Heroes of Modularity

Building Plugins with WebAssembly Power

TABLE OF CONTENTS

01

Set goals

Talk about what we want to create

02

Reseach

Consider what options we have

03

WASM

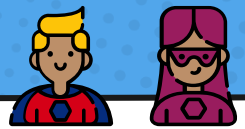
Use WebAssembly for our goal

04

Demo

Look at an example of a real application

OUR REQUIREMENTS



Cross platform

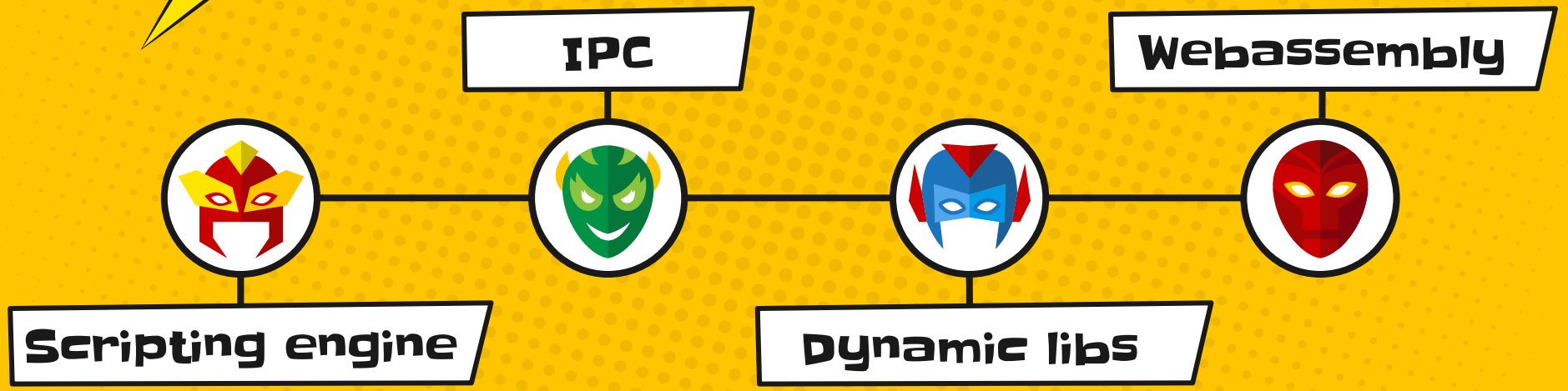
Low overhead

Safety

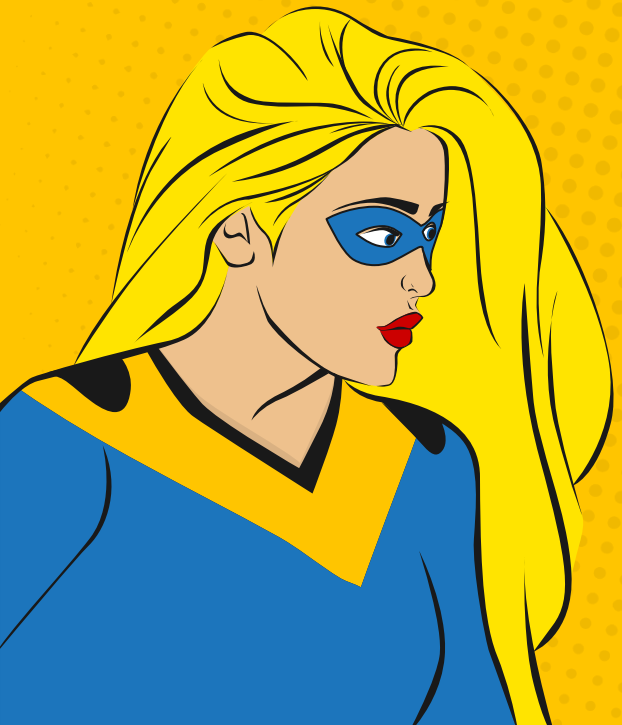
Backward compatibility

Port existing code

FOUR SUPERHEROES



SCRIPTING ENGINE



Cross-platform	★★★★★
Low overhead	★★★★☆
Safety	★★★★★
Back. Compatibility	★★★★★
Port existing code	☆☆☆☆☆

INTER PROCESS COMMUNICATION

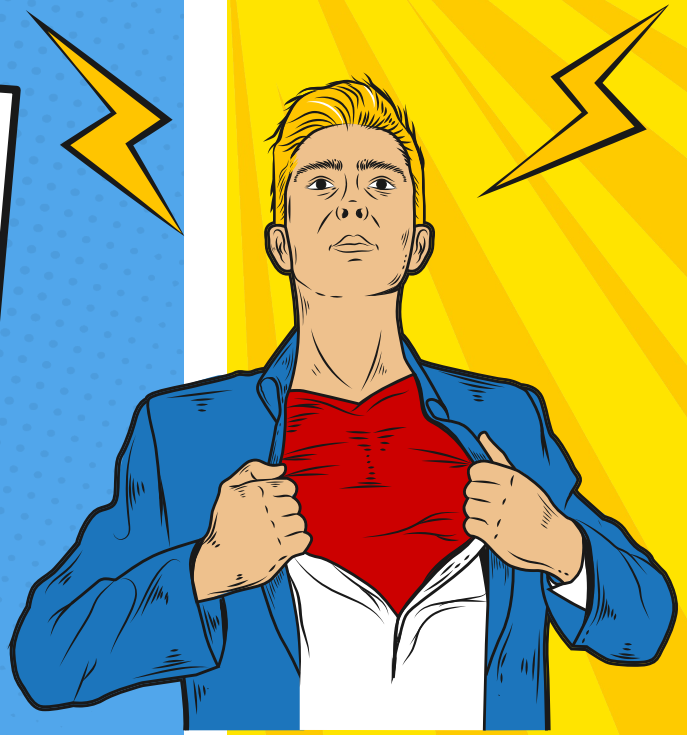
Cross-platform	★★★★★
Low overhead	★★★★☆
Safety	★★☆☆☆
Back. Compatibility	★★★★★
Port existing code	★★★★★

DYNAMIC LOAD LIBRARIES

Cross-platform	★★★★★
Low overhead	★★★★★
Safety	★★★★☆
Back. Compatibility	★★★☆☆
Port existing code	★★★★★

WEBASSEMBLY

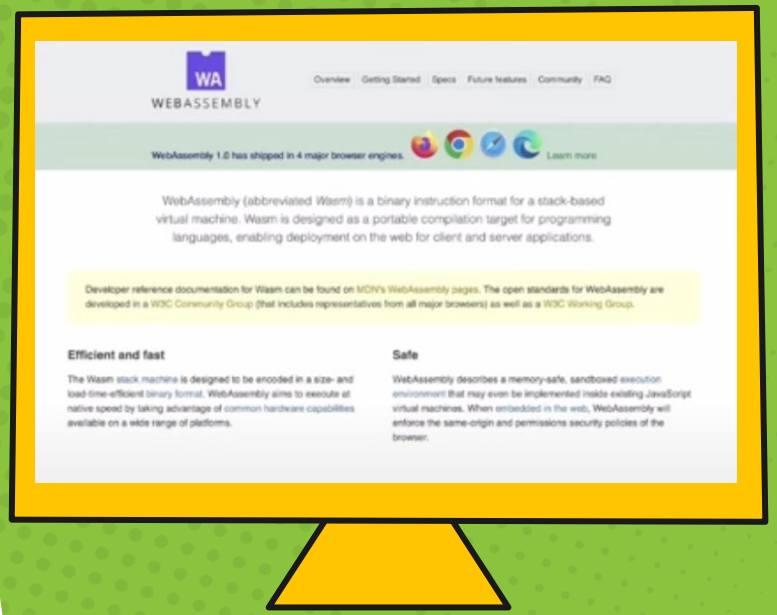
Cross-platform	★★★★★
Low overhead	★★★☆☆
Safety	★★★★★
Back. Compatibility	★★★★☆
Port existing code	★★★★☆



WEBASSEMBLY

WebAssembly (abbreviated *Wasm*) is a binary instruction format for a **stack-based virtual machine**.

Wasm is designed as a **portable** compilation target for programming languages, enabling deployment on the web for client and server applications.



SOUNDS SIMILAR...

It is a general-purpose programming language intended to **let programmers *write once, run anywhere***, meaning that compiled Java code can run on all platforms that support Java without the need to recompile.

Java applications are typically **compiled to bytecode that can run on any Java virtual machine** (JVM) regardless of the underlying computer architecture.

Java (programming language)

117 languages

Article Talk Read View source View history Tools

From Wikipedia, the free encyclopedia

Not to be confused with Java (software platform), JavaScript, Java, or Javanese language.
"Openframe" redirects here. For the ten-pin bowling term, see Open frame.

Java is a high-level, class-based, object-oriented programming language that is designed to have as few implementation dependencies as possible. It is a general-purpose programming language intended to let programmers *write once, run anywhere* (WORA),[16] meaning that compiled Java code can run on all platforms that support Java without the need to recompile.[17] Java applications are typically compiled to bytecode that can run on any Java virtual machine (JVM) regardless of the underlying computer architecture. The syntax of Java is similar to C and C++, but has fewer low-level facilities than either of them. The Java runtime provides dynamic capabilities (such as reflection and runtime code modification) that are typically not available in traditional compiled languages.

Java gained popularity shortly after its release, and has been a popular programming language since then.[18] Java was the third most popular programming language in

Java

Java

Paradigm Multi-paradigm: generic, object-oriented (class-based), functional,

WEBASSEMBLY

WebAssembly (Wasm) defines a **portable binary-code** format and a corresponding text format for executable programs as well as software interfaces for facilitating **communication** between such **programs** and their **host** environment.

The main goal of WebAssembly is to facilitate **high-performance** applications on web pages, but it is also designed to be usable in **non-web** environments. It is an open standard intended to support **any language** on **any operating system**, and in practice many of the most popular languages already have at least some level of support.

WEBASSEMBLY

Long time ago...

WASM CORE

- Rust
- C / C++
- Python
- Go
- Javascript / Typescript
- Kotlin / Java

At current moment...

WASIP2

- Rust (cargo component build)
- Python (componentize-py)
- Go (TinyGo)
- Javascript / Typescript (jco)
- ...

WEBASSEMBLY FORMAT

WebAssembly has two formats: binary and text formats.

Webassembly need runtime (Stack based virtual machine)

Webassembly has some limitations (only int and float, linear memory usage)

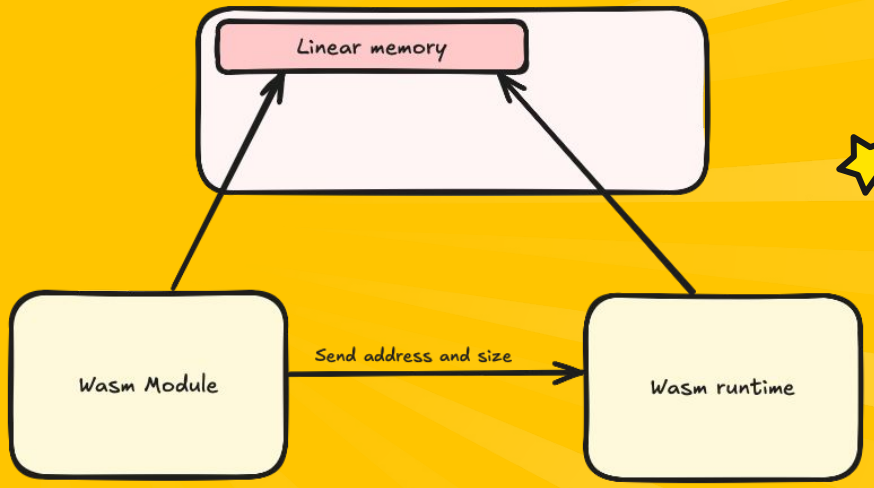
```
1  (module
2    (type $type1 (func (param i32) (result i32)))
3    (type $type2 (func (param i32))) ;; inner types in module
4    (import "env" "sqrt"             ;; Import the sqrt function
5      (func $sqrt (type $type1)))    ;; function signature is type1
6    (import "env" "print"            ;; Import the print function
7      (func $print (type $type1)))   ;; function signature is type2
8    (func $main          ;; define a function
9      (type $type1)      ;; function signature is type1
10     (local $var i32)   ;; define a local variable
11     loop               ;; Enter a separate stack space
12       block            ;; Enter a separate stack space
13         local.get 0     ;; Push the first param on stack
14         call $sqrt      ;; Call imported sqrt function
15         local.set $var  ;; Pop return value to $var
16       end
17       local.get $var  ;; Push return value on the stack
18       br_if 1         ;; Out of the execution of the function
19       local.get $var  ;; The param of the $print function
20       i32.const 1   ;; Push the element id of the function to be called
21       call_indirect (type $type2) ;; Signature of the function is type2
22       br 0        ;; Jump to line 11. Restart external loop
23     end)
24   (table 5 5 funcref)
25   (elem (i32.const 1) func $print)) ;; The first element points $print
```

WASM MODULE ABI

WASM module and runtime both has full access to shared memory

WASM module and runtime should trust to each other

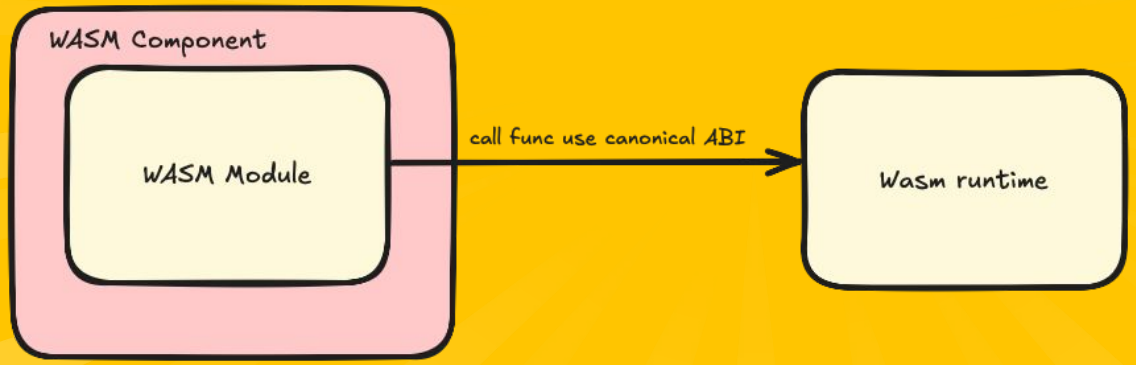
ABI can change depending on the language



DEMO TIME

COMPONENT

WASM Component and runtime work in shared nothing paradigm



COMPONENT

Components, as we know, are building blocks. Logically, components are **containers for modules or other components**, which express their interfaces and dependencies via WIT. Conceptually, components are **self-describing units** of code that interact only through **interfaces**. “self-describing” means components have interface descriptions inside. Physically, a component is a specially-formatted WebAssembly file. Internally, the component could include multiple traditional (“core”) WebAssembly modules, and sub-components, composed via their imports and exports

Canonical ABI

ABI - **A**pplication **B**inary **I**nterface

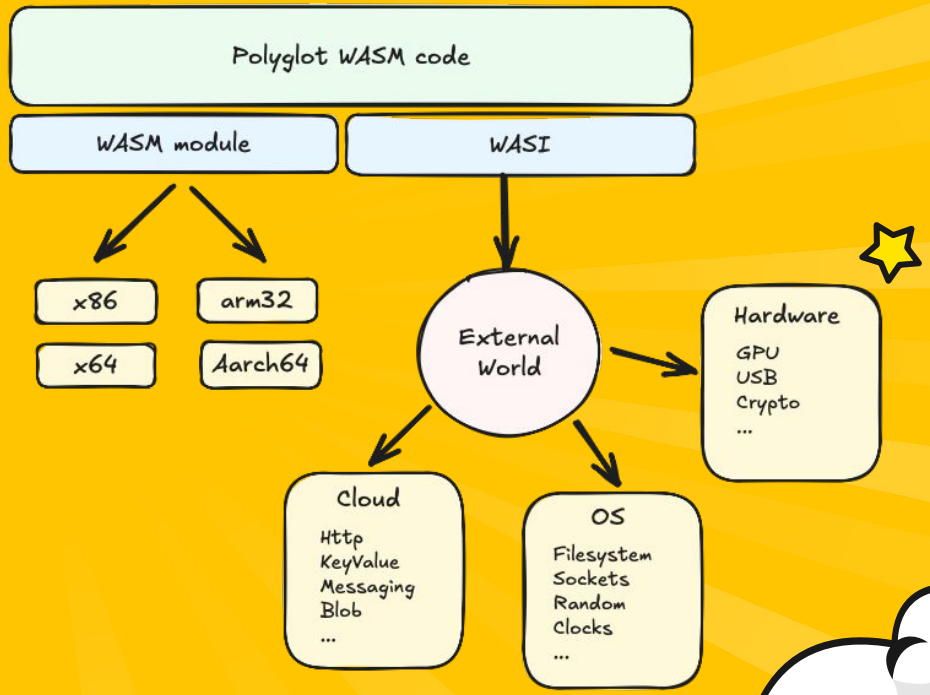
Is an **agreement** on how to transfer data in a binary format. All WASM components **must adhere** this agreement.

This makes WASM components compatible across different languages.

WASI

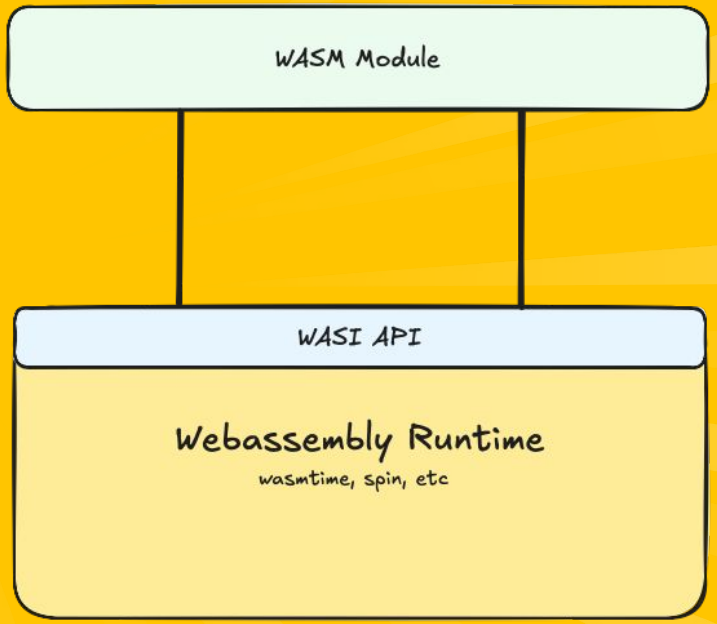
Webassembly started to be used outside the browser and it was necessary to determine how to run the code and interact with the outside world

WASI is **portable**, **modular**, **runtime independent** API for interaction with the outside world



RUNTIME

Webassembly started to be used outside the browser and it was necessary to determine how to run the code and interact with the outside world

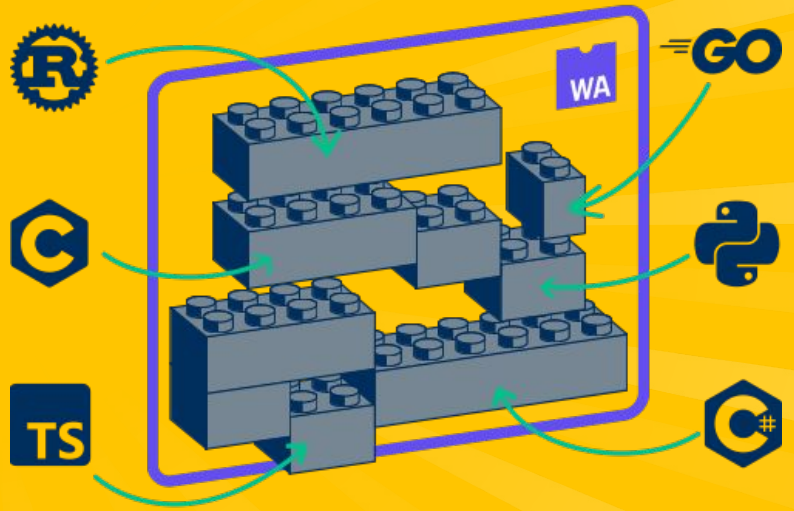


WEBASSEMBLY COMPONENTS

Containers for **modules**, or other **components** which express their **interfaces** and **dependencies** via **WIT**

Components interact with other **world** only using **interfaces** instead using **shared memory**

Components can contain multiple traditional **core module** or **sub-components**, composed via **interfaces**

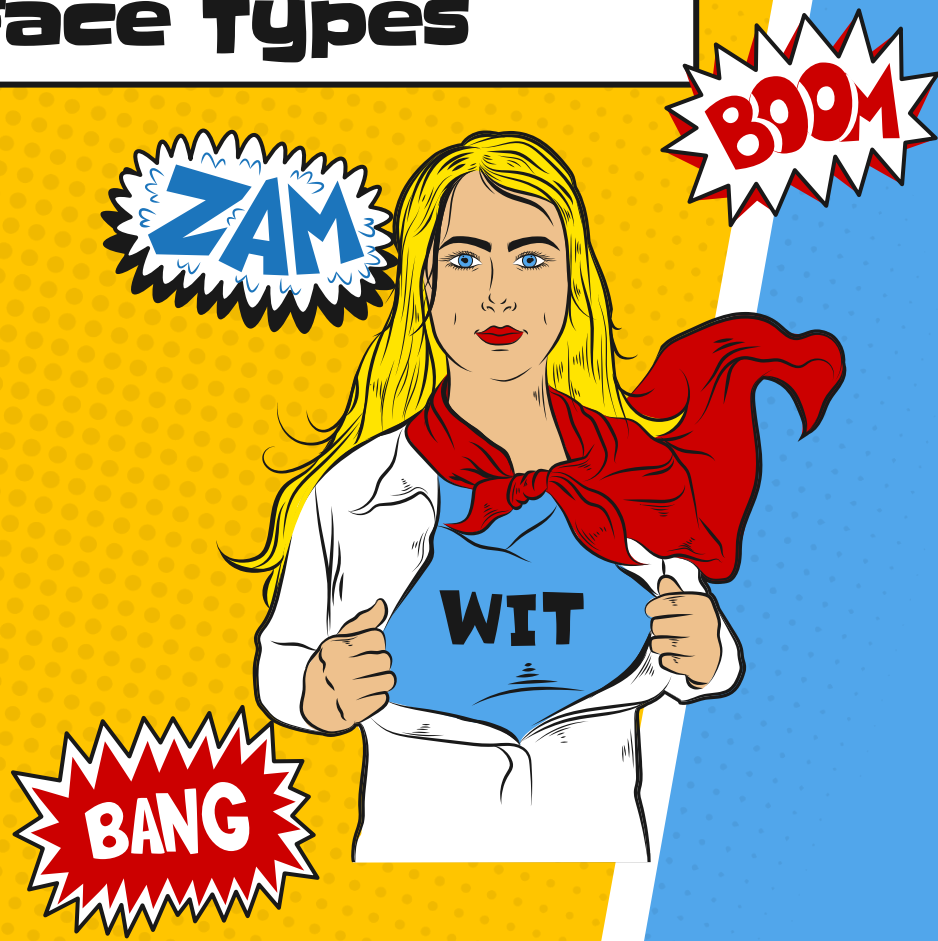


Webassembly Interface Types

WIT is used to describe WASM components **Interfaces** and **Worlds**

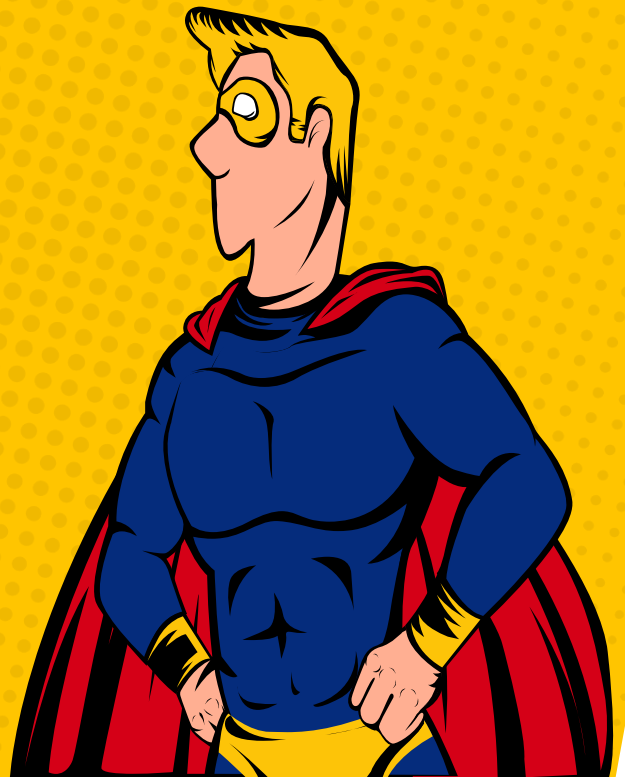
WIT is not a programming or scripting language. It's just an **IDL**.

WIT is a contract that is independent of programming languages.



WIT ECOSYSTEM

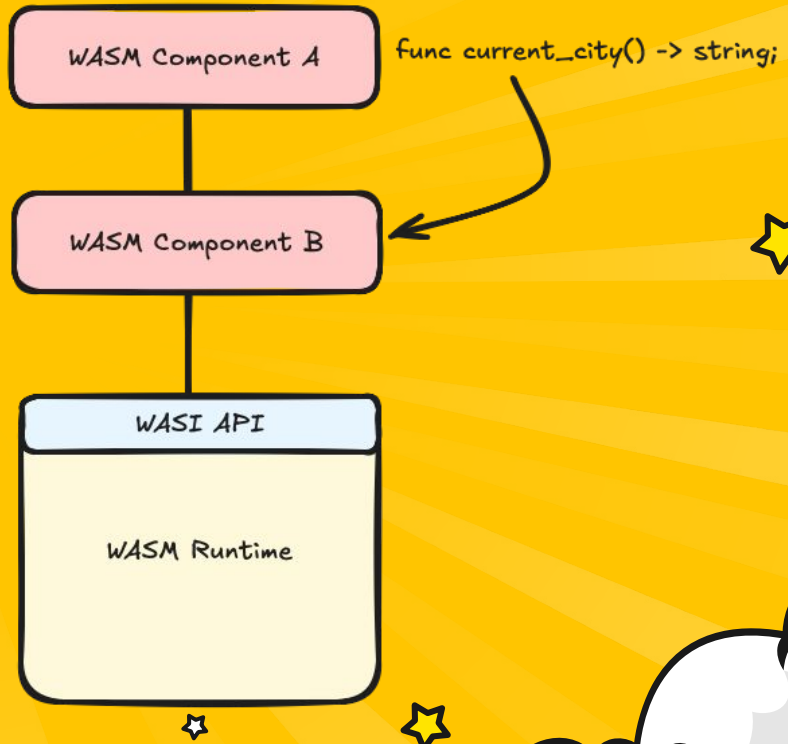
- WASI:IO
- WASI:CLOCKS
- WASI:CLI
- WASI:HTTP
- WASI:KEYVALUE
- WASI:SOCKETS
- WASI:CLOCK
- WASI:RANDOM
- WASI:LOGGING
- ...



COMPONENT COMPOSING

We can compose one web assembly component into another and extend functionality.

Each component can use different language



DEMO TIME


WARG
WAC
ECOSYSTEM
WIZER
WKG

Components workflow

Authoring *cargo component, spin new, etc*

Composing *wac plug, spin*

Running *wasmtime, spin, spinkube, Nginx UNIT*

Distributing *registry OCI / WARG, wa.dev*

WASIp3

- Integrates with source-language concurrency features. For example if we have wit file with async function we can write it using C# for example and import into Python. In additional we can use streaming.
- Make concurrent component composable
- Deperecate wasi:io
- Provide separate isolation levels to components

More info:
https://www.youtube.com/watch?v=hgWkiWeGEzk

Async WASI

new with 0.3

```
async Task<Model> load(string name) {
  var data = await blockingOperation(name);
  return new Model(data);
}
```

export

already in 0.2

```
load: func(name: string) -> result<model>;
```

import

new with 0.3

```
from 'my-component' import load

async def some_func():
  one = asyncio.create_task(load("input1.txt"))
  two = asyncio.create_task(load("input2.txt"))
  models = await asyncio.gather(one, two)
  ...
```

```
async function transform(readableStream) {
  return readableStream.pipeThrough(
    new DecompressionStream('gzip'));
}
```

JS

export

new with 0.3

```
transform: func(in: stream<u8>) -> stream<u8>;
```

import

```
import "mycomponent"

func someFunc(in: <-chan int) {
  for i := range mycomponent.transform(in) {
    ...
  }
}
```

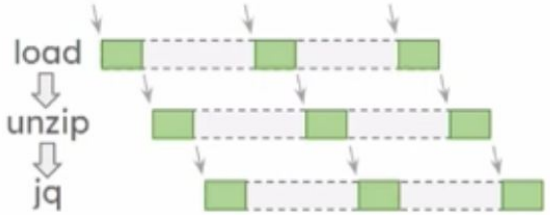
GO

COMPOSED ASYNC

Unix-pipeline-style composition:

```
interface tools {
  load: func(path: string) → stream<u8>;
  unzip: func(in: stream<u8>) → stream<u8>;
  jq: func(in: stream<u8>, query: string) → string;
}
```

```
// client.js
import { load, decompress, jq } from 'tools';
let first = await jq(unzip(load('input.txt.gz')), '.[0]');
console.log(first);
```



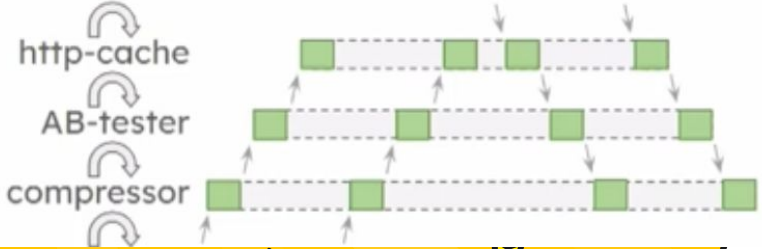
Service-chaining-style composition:

```
world chainable-handler {
  import wasi:http/handler;
  export wasi:http/handler;
}
```

target

- http-cache
- AB-tester
- compressor

```
// compound-handler.wac
let cache = new tools:http-cache { ... };
let ab = new tools:AB-tester { cache };
let compressor = new tools:compressor { ab };
export compressor[wasi:http/handler];
```



DEMO TIME

LINKS

- Demo examples - examples from demo from presentation
- octabot - host system for running plugins
- plugins for octabot - examples of working plugins
- skds for rust, go and typescript
- octa - builder like go-task, but with plugin system written using IPC communication