A Brief Introduction to WebAssembly

PLCT Intern Tech Report

Jiang Yuchen

2021.1.13

Contents of This Report

A. WebAssembly as a new web technology
   1. The First Glance
   2. Mechanism & Usage
   3. Why WebAssembly?

B. WebAssembly Micro Runtime
   1. The Product Contents
   2. The Code Base Structure
   3. Understanding WAMR’s Workflow
   4. Usage from the User Side

What is WebAssembly?

Definition

- 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.

WebAssembly 1.0 has shipped in 4 major browser engines.

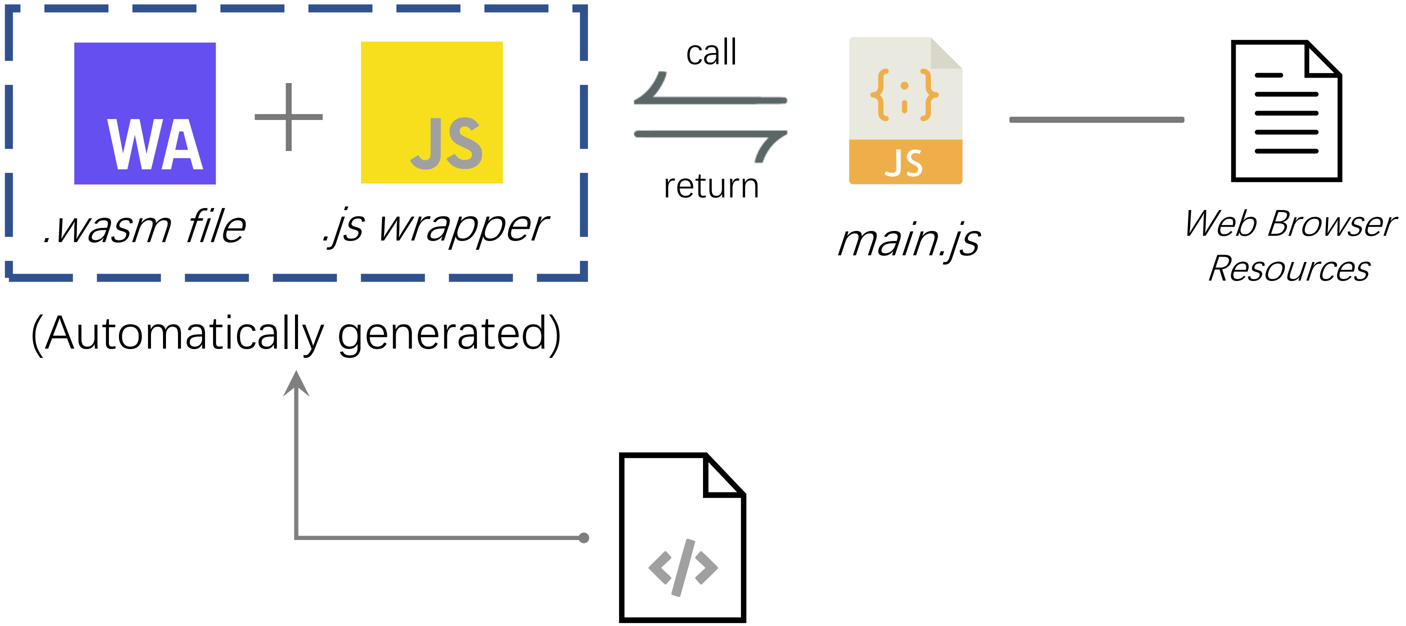
Reference: https://webassembly.org/

Significance

- Native speed on web platform with AOT compilation.
- Put more computation on user-side.
- Reuse legacy (old) code (e.g. in C++).

- A web-based virtual machine! Compile once, run everywhere.
- Generate from any language with a LLVM backend.
- Also supports non-web embeddings (like WAMR).

The Big Picture: How It Works



(Automatically generated)

C/C++, Rust, Python, Go, ...

Reference: https://dl.acm.org/doi/10.1145/3062341.3062363

Wasm Binary Code

C++	Binary *(.wasm file)*	Textual Equivalent

int factorial(int n) {
    if (n == 0) return 1;
    else return n * factorial(n-1);
}
	20 00 42 00 51 04 7e 42
	01 05 20 00 20 00 42 01
	7d 10 00 7e 0b
		get_local 0
		i64.const 0
		i64.eq
		if i64
		    i64.const 1
		else
		    get_local 0
		    get_local 0
		    i64.const 1
		    i64.sub
		    call 0
		    i64.mul
		end

How to use WebAssembly ?

C/C++: Use Emscription

```
$ emcc test.c -Os -s WASM=1 -s SIDE_MODULE=1 -o test.wasm
```

Reference: https://emscripten.org/

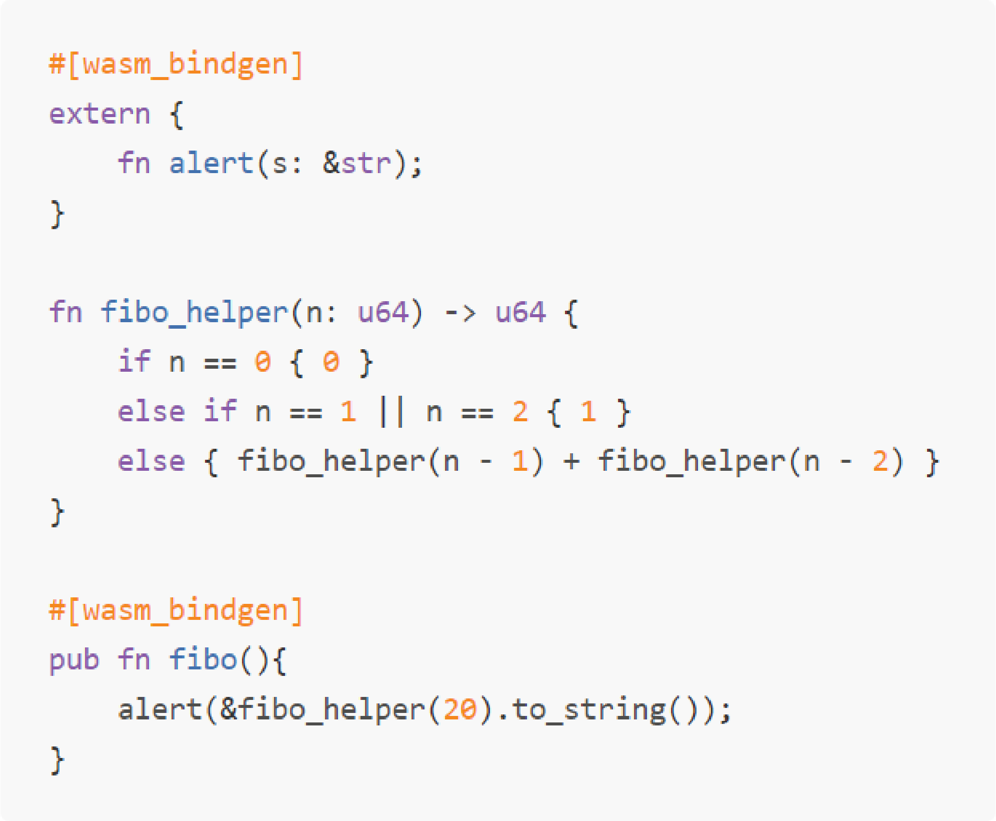
Rust: Use wasm-pack

- Follow the tutortial on this link.

Reference: https://rustwasm.github.io/

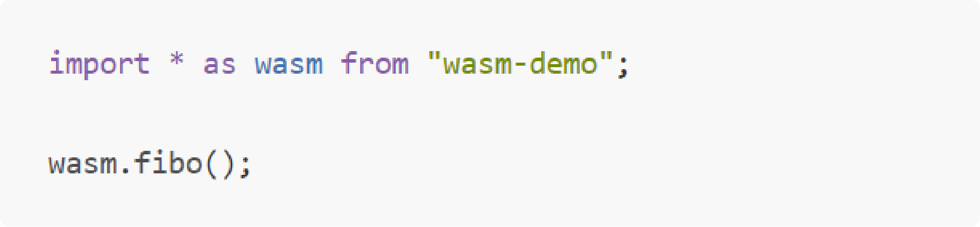
Find Fibonacci

Rust: wasm-demo.rs



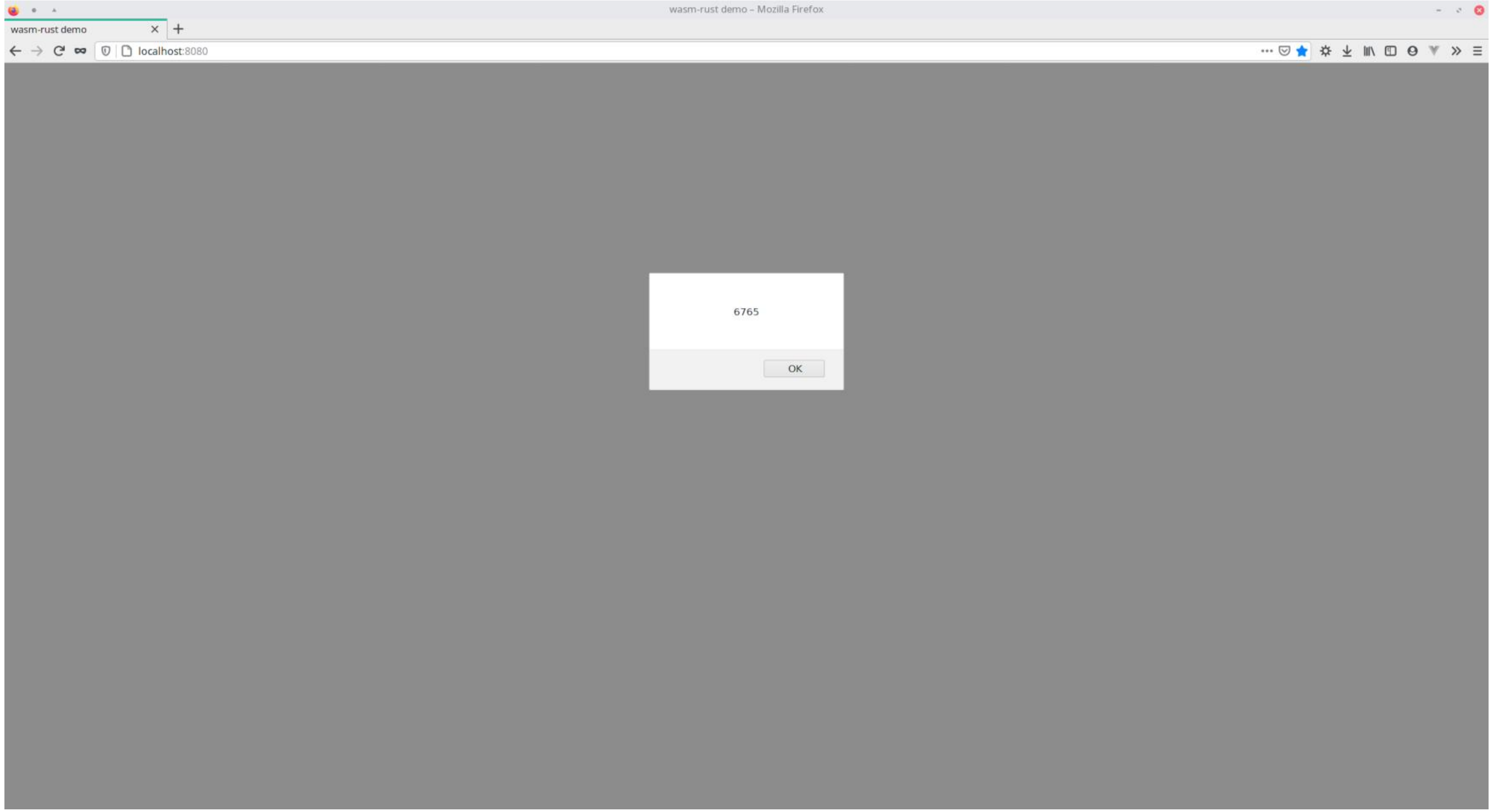
```rust
#[wasm_bindgen]
extern {
    fn alert(s: &str);
}

fn fibo_helper(n: u64) -> u64 {
    if n == 0 { 0 }
    else if n == 1 || n == 2 { 1 }
    else { fibo_helper(n - 1) + fibo_helper(n - 2) }
}

#[wasm_bindgen]
pub fn fibo(){
    alert(&fibo_helper(20).to_string());
}
```

JavaScript: index.js



```javascript
import * as wasm from "wasm-demo";

wasm.fibo();
```

Find Fibonacci: Output



```
6765
```

fibo_helper()

Just a wasm demo.

```
(func $_ZN17wasm_demo11fibo_helper17he7e525125906ff5eE (type $t5) (param $p0 i64) (result i64)
    (local $l1 i64)
    i64.const 1
    local.set $l1
    block $B0
      local.get $p0
      i64.const -1
      i64.add
      local.tee $p0
      i64.const 2
      i64.lt_u
      br_if $B0
      i64.const 0
      local.set $l1
      loop $L1
        local.get $p0
        call $_ZN17wasm_demo11fibo_helper17he7e525125906ff5eE
        local.get $l1
        i64.add
        local.set $l1
        local.get $p0
        i64.const -2
        i64.add
        local.tee $p0
        i64.const 1
        i64.gt_u
        br_if $L1
      end
      local.get $l1
      i64.const 1
      i64.add
      local.set $l1
    end
    local.get $l1)
```

Wasm in its Textual Format

C++	WAT (S-expression)

int factorial(int n) {
    if (n == 0)
        return 1;
    else
        return n * factorial(n-1);
}
	(func factorial
	    get_local 0
	    i64.const 0
	    i64.eq
	    if i64
	        i64.const 1
	    else
	        get_local 0
	        get_local 0
	        i64.const 1
	        i64.sub
	        call 0
	        i64.mul
	    end)

Wasm Operations

get_local 0	push local variable [0]
i64.const 0	push 0 to the stack top
i64.eq	pop, pop, comp, push
if i64	if branch with condition top
else	else branch
i64.sub	pop, pop, sub, push
call 0	call function [0] with stack top
i64.mul	pop, pop, mul, push
end	return with stack top value

Details & Reference: https://webassembly.github.io/spec/core/exec/index.html

Why WebAssembly?

What’s the use case?

Use Cases

- Web-based games
- Live 3D rendering
- Bitcoin and chain blocks (seriously?)

Clarifications

- Why not abandon JavaScript entirely?
  - UI design requires flexibility.
  - More rewriting, less output.
- Is it always faster than V8-powered JS scripts?
  - It depends.
- It’s a newborn technology, so…
  - Can’t directly access JS controlled memory.
  - No GC (not yet).
  - Immature when compared to old tools such as WebGL.
  - Still much slower than native C/C++/Rust.

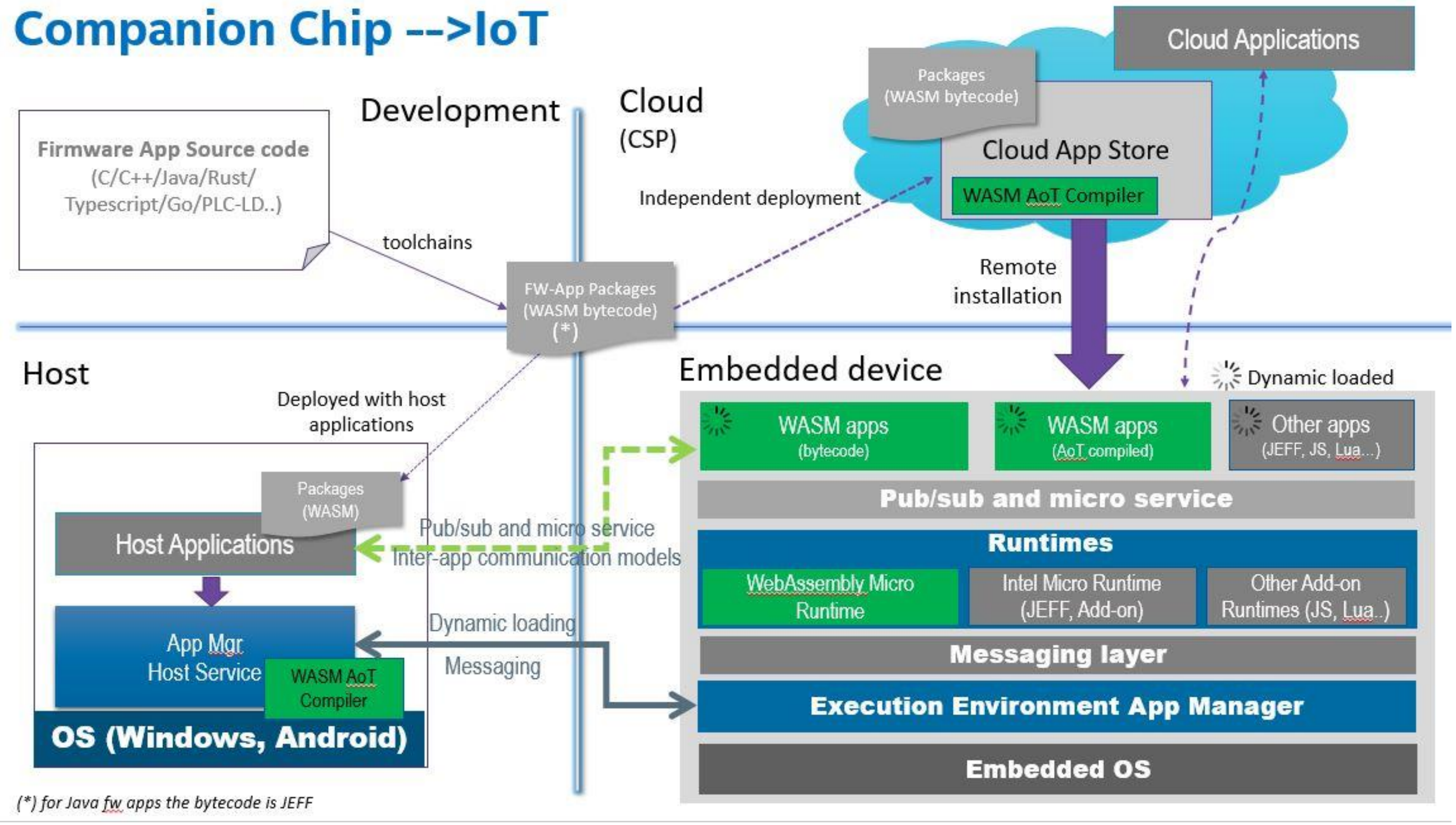
WebAssembly Micro Runtime

Source Code Analysis

Definition

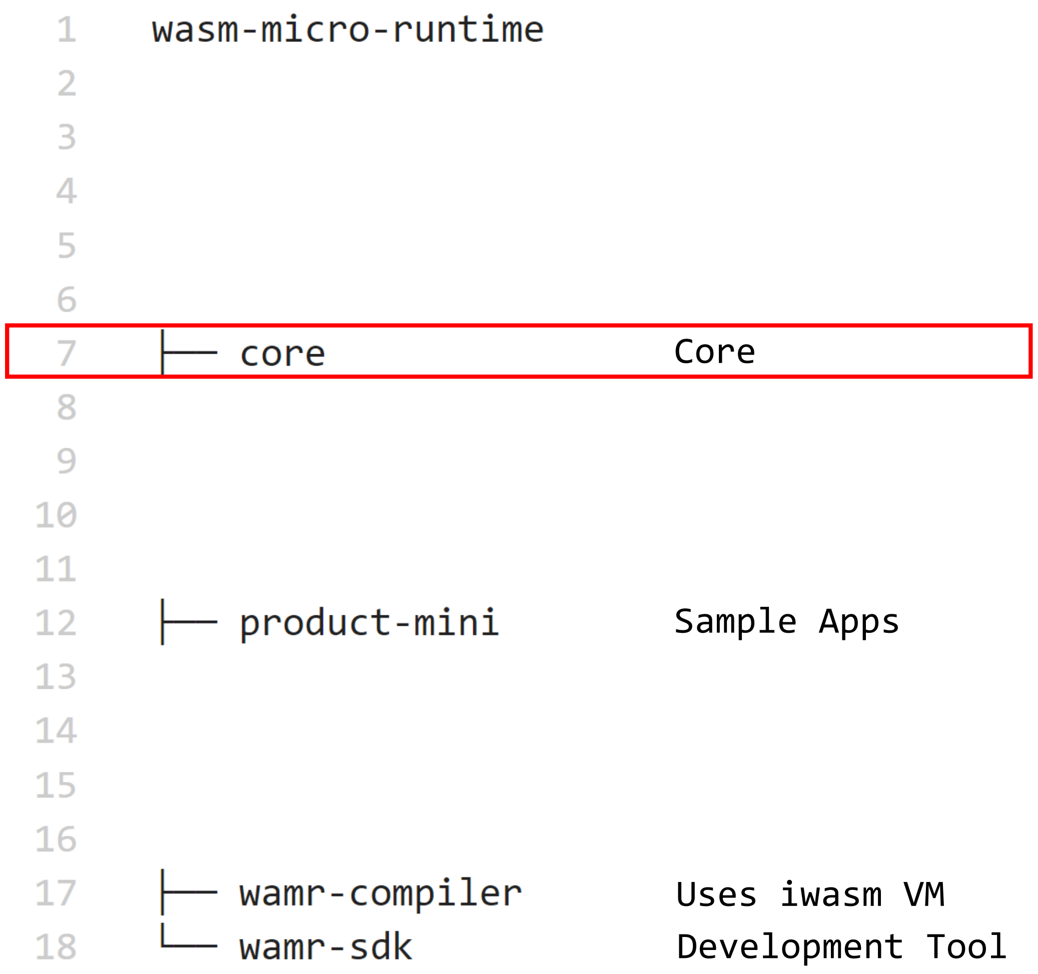
- WebAssembly Micro Runtime (WAMR) is a standalone WebAssembly runtime with a small footprint.
    1. “iwasm” VM Core (supports both AOT and JIT compilation)
    2. Application Framework and Supporting APIs
    3. Dynamic management of the WASM applications

The Big Picture



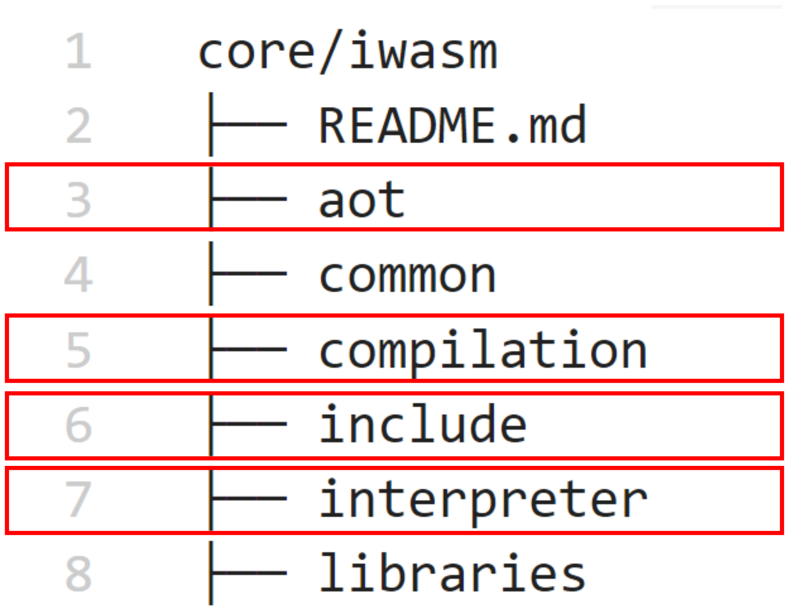
(*) for Java fw apps the bytecode is JEFF

Code Base Structure



```
wasm-micro-runtime





├── core                      Core




├── product-mini              Sample Apps




├── wamr-compiler             Uses iwasm VM
└── wamr-sdk                  Development Tool
```

core/iwasm/

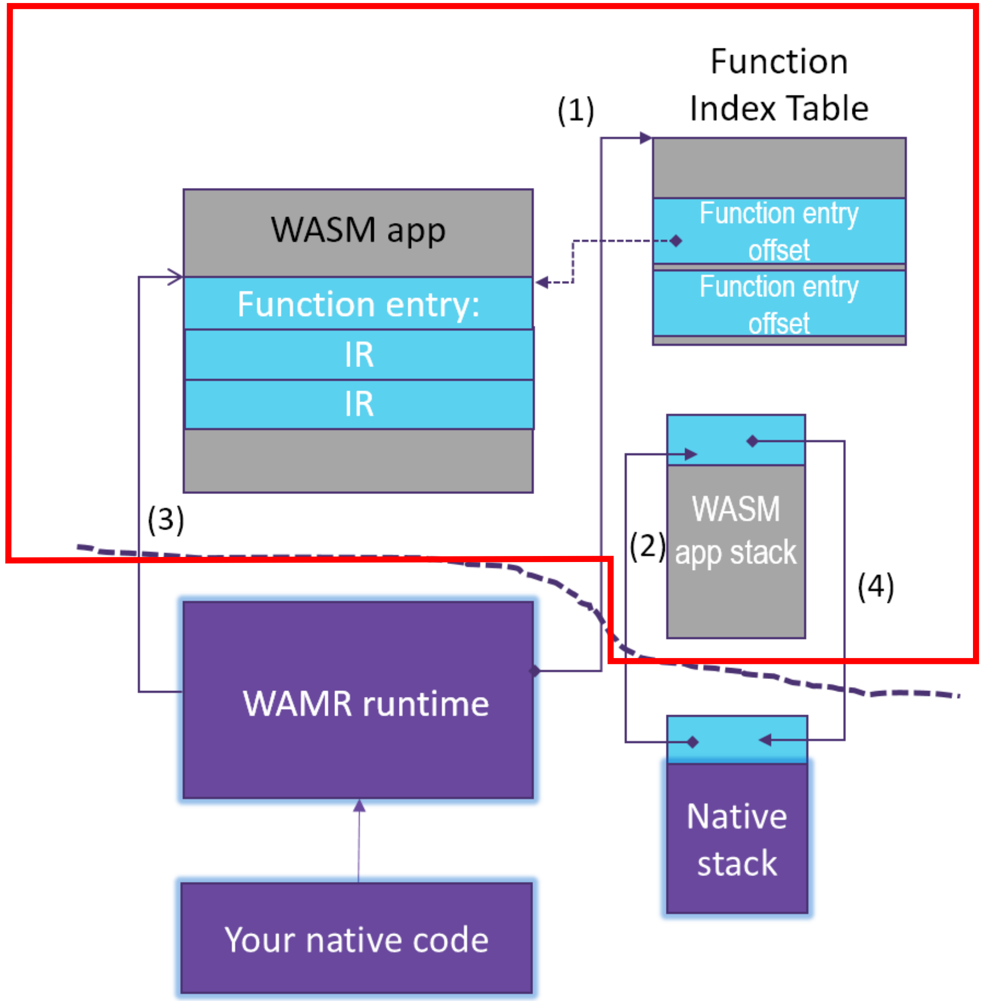


```
core/iwasm
├── README.md
├── aot
├── common
├── compilation
├── include
├── interpreter
├── libraries
```

core/iwasm/

- include: provides embedding APIs
- aot: AoT loader and runtime
- compilation: AoT compilation
- interpreter: A wasm interpreter with loader and runtime

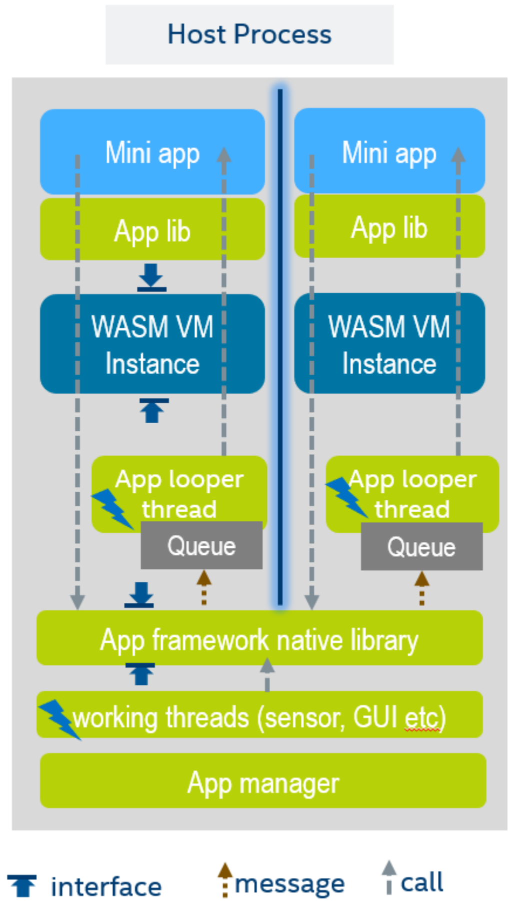
Dive into the workflow



WASM App Structure

```
(module
    (type   ... )
    (import ... )
    (func   ... )
    (table  ... )
    (mem  ... )
    (global ... )
    (export ... )
    (start  ... )
    (elem    ... )
    (data   ... )
)
```

WASM App Framework



Questions

Thank you.