidecar

ConfigMaps are good for experimentation.

Production: Kubernetes volumes or (Istio 1.9) HTTP remote fetch of Wasm

```
kubectl create configmap new-filter \
  --from-file=new-filter.wasm=mycode.wasm
```





It isn't enough just to put the code on the sidecar. We also need to tell Envoy where in the "filter chain" to invoke the filter.

Which pods, where in Envoy, and your filter's parameters

EnvoyFilter

Istio YAML

```
kind: EnvoyFilter
metadata:
  name: httpbin-myfilter
spec:
workloadSelector:
   labels:
    app: httpbin
  configPatches:
  - applyTo: HTTP_FILTER
   match:
     context: SIDECAR_INBOUND
     listener:
       filterChain:
        filter:
          name: envoy.http_connection_manager
          subFilter:
            name: envoy.router
patch:
     operation: INSERT_BEFORE
     value:
       name: myfilter
       typed_config:
        '@type': type.googleapis.com/udpa.type.v1.TypedStruct
      type_url: type.googleapis.com/envoy.extensions.filters.http.wasm.v3.Wasm
       value:
         config:
           configuration:
             '@type': type.googleapis.com/google.protobuf.StringValue
             value: "{ "regex": "banana/([0-9]*)", "replace": "status/$1" }"
             root_id: "regex_repl"
             vm_config:
               code:
                 local:
                   filename: /var/local/wasm/new-filter.wasm
               runtime: envoy.wasm.runtime.v8
               vm_id: myexperiment-1.9
```

Pods to receive configuration

Same pods as

`kubectl get pods -selector app=httpbin`

(Leave this blank if every pod should get it.)

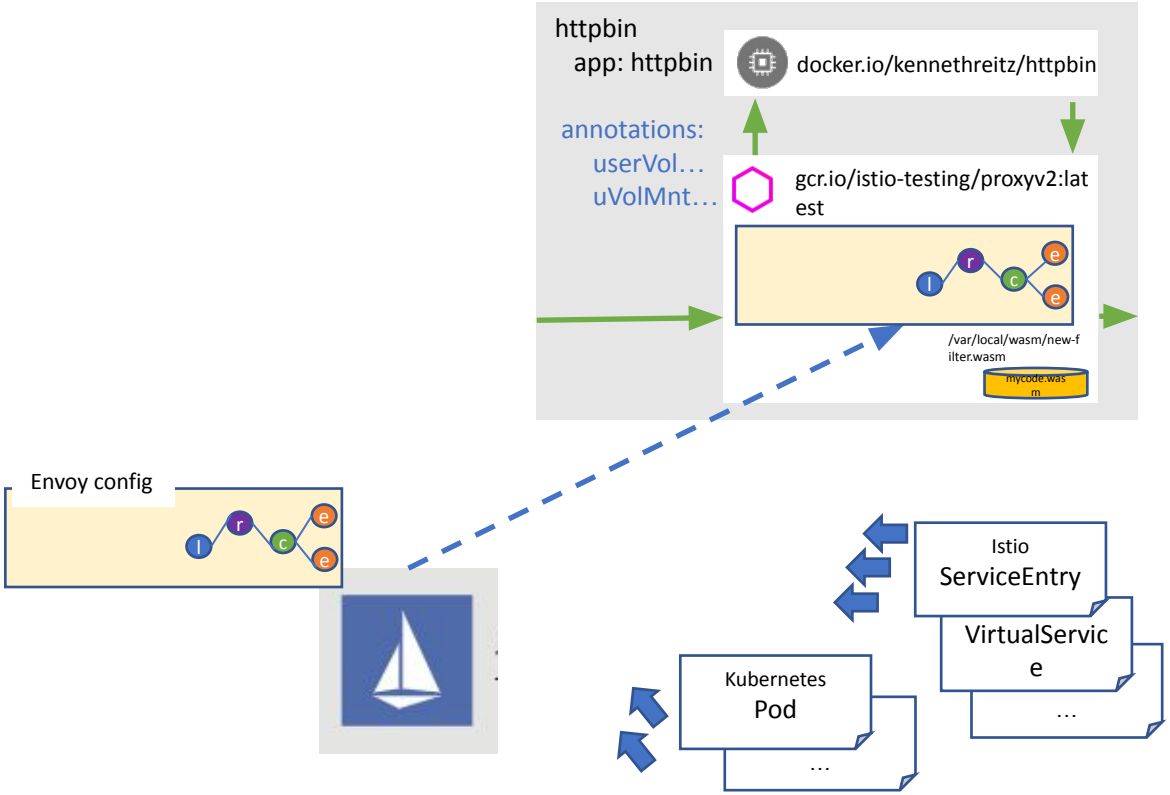
Where to attach the Envoy configuration

To HTTP connection manager

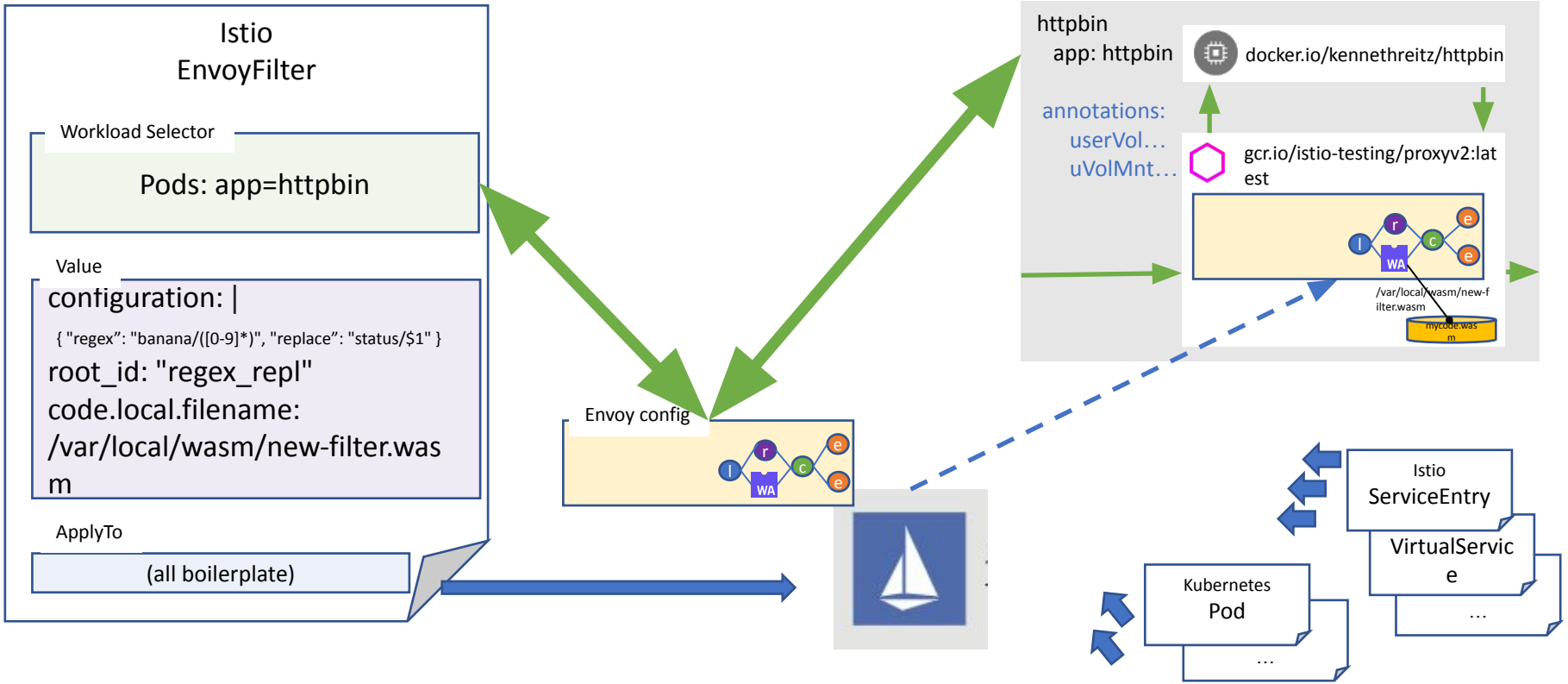
Some Envoy configuration

- Config parameters for the filter
- We named our extension "**regex_repl**" in the source code
- We mounted it at **/var/local/wasm/new-filter.wasm**

Before: New .wasm unknown to Envoy



After: Running code on Envoy



Thank You!

But wait, there’s more…

The first dozen places I got stuck when I tried to write my own filters

Do I need to write a filter at all?

- Before you begin, check Envoy's catalog to see if there is already a filter that does what you want!
  - https://www.envoyproxy.io/docs/envoy/latest/api-v3/config/filter/filter
  - If so, you can add use Istio's EnvoyFilter to add it to your pods, without writing a new filter.
  - *Warning: Possible upgrade risk*
- The functionality might be elsewhere.
  - **The Regex filter I'm showing is not needed; Envoy exposes regex_rewrite on `config.route.v3.RouteAction`**
- Github might have what you need (in source format)
  - Start at https://github.com/istio-ecosystem/wasm-extensions
- Solo.io's WebAssembly Hub might already have what you need (as a binary)

Network Filters

Echo
SNI Cluster
Client TLS authentication
Direct response
Dubbo Proxy
Dubbo Proxy Route Configuration
Echo
Network External Authorization
HTTP connection manager
Kafka Broker
Local rate limit
Mongo proxy
MySQL proxy
Postgres proxy
Rate limit
RBAC
Redis Proxy
RocketMQ Proxy
Rocketmq Proxy Route Configuration
SNI Cluster Filter
SNI dynamic forward proxy
TCP Proxy
Thrift Proxy Route Configuration
Thrift Proxy
Wasm
ZooKeeper proxy

HTTP Filters

CORS processing
AWS DynamoDB
gRPC HTTP/1 bridge
gRPC Web
Adaptive Concurrency
AWS Lambda
AwsRequestSigning
Buffer
Compressor
Cors
CSRF
Decompressor
Dynamic forward proxy
Dynamo
External Authorization
Fault Injection
gRPC HTTP/1.1 Bridge
gRPC HTTP/1.1 Reverse Bridge
gRPC-JSON transcoder
gRPC statistics
gRPC Web
Gzip
Header-To-Metadata Filter
Health check
IP tagging
JWT Authentication
Kill Request
Local Rate limit
Lua
OnDemand
Original Src Filter
Rate limit
RBAC
Router
Squash
Tap
Wasm

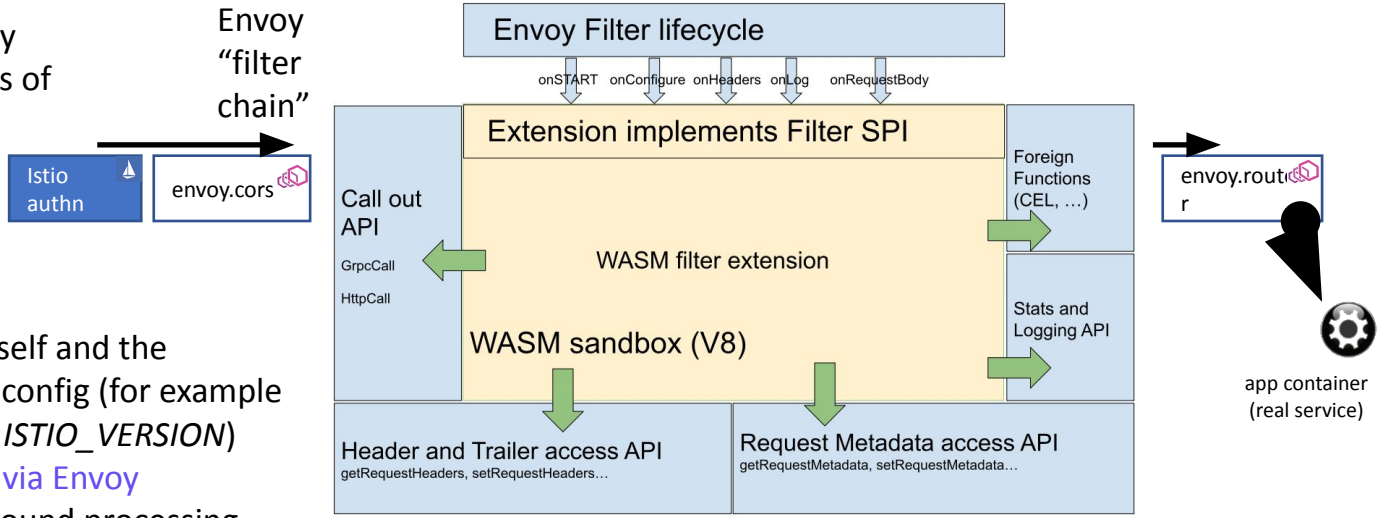
Can the filter be implemented?

Filters run in a "sandbox" with significant restrictions

- No access to operating system.
- `std::getenv("PATH")` returns ""
- You can't do networking things
- C++ `sleep()` doesn't sleep, no JavaScript `setTimeout()`
- The only calls allowed are Envoy functions and the SDK functions of your language.

Language restrictions

- AssemblyScript has no JavaScript regular expressions, no JSON.stringify()
- JSON parsing in TinyGo works (but build env currently requires Bazel)



Filters can

- Read Envoy configuration for itself and the *bootstrap* section of the Envoy config (for example the *node.id*, or *node.metadata.ISTIO_VERSION*)
- Limited HTTP and GRPC access via Envoy
- *onTick()* can be used for background processing

Diagram: https://istio.io/latest/docs/concepts/wasm/

Coding style and SDK stumbling blocks

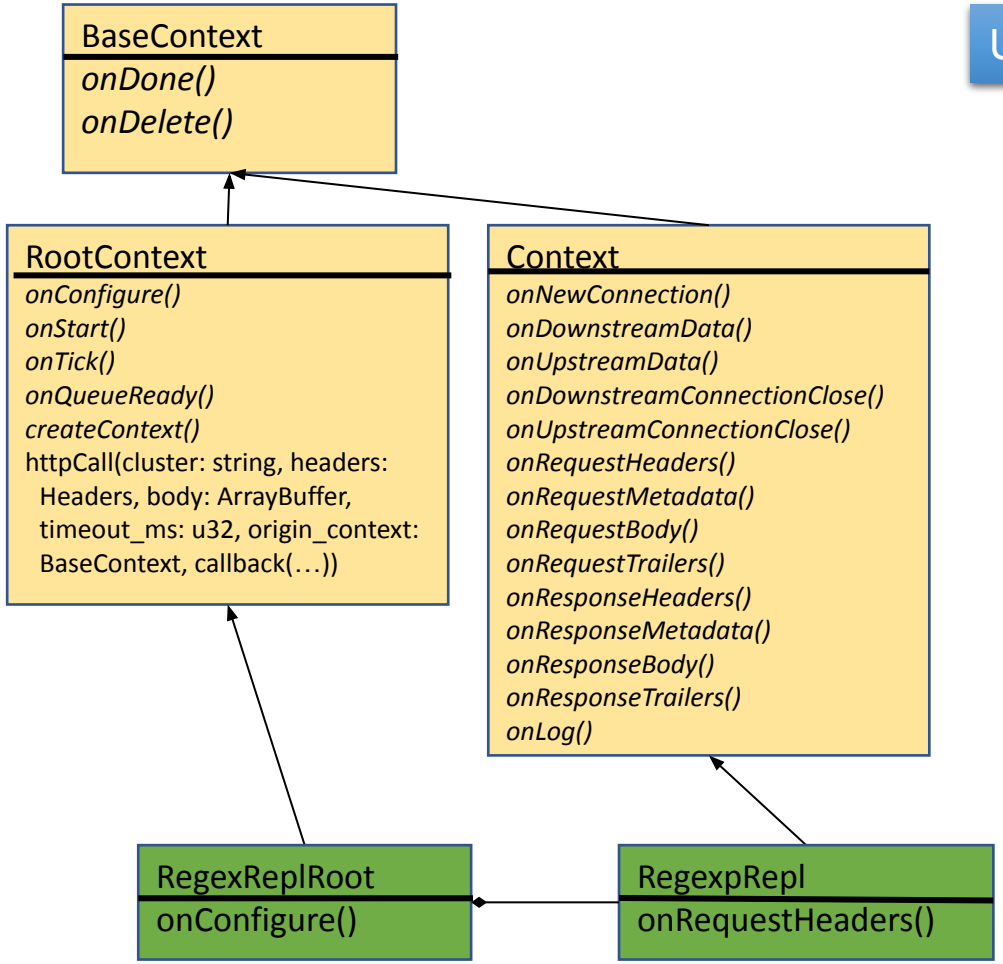
- Filters implement use a non-blocking “event-driven” flow, like JavaScript.
  - It helps if you have coded in that style before.
- Descriptions of many constants you’ll need are only given in Envoy headers
  - https://github.com/envoyproxy/envoy/blob/master/include/envoy/http/filter.h
- Where is *main()*?
- Read through the source of the SDK for your language
  - You need to know SDK-provided global methods
  - Your editor’s auto-completion isn’t enough

UML model of SDKs

This particular example comes from Solo.io's AssemblyScript SDK, but the C++ SDK is similar.

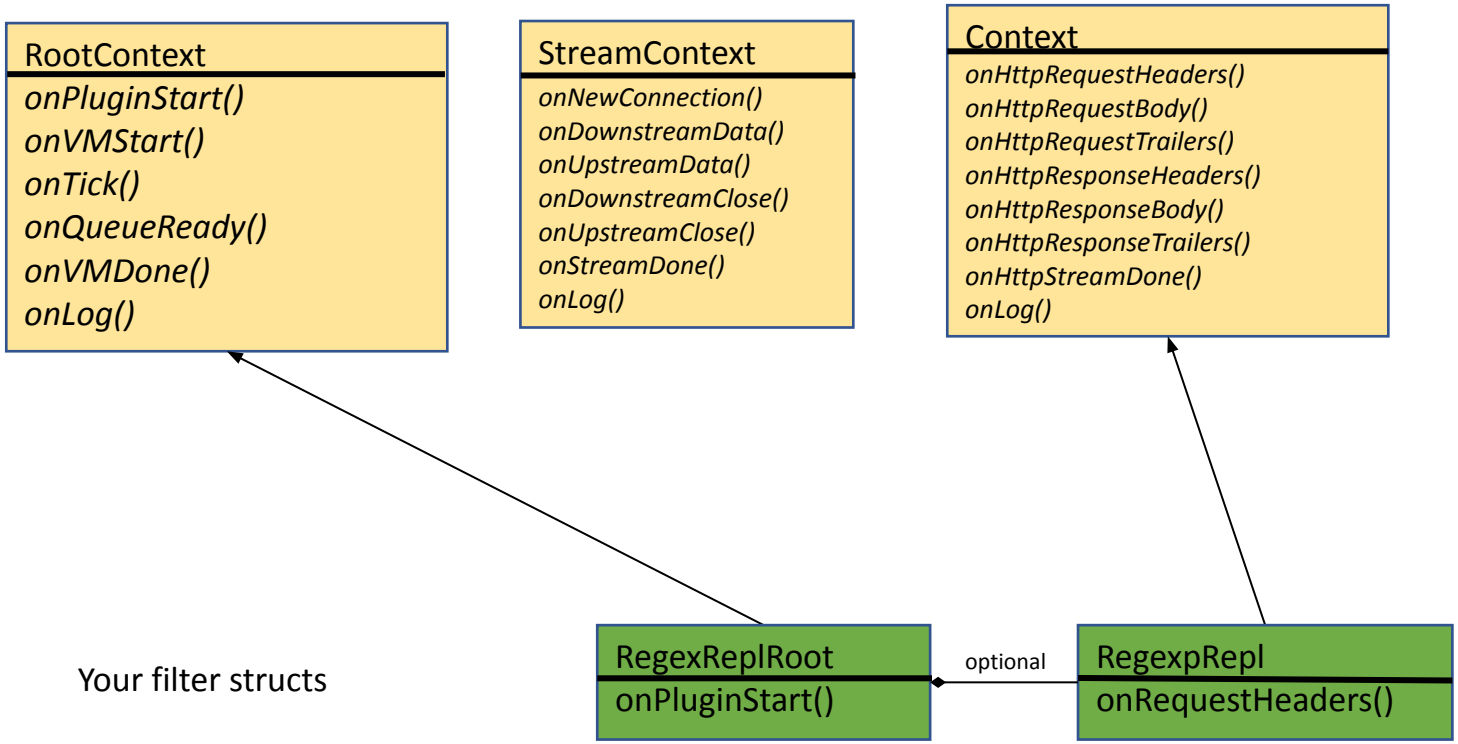
SDK-provided base classes

Your filter classes



UML model of Go SDK

Interfaces



The Go SDK is factored to make a distinction between HTTP and stream filters.

How WASM code using the SDK starts

- **JavaScript**

```
registerRootContext( () => {
   return RootContextHelper.wrap(new RegexReplRoot());
}, "regex_repl");
```

- **C++**
  - There's no `main()`

```
static RegisterContextFactory register_AuthContext(
   CONTEXT_FACTORY(RegexRepl),
   ROOT_FACTORY(RegexReplRoot),
   "regex_repl");
```

- **Go** (TinyGo)
  - There is a `main()`

```
func newContext(rootContextID, contextID uint32) proxywasm.HttpContext { return
&regexRepl{contextID: contextID} }
func main() {
   proxywasm.SetNewHttpContext(newContext)
   // Go SDK doesn't offer a string name!
}
```

Envoy's loading of your .wasm file triggers static constructors (C++), *main()* (Go), or code outside methods (AssemblyScript)

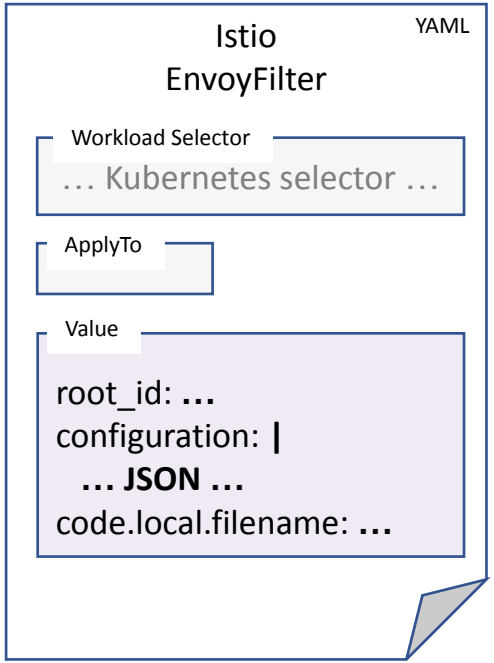
Envoy starts your code and calls onStart() three times – or more if Envoy's *--concurrency* is set.

Configuring filters

Deployers/Admins: Pass serialized JSON

```
root_id: regex_repl
configuration: |
  {
    "regex": "banana/([0-9]*)",
    "replace": "status/$1"
  }
```

Yes, JSON inside YAML!



Developers: Get bytes, explicitly deserialize

C++:

- WasmDataPtr wdp = getBufferBytes(WasmBufferType::PluginConfiguration, 0, configuration_size);

AssemblyScript:

- let s = this.root_context.getConfiguration();

Go:

- data, err := proxywasm.GetPluginConfiguration(pluginConfigurationSize)

The Go and C++ versions can ONLY be called during *onConfigure()*. Most global functions have that characteristic.

The header handler isn't supplied with the headers…

C++

Context

```
virtual FilterHeadersStatus
onRequestHeaders(uint32_t, bool) {
  …
}
```

```
FilterHeadersStatus Context::onRequestHeaders(
  uint32_t hdr_cnt, bool eos) {

    WasmDataPtr wdpUserAgent = getRequestHeader("User-Agent");
    std::string sStatus = wdpUserAgent->toString();

    WasmDataPtr headerPairs = getRequestHeaderPairs();
    auto headerVec = headerPairs->pairs();
```

AssemblyScript (JavaScript / TypeScript)

```
onRequestHeaders(a: u32, end_of_stream:
bool): FilterHeadersStatusValues {
  …
}
```

```
onRequestHeaders(hdr_count: u32, eos: bool):
  FilterHeadersStatusValues {

    let sStatus: string =

    stream_context.headers.request.get("User-Agent")

    let headerPairs: Headers =
     stream_context.headers.request.get_headers()
```

onXXXHeaders() parameters don't provide access. Global functions fetch them.

httpCall()

```
WasmResult httpCall(
   std::string_view uri, const HeaderStringPairs &request_headers,
   std::string_view request_body, const HeaderStringPairs &request_trailers,
   uint32_t timeout_milliseconds, HttpCallCallback callback);
```

I will be publishing an entire blog on how to use this function. It's also in the GitHub repo I mentioned earlier.

- "uri" isn't a URI, it's an Envoy "cluster name"
- The HTTP Method, Path and Authority as treated as request "headers"
  - Envoy is picky about parameters and miserly about error response info
- The callback is async
  - onXXX() methods invoke httpGet() then return *StopIteration*
  - Callbacks must call *setEffectiveContext()* **and** *continueRequest()*

Debugging

- If you are coming from a browser-based Wasm environment, you may expect a debugger such as Chrome DevTools.
  - I haven't found anything like that. I'm using log statements!
  - *istioctl proxy-config log deployment/httpbin --level wasm:trace*
  - *kubectl logs deploy/httpbin -c istio-proxy --follow*
- Watch out for "linker errors":
  - *error envoy wasm Failed to load Wasm module due to a missing import: env._ZN14RegexpReplRoot6onTickEv*
- To get Envoy to pick up changes after a recompile delete then recreate the Istio EnvoyFilter.
  - It isn't sufficient to just change the underlying mounted storage
  - Watch out for timing
    - Kubernetes can be slow to update ConfigMap mounts (up to a minute)
    - Envoy can be slow to stop running plugins, also up to a minute, log statements from the earlier version will appear during this time.

Open Source Landscape

envoy-wasm
Envoy itself w/WASM and
envoy_filter_http_wasm_example.cc

https://envoyproxy.slack.com/
#envoy-wasm

Envoy itself,
many contributors,
CNCF

Istio itself

proxy
WASM plugins used by Istio
(Stats and Metadata Exchange)

https://isto.slack.com/
#extensions-telemetry

wasm-extensions
Examples

spec
ABI
Specification

proxy-wasm-c
pp-sdk
C++ for plugins

proxy-wasm-r
ust-sdk
Rust plugins

proxy-wasm-c
pp-host
C++ for host

proxy-runtime
AssemblyScrip
t SDK

wasme
tooling

https://solo-io.slack.com/
#web-assembly-hub

https://github.com
/tetratelabs

proxy-wasm-go-sdk
TinyGo SDK

Bibliography

- Documentation:
  - https://istio.io/latest/docs/concepts/wasm/
- Blogs:
  - Yaroslav Skopets, “Taming a Network Filter”
    - https://blog.envoyproxy.io/taming-a-network-filter-44adcf91517
    - This video shows TCP filters, which I did not cover today!
- Videos:
  - Idit Levine and Scott Weiss’ IstioCon presentation https://events.istio.io/istiocon-2021/sessions/developing-debugging-webassembly-filters/ **earlier today!**
  - Daniel Grimm, “Hacking the Mesh: Extending Istio with WebAssembly Modules”, *DevNation* https://www.facebook.com/openshift/videos/375151853879233 *(starts 1:45 minutes in)*
  - John Plevyak & Dhi Aurrahman, “Extending Envoy with WebAssembly”, KubeCon EU 2019, **https://www.youtube.com/watch?v=XdWmm_mtVXI**
    - John Plevyak, starts 13 minutes in, gives a high-level overview
- Source code repo for this talk: github.com/esnible/wasm-examples

Backup

- (Backup slides, if we have a Q&A and I am asked about specific surprises)

Unpleasant Surprises

C++

```
WasmDataPtr headerPairs = getRequestHeaderPairs();
auto headerVec = headerPairs->pairs();
```

This code only works in onRequestHeaders(). If you call it later, for example in onResponseHeaders(), it's empty. (Save a copy of anything you'll use beyond the current member function.)

```
auto headerPairs = getRequestHeaderPairs() ->pairs();
```

This code *almost* works. It's an easy mistake to make if you are coming from a garbage-collected language.

Tip: don't use *auto.*
Use *std::vector<std::pair<std::string_view, std::string_view>>*

C++ programmers know to be wary of implicit destruction and *free()* when they see *string_view.*

AssemblyScript (JavaScript / TypeScript)

```
let headerPairs: Headers
 =
stream_context.headers.request.get_headers
()
```

Making httpCalls

SDK API looks straightforward:

```
WasmResult httpCall(
   std::string_view uri, const HeaderStringPairs &request_headers,
   std::string_view request_body, const HeaderStringPairs &request_trailers,
   uint32_t timeout_milliseconds, HttpCallCallback callback);
```

Three of the parameters confuse beginners

- uri isn't a URI !!!
- Is that a GET or a POST? Where to put HTTP PATH and Method?
- How does my std::function<void(uint32_t, size_t, uint32_t)> callback get a this into my Context?

Troubleshooting failures

No detailed error messages. Any mistakes receive *WasmResult::BadArgument,* an enum which is the digit 2

After my callback has done, how do I "rejoin" or "rendezvous" with the onXXX() method?

http method and path

```
WasmResult httpCall(std::string_view uri, const HeaderStringPairs &request_headers,
std::string_view request_body, const HeaderStringPairs &request_trailers,
uint32_t timeout_milliseconds, HttpCallCallback callback);
```

Envoy uses “headers” for non-header things such as the HTTP method. Prepended with a colon. I figured this out from Lua examples!

```
std::vector<std::pair<std::string, std::string>> callHeaders;
callHeaders.push_back(std::pair<std::string, std::string>(":method", "GET"));
callHeaders.push_back(std::pair<std::string, std::string>(":path", "/"));
callHeaders.push_back(std::pair<std::string, std::string>(":authority", "example.com"));

WasmResult wr = this->root()->httpCall(
  "outbound|80||example.com",
  callHeaders,
  "", HeaderStringPairs(), 5000, …);
```

“uri” must be a cluster name, Something you see at :15000/clusters when looking at the Envoy dashboard, And perhaps defined with a ServiceEntry.

httpCall callback (C++)

```
WasmResult httpCall(std::string_view uri, const HeaderStringPairs &request_headers,
std::string_view request_body, const HeaderStringPairs &request_trailers,
uint32_t timeout_milliseconds, HttpCallCallback callback);
```

AssemblyScript Works well.

C++ Use ***bind()***, similar to what you would do with JavaScript, to make a regular function pointer out of a member function.

```
WasmResult wr = this->root()->httpCall(
  "outbound|80||example.com",
  callHeaders,
  "", HeaderStringPairs(), 5000,
  std::bind(&MyClass::myHttpCallback, this,
    std::placeholders::_1, std::placeholders::_2, std::placeholders::_3));
```

httpCall callback (Go)

```
DispatchHttpCall(upstream string,
headers [][2]string, body string, trailers [][2]string,
timeoutMillisecond uint32, callBack HttpCalloutCallBack) (calloutID uint32, err error)
```

```
_, err := proxywasm.DispatchHttpCall(
  "outbound|80||example.com",
  headers,
  "", [][2]string{},
  5000, createMyHttpCallback(ctx))
```

Go: Write a function that returns a *closure*

```
func createMyHttpCallback(mc *myClass) proxywasm.HttpCalloutCallBack {
 return func(numHeaders, bodySize, numTrailers int) {
  mc.httpCallResponseCallback(numHeaders, bodySize, numTrailers)
 }
}
```

The httpCall doesn't block

```
FilterHeadersStatus MyContext::onRequestHeaders(uint32_t
header_count, bool end_of_stream) {

  …

  WasmResult wr = this->root()->httpCall(…)
  if (wr == WasmResult::Ok) {
    return FilterHeadersStatus::StopIteration;
  }

  logWarn("context_id:" + strId_ + " MyContext
::onResponseHeaders() httpCall() failed. wr is " + toString(wr));
  return FilterHeadersStatus::Continue;
}
```

Ok doesn't mean the HTTP succeeded with a *200 OK*. It means the parameters were accepted and the callback will be invoked in the future.

StopIteration tells Envoy to stop working for now. Later we'll rejoin the filter chain where we left off.

Continue tells Envoy to let the rest of the filters run. If we don't return this, nothing happens and the caller may eventually time out.

If you've programmed in Node.js, this should make sense

Handling the callback

The callbacks receive a size and some counts. Get the real headers and body from global functions

```
void MyContext::myHttpCallback(uint32_t headerCount, size_t body_size, uint32_t trailerCount) {
  WasmDataPtr wdpGhmv = getHeaderMapValue(WasmHeaderMapType::HttpCallResponseHeaders, ":status");
  uint32_t status = std::stoi(wdpGhmv->toString());
  WasmDataPtr wdpBody = getBufferBytes(WasmBufferType::HttpCallResponseBody, 0, body_size);

  WasmResult wrSec = setEffectiveContext();
  if (wrSec != WasmResult::Ok) {
    logWarn("context_id:" + strId_ + " Experiment::myHttpCallback() setEffectiveContext() FAILED. wrSec is "
      + toString(wrSec));
  }

   … your code …

  WasmResult wr = continueRequest();
  if (wr != WasmResult::Ok) {
    logWarn("context_id:" + strId_ + " MyContext ::myHttpCallback() continueRequest() FAILED. wr is " + toString(wr));
  }
}
```

After *setEffectiveContext()* you can call *setResponseHeaderPairs()* and friends, *continueRequest(),* or even *sendLocalResponse().*
If you don't call *continueRequest()* or *sendLocalResponse(),* the caller will not receive anything and may eventually time out.

If you forget *setEffectiveContext()* nothing happens.
continueRequest() returns *Ok*, yet nothing happens.