**Engineer**

# WebAssembly

a portable binary code format that runs in a sandbox

## What it is

WebAssembly, usually shortened to Wasm, is a binary format for executable code. Its core specification calls it “a safe, portable, low-level code format designed for efficient execution and compact representation”, and describes it as a virtual instruction set: the instructions of an imagined processor, which a browser or another host first checks and then translates into machine code for the real one. The short form is a contraction, not an acronym, which is why the specification writes Wasm and not WASM.

A `.wasm` file is usually the output of a compiler for some other language. The main goal, in the specification’s own account, is fast applications on the Web, but the format makes no assumptions about the Web, so it can also run as a stand-alone virtual machine or inside other programs. The standard is layered to match: the core specification defines the instructions, the formats, validation and execution, and no particular environment. Separate documents define the JavaScript API that compiles, links and calls modules, and the Web API that loads them over the network.

The first version became a W3C Recommendation on 5 December 2019. Version 2.0, essentially finished three years before its successor, added instructions that work on vectors, bulk operations on memory, functions with more than one return value, and a simple first form of reference types. On 17 September 2025 the project announced version 3.0 as the standard now in force, and said it was already shipping in most major browsers. The same announcement names Java, OCaml, Scala, Kotlin, Scheme and Dart among the languages that now target Wasm through its garbage collector.

The vocabulary is small. The unit of code is a module: it holds functions, tables, memories and global variables, imports what it needs, each import named by a module name and an item name, and exports what it offers under names of its own. There are four number types, 32-bit and 64-bit integers and floating-point numbers, written `i32`, `i64`, `f32` and `f64`, plus a single 128-bit vector type and opaque references. Integers carry no sign of their own; each operation reads them as signed or unsigned, and a 32-bit integer also serves as a Boolean. Memory addresses use `i32` or, for a 64-bit memory, `i64`. Instructions take their operands from an implicit stack and leave their results on it, and control flow is structured: blocks, loops and conditionals nest, and a branch can only target one of them.

Byte-addressed data lives in a linear memory, one continuous, writable array of bytes that starts at a declared size and can grow while the program runs. A load or a store outside its current size does not reach anything else. It traps, which stops execution at once, and a trap cannot be handled inside Wasm, only reported to the host around it. The example below uses the text format the specification defines for people to read.

Use the concept in a decision

Put “What is WebAssembly?” to work in a real decision. We can help relate the concept to the system you are building or buying. Interfaces that load quickly, work on a phone and behave the same in every browser your customers happen to use.

**Explore our services** Get in touch →

Test the assumptions behind the term

Use the explanation and linked work to frame questions for your project. We can examine the assumptions with you before a tool is chosen. Directed full-stack GIS map development — full-stack mapping, from database to browser

**Directed full-stack GIS map development** Get in touch →

## Why it matters

Speed is the first reason. The design goals ask for performance close to native code and a format that a single fast pass can decode, validate and compile, whether the compiler runs just in time or ahead of time. The binary is meant to be more compact than both text and native machine code, and it is streamable: work can begin before the last byte has arrived. The binary format even keeps each function’s type declaration in a different section from its body, so that the bodies can be compiled in parallel and while they stream in. A team that already owns a well-tested library in another language can compile it for the browser instead of rewriting it in JavaScript.

Portability is the second. The same module runs on any modern processor, desktop or mobile, and in any host that embeds an engine: a browser, a stand-alone runtime or another program. The specification favours no particular language, programming model or object model, so the format is a common target rather than one language's private runtime.

Containment is the third. A module has no ambient access to the computer it runs on. Files, the network, the page and the clock are out of reach unless the host hands it a function for them as an import, so the host decides exactly what the code may touch. On the Web, the Web API notes that Wasm can run wherever JavaScript can and has essentially the same threat model, and cannot expose more than JavaScript already could. Version 3.0 widened what fits inside that boundary: memories that can be addressed with 64 bits, limited on the web to 16 gigabytes, several memories in one module, a garbage collector for languages that expect one, and exceptions that no longer have to escape to the host, which the announcement calls neither portable nor efficient.

## What it looks like in practice

Here is a module in the text format, with a single exported function that adds two 32-bit integers:

```
(module
  (func (export "add") (param i32 i32) (result i32)
    local.get 0
    local.get 1
    i32.add))
```

The function's type says it takes two `i32` values and returns one. The body is three instructions on the stack: `local.get 0` pushes the first parameter, `local.get 1` pushes the second, and `i32.add` pops both and pushes their sum, which is left on the stack as the result. `(export "add")` gives the function the name the outside world sees. Converted to the binary format, the file starts with the four bytes `0x00 0x61 0x73 0x6D`, a zero followed by the letters asm, and then the binary format's version, which is still 1.

In a browser, the JavaScript API loads it:

```
const url = "/add.wasm";
const { instance } = await WebAssembly.instantiateStreaming(fetch(url));
instance.exports.add(2, 3); // 5
instance.exports.add(2147483647, 1); // -2147483648
```

`instantiateStreaming` compiles the module while the response is still arriving and resolves to the compiled module and an instance of it. The instance's exports are ordinary JavaScript functions. The second call shows what the types mean: integer addition in Wasm wraps around modulo 2 to the power of 32, and the JavaScript API reads an `i32` result as signed, so one more than the largest positive value comes back as the most negative one. Nothing is thrown. A trap would be, though: if a function reads outside its memory, the call ends in a `WebAssembly.RuntimeError`.

A module that needs something from its host says so. With `(import "env" "log" (func (param i32)))` in the module, instantiation needs a second argument, `{ env: { log: console.log } }`, and the module can call `log` like any of its own functions. Leave the second argument out and instantiation fails with a `TypeError`; pass an `env` object without a callable `log` and it fails with a `WebAssembly.LinkError`, in both cases before any of the module's code has run. Imports and exports are the whole boundary, with declared types that include numbers and references; an address in the module's memory is an `i32`, or an `i64` for a 64-bit memory.

A portfolio entry on building a native macOS client in Swift 6 describes the other way of reusing code from another language. In that work, carried out in August 2026, a Rust core was linked into a Swift application as a static archive and reached through its C interface, which the entry treats as the contract. A thin wrapper owns every pointer and every string that crosses the boundary, and everything the interface returns has an ownership rule, where a mistake means a leak or a crash with no compiler warning. No Wasm is involved there, which is what makes the comparison useful: the same problem of a narrow boundary between two languages, solved with native linking instead of an import list and a sandboxed memory.

## Where it goes wrong

**The promise rejects before anything compiles.** A missing `Content-Type`, or any value other than `application/wasm` with case ignored, fails with a `TypeError`, and the specification notes that extra parameters are not allowed, not even an empty `application/wasm;`. A status that is not ok fails the same way, and so does a

response from another origin that CORS does not make same-origin. Fix the server's media type and headers. `WebAssembly.instantiate` on bytes you have fetched yourself skips the media type check but loses the streaming compilation.

**A view of memory goes dead.** JavaScript sees a module's memory through `memory.buffer` and the arrays built on it. For unshared memory, this buffer is fixed-length by default. When the memory grows, the JavaScript API detaches that fixed-length buffer and replaces it with a new one, so arrays built on the old buffer are cut off from the memory. This happens after `memory.grow()` called from JavaScript and also after a `memory.grow` instruction inside the module, during a call that looks harmless from outside. Create views from `memory.buffer` again after any call that may grow the memory, rather than keeping one for the lifetime of the page.

**Expecting Wasm to reach the page by itself.** A module has no ambient access to anything, so every effect on the page, the network or storage goes through a function JavaScript passes in as an import. Types bite at the same boundary: an `i64` crosses as a `BigInt`, not a number, and a function with a `v128` parameter or result throws a `TypeError` each time JavaScript calls it. Design the import and export list as an interface, with plain numbers on it, and keep vectors on the Wasm side.

**Taking the sandbox for a safe program.** The sandbox protects the host from the module. It does not protect the module from itself. The specification states that it cannot stop an unsafe language compiled to Wasm from corrupting its own memory layout inside linear memory: a buffer overflow in compiled code overwrites the module's own data, and only an access past the end of the whole memory traps. Keep the source language's own checks and tests, and treat data the module receives from outside as untrusted input.

**Shipping the binary without protecting it.** The registration of `application/wasm` says the format includes no integrity or privacy protection, and that where such protection is needed it must come from outside, through HTTPS for example. As with JavaScript, some ways of loading and running it are limited by Content Security Policy or by the same-origin policy. And because Wasm becomes machine code on real hardware, the core specification warns that it is potentially open to side-channel attacks at the hardware level. Serve modules over HTTPS, and guard who may change a module as closely as who may change a script.

# Further reading

- WebAssembly Specification — the core specification for version 3.0: structure, validation, execution and the binary and text formats.
- Introduction — the definition, the design goals, the scope of each document and the security considerations.
- WebAssembly JavaScript Interface — compiling and instantiating from JavaScript, how values cross, `Memory.grow` and the three error classes.
- WebAssembly Web API — `instantiateStreaming`, the checks on the response and the `application/wasm` registration.
- Wasm 3.0 Completed — the announcement of 17 September 2025 and what version 3.0 adds.
- WebAssembly Core Specification 1.0 — the W3C Recommendation of 5 December 2019.

Glossary

Choose for your actual constraints

Describe the workflow, the existing interfaces and the users who need the change. We can define a testable delivery and a handover that lets your team maintain it.

**See what it costs** Get in touch →

## Make the terminology practical

Ask how “What is WebAssembly?” applies to your organisation. A concrete example from your current system is enough to start the discussion.

**Explore our services**

welcome@engineer.company +45 50 10 61 84 Copenhagen, Denmark

Take this page with you

Contact card

**Carefully Crafted With Love by D.**

**CVR 45969703 · Tagensvej 105C, 3. th, 2200 København N**

**Copyright © 2025 – present · Engineer ApS · All rights reserved.**

**Printed from https://engineer.company/glossary/webassembly/**