What HTTP Needs from WebAssembly

(IMHO)

Cover Pass-Through Use Cases

- I.e., a proxy plug-in API
- This is what VCL does for Fastly/Varnish
- E.g.,
  - Modify request headers (including method/path/query)
  - Modify response headers (including status code)
  - Modify cache behaviour
  - Synthesise a response based upon request attributes (including errors/ACL)
  - Route based upon request attributes
  - Advanced logging
  - Interpose authentication (HTTP auth, Cookie, etc.)
  - Manifest rewriting / ad insertion
- Implies a “fast path” / “forwarding” API
- Sometimes needs access to request/response bodies, sometimes not

Cover Aggregation Use Cases

- I.e., a traditional client / server API that can be “glued” together
- This is what people currently do with NodeJS, etc.
- E.g.,
  - Edge Side Includes (ESI)
  - GraphQL
  - Microservices/page assembly a la Y!, Amazon, etc.
- Implies ability to make multiple, parallel “back-end” requests
- Needs full access to request/response bodies

Allow Body Streaming

- Need to avoid buffering complete bodies
  - Performance
  - Resource management
- Hopefully this is pretty obvious
- But, it does bring some complexity

Hide the Framing Layer(s)

Guest code shouldn’t need to care about HTTP framing

- Security issues are often related to framing
- Guests may share connections, so they need to be isolated
- The host can optimise if it handles framing exclusively

This implies that messaging parsing/serialisation and chunking is all abstracted out

… and Normalise Input Where Possible

- E.g., complete request URI should be available, don’t force people to calculate it
- Trim whitespace on header values

… BUT Allow Granular Access to Header Data

Some people want to build WAFs and do other kinds of client profiling

Typically, this means they need access to the raw header names of the request, in order

Expose Both Headers and Trailers

Trailers are uncommon, but useful / critical for some use cases

E.g.,

- gRPC
- Server-Timing
- microservices

Implies a way of setting/getting that’s functional for both, and a way of distinguishing them

Allow Individual Header Access

- Many, many security problems are caused by incorrect header parsing
- HTTP headers are no longer textual on the wire in h2/h3
- HPACK/QPACK are designed to allow partial access; i.e., you don’t have to decompress the whole header block to grab a value
- A binary encoding of Structured Headers will (eventually) allow much more efficient wire formats for individual header values

All of this implies that we need:

- A way to get/set/delete individual header values by name, *and*
- A way to extend the API in the future for Structured Headers

Expose Non-Final Responses

HTTP has non-final (1xx) responses and final responses (everything else)

- They’re rare, but pretty useful when you need one
- It’s difficult to tack them on later (although possible, e.g., NodeJS)

Integrate Caching

- VCL’s “hooks” (vcl_hash, vcl_recv, vcl_hit, vcl_miss, etc.) provide a lot of capability and flexibility, but are sometimes difficult to work with
    - Modify request before cache lookup (including cache key)
    - Modify response before inserting it into cache
    - Modify response after retrieving it from cache
    - Vary behaviour based upon cache hit, miss, etc.
- These cases can be addressed in a client/server focused API with good Cache Control mechanisms - further work required

Expose Context

There’s a lot of contextual information / controls necessary for servers and clients:

- Peer and local IP / port
- HTTP version
- TLS version, cipher, SNI, ALPN
- Various timeouts
- Limits for HTTP header length, cardinality, body sizes, etc.
- Whether to implement features like body compression, redirects, retries

Implies: Context on the request/response exchange that can be read-only or read/write for metadata and controls.

Crude Straw-Man on the Back of an Envelope

```
serve_request() -> context
make_request() -> context
context.getmeta(name) -> value
context.setmeta(name, value)
context.request() -> message
context.response() -> message
message.headers   // fields
message.body      // streamable
message.trailers  // fields
message.forward(context)  // for pass-through use cases
fields.getNames() -> [name, ...]
fields.get(name) -> [field-value, ...]
fields.set(name, value)
fields.delete(name)
```

This explicitly ignores concurrency issues