Engineer

WebAssembly

formato binario portátil de código con ejecución aislada

Qué es

WebAssembly, que suele abreviarse como Wasm, es un formato binario para código ejecutable. Su especificación principal lo llama «un formato de código seguro, portable y de bajo nivel, diseñado para una ejecución eficiente y una representación compacta» y lo describe como un conjunto de instrucciones virtual: las instrucciones de un procesador imaginario, que un navegador u otro entorno anfitrión primero comprueba y después traduce a código máquina para el procesador real. La forma corta es una contracción, no una sigla; por eso la especificación escribe Wasm y no WASM.

Un archivo `.wasm` suele ser el resultado de un compilador para otro lenguaje. El objetivo principal, según la propia especificación, es que las aplicaciones web sean rápidas, pero el formato no presupone la web: también puede ejecutarse como una máquina virtual independiente o dentro de otros programas. El estándar se divide en capas para reflejarlo: la especificación principal define las instrucciones, los formatos, la validación y la ejecución, sin un entorno concreto. Otros documentos definen la API de JavaScript que compila, enlaza e invoca los módulos, y la API web que los carga por la red.

La primera versión se convirtió en Recomendación del W3C el 5 de diciembre de 2019. La versión 2.0, prácticamente terminada tres años antes de su sucesora, añadió instrucciones vectoriales, operaciones masivas sobre la memoria, funciones con más de un valor de retorno y una primera forma sencilla de tipos de referencia. El 17 de septiembre de 2025, el proyecto anunció la versión 3.0 como estándar vigente e indicó que ya se distribuía en la mayoría de los principales navegadores. El mismo anuncio cita Java, OCaml, Scala, Kotlin, Scheme y Dart entre los lenguajes que ya generan Wasm mediante su recolector de basura.

El vocabulario es reducido. La unidad de código es un módulo: contiene funciones, tablas, memorias y variables globales; importa lo que necesita, con un nombre de módulo y otro de elemento para cada importación, y exporta lo que ofrece bajo sus propios nombres. Hay cuatro tipos numéricos, enteros y números en coma flotante de 32 y 64 bits, escritos `i32`, `i64`, `f32` y `f64`, además de un único tipo vectorial de 128 bits y referencias opacas. Los enteros no tienen signo por sí mismos: cada operación los interpreta con o sin signo, y un entero de 32 bits también sirve como booleano. Las direcciones de memoria usan `i32` o, para una memoria de 64 bits, `i64`. Las instrucciones toman sus operandos de una pila implícita y dejan en ella los resultados. El flujo de control es estructurado: bloques, bucles y condicionales se anidan, y un salto solo puede dirigirse a uno de ellos.

Los datos direccionados por bytes residen en una memoria lineal, una matriz continua de bytes que admite escritura, parte de un tamaño declarado y puede crecer durante la ejecución. Una lectura o escritura fuera de su tamaño actual no alcanza nada más: provoca una trampa que detiene inmediatamente la ejecución. Una trampa no puede manejarse dentro de Wasm, solo notificarse al entorno anfitrión. El ejemplo siguiente usa el formato textual que la especificación define para que lo lean personas.

Utilice el concepto en una decisión

Aplique «¿Qué es WebAssembly?» a una decisión real. Podemos ayudarle a relacionar el concepto con el sistema que está construyendo o comprando. Interfaces que cargan rápido, funcionan en el teléfono y se comportan igual en los navegadores que utilizan sus clientes.

Explore nuestros servicios Póngase en contacto →

Compruebe los supuestos detrás del término

Utilice la explicación y los trabajos enlazados para formular preguntas sobre su proyecto. Podemos examinar los supuestos con usted antes de elegir una herramienta. Dirección del desarrollo integral de mapas GIS — cartografía integral, de la base de datos al navegador

Dirección del desarrollo integral de mapas GIS Póngase en contacto →

Por qué importa

La primera razón es la velocidad. Los objetivos de diseño piden un rendimiento cercano al código nativo y un formato que una sola pasada rápida pueda decodificar, validar y compilar, tanto si el compilador trabaja durante la ejecución como si lo hace de antemano. El binario pretende ser más compacto que el texto y que el código máquina nativo, y admite procesamiento en flujo: el trabajo puede empezar antes de que llegue el último byte. Incluso mantiene la declaración de tipo de cada función en una sección distinta de su cuerpo para

que los cuerpos puedan compilarse en paralelo y mientras llegan. Un equipo que ya cuenta con una biblioteca bien probada en otro lenguaje puede compilarla para el navegador en lugar de reescribirla en JavaScript.

La segunda es la portabilidad. El mismo módulo se ejecuta en cualquier procesador moderno, de escritorio o móvil, y en cualquier anfitrión que incorpore un motor: un navegador, un entorno de ejecución independiente u otro programa. La especificación no favorece ningún lenguaje, modelo de programación ni modelo de objetos concreto. El formato es, por tanto, un destino común de compilación y no el entorno de ejecución privado de un lenguaje.

La tercera es el aislamiento. Un módulo no dispone de acceso implícito al equipo donde se ejecuta. Los archivos, la red, la página y el reloj quedan fuera de su alcance salvo que el anfitrión le entregue una función para ellos como importación; el anfitrión decide exactamente qué puede tocar el código. En la web, la API web señala que Wasm puede ejecutarse donde se ejecute JavaScript, tiene esencialmente el mismo modelo de amenazas y no puede exponer más de lo que JavaScript ya podría. La versión 3.0 amplió lo que cabe dentro de ese límite: memorias direccionables con 64 bits, limitadas en la web a 16 gigabytes; varias memorias en un módulo; un recolector de basura para lenguajes que cuentan con él; y excepciones que ya no tienen que salir al anfitrión. El anuncio califica esa salida al anfitrión de no portátil e ineficiente.

Cómo se aplica en la práctica

Este módulo en formato textual exporta una sola función que suma dos enteros de 32 bits:

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

El tipo de la función indica que recibe dos valores `i32` y devuelve uno. El cuerpo son tres instrucciones sobre la pila: `local.get 0` apila el primer parámetro, `local.get 1` el segundo, e `i32.add` retira ambos y apila su suma, que queda como resultado. `(export "add")` da a la función el nombre que ve el exterior.

Convertido al formato binario, el archivo empieza por los cuatro bytes `0x00 0x61 0x73 0x6D` , un cero seguido de las letras asm, y después por la versión del formato binario, que sigue siendo 1.

En un navegador, la API de JavaScript lo carga así:

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

`instantiateStreaming` compila el módulo mientras todavía llega la respuesta y resuelve la promesa con el módulo compilado y una instancia. Las exportaciones de la instancia son funciones ordinarias de JavaScript. La segunda llamada muestra qué significan los tipos: la suma de enteros en Wasm se calcula módulo 2 elevado a 32, y la API de JavaScript interpreta un resultado `i32` con signo. Por eso, una unidad más que el mayor valor positivo vuelve como el valor más negativo. No se lanza ninguna excepción. Una trampa sí la provocaría: si una función lee fuera de su memoria, la llamada termina con un `WebAssembly.RuntimeError` .

Un módulo declara lo que necesita de su anfitrión. Si contiene `(import "env" "log" (func (param i32)))` , la instanciación necesita un segundo argumento, `{ env: { log: console.log } }` , y el módulo puede llamar a `log` como a cualquiera de sus propias funciones. Si se omite el segundo argumento, la instanciación falla con un `TypeError` ; si se pasa un objeto `env` sin una `log` invocable, falla con un `WebAssembly.LinkError` . En ambos casos ocurre antes de que se ejecute código del módulo. Las importaciones y exportaciones constituyen toda la frontera, con tipos declarados que incluyen números y referencias; una dirección en la memoria del módulo es un `i32` , o un `i64` para una memoria de 64 bits.

Una entrada del portafolio sobre un cliente nativo de macOS en Swift 6 describe la otra forma de reutilizar código de otro lenguaje. En ese trabajo, realizado en agosto de 2026, un núcleo de Rust se enlazó a una aplicación Swift como archivo estático y se utilizó a través de su interfaz C, que la entrada trata como el contrato. Una envoltura fina posee cada puntero y cada cadena que cruza la frontera; todo lo que devuelve la interfaz tiene una regla de propiedad. Equivocarse causa una fuga o un cierre inesperado sin aviso del compilador. Ahí

no interviene Wasm, y eso hace útil la comparación: el mismo problema de una frontera estrecha entre dos lenguajes, resuelto mediante enlace nativo en lugar de una lista de importaciones y una memoria aislada.

Dónde surgen los problemas

La promesa se rechaza antes de compilar nada. Si falta `Content-Type`, o su valor es distinto de `application/wasm` sin distinguir mayúsculas, se produce un `TypeError`. La especificación señala que no se admiten parámetros adicionales, ni siquiera un parámetro vacío como en `application/wasm;`. Un estado no satisfactorio falla igual, y también una respuesta de otro origen cuyo acceso no esté autorizado por CORS. Corrija el tipo de medio y las cabeceras del servidor. `WebAssembly.instantiate` sobre bytes obtenidos por su cuenta evita la comprobación del tipo de medio, pero pierde la compilación en flujo.

Una vista de memoria deja de funcionar. JavaScript ve la memoria de un módulo mediante `memory.buffer` y las matrices construidas sobre él. Para memoria no compartida, ese búfer tiene longitud fija de forma predeterminada. Cuando la memoria crece, la API de JavaScript desconecta ese búfer de longitud fija y lo sustituye por otro; las matrices construidas sobre el antiguo quedan desconectadas de la memoria. Ocurre tras `memory.grow()` desde JavaScript y también tras una instrucción `memory.grow` dentro del módulo, durante una llamada que parece inocua desde fuera. Vuelva a crear las vistas desde `memory.buffer` después de cualquier llamada que pueda ampliar la memoria, en lugar de conservar una durante toda la vida de la página.

Esperar que Wasm acceda por sí solo a la página. Un módulo no tiene acceso implícito a nada: cada efecto sobre la página, la red o el almacenamiento pasa por una función que JavaScript le entrega como importación. Los tipos también importan en esa frontera: un `i64` cruza como `BigInt`, no como un valor de tipo Number, y una función con un parámetro o resultado `v128` lanza un `TypeError` cada vez que JavaScript la llama. Diseñe la lista de importaciones y exportaciones como una interfaz con números sencillos y mantenga los vectores del lado de Wasm.

Confundir un entorno aislado con un programa seguro. El aislamiento protege al anfitrión frente al módulo. No protege al módulo frente a sí mismo. La especificación afirma que el aislamiento no puede impedir que un lenguaje inseguro compilado a Wasm corrompa su propia distribución de memoria dentro de la memoria lineal: un desbordamiento de búfer en código compilado

sobrescribe los datos del módulo, y solo un acceso más allá del final de toda la memoria provoca una trampa. Mantenga las comprobaciones y pruebas del lenguaje de origen y trate como no fiables los datos que el módulo recibe del exterior.

Distribuir el binario sin protegerlo. El registro de `application/wasm` indica que el formato no incluye protección de integridad ni de privacidad; cuando se necesitan, deben venir de fuera, por ejemplo mediante HTTPS. Como ocurre con JavaScript, algunas formas de cargarlo y ejecutarlo están limitadas por Content Security Policy o por la política del mismo origen. Y puesto que Wasm se convierte en código máquina sobre hardware real, la especificación principal advierte de que puede ser vulnerable a ataques de canal lateral en el hardware. Sirva los módulos mediante HTTPS y controle quién puede cambiar un módulo con el mismo cuidado que quién puede cambiar un script.

Lecturas adicionales

- WebAssembly Specification — especificación principal de la versión 3.0: estructura, validación, ejecución y formatos binario y textual.
- Introduction — definición, objetivos de diseño, alcance de cada documento y consideraciones de seguridad.
- WebAssembly JavaScript Interface — compilación e instanciación desde JavaScript, paso de valores, `Memory.grow` y las tres clases de error.
- WebAssembly Web API — `instantiateStreaming`, comprobaciones de la respuesta y registro de `application/wasm`.
- Wasm 3.0 Completed — anuncio del 17 de septiembre de 2025 y novedades de la versión 3.0.
- WebAssembly Core Specification 1.0 — Recomendación del W3C del 5 de diciembre de 2019.

Glosario

Elija según sus restricciones reales

Describa el flujo de trabajo, las interfaces existentes y los usuarios que necesitan el cambio. Podemos definir una entrega comprobable y un traspaso que permita a su equipo mantener lo entregado.

Vea cuánto cuesta Póngase en contacto →

Haga práctica la terminología

Pregunte cómo se aplica «¿Qué es WebAssembly?» a su organización. Basta un ejemplo concreto de su sistema actual para iniciar la conversación.

Explore nuestros servicios

welcome@engineer.company +45 50 10 61 84 Copenhague, Dinamarca

Llévese esta página

Tarjeta de contacto

Creado con esmero y cariño por D.

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

Derechos de autor © 2025 – actualidad · Engineer ApS · Todos los derechos reservados.

Impreso desde https://engineer.company/es/glossary/webassembly/