Engineer

WebAssembly

portables Binärformat für Code, der in einer Sandbox läuft

Was es ist

WebAssembly, meist Wasm abgekürzt, ist ein Binärformat für ausführbaren Code. Die Kernspezifikation nennt es ein sicheres, portables Codeformat auf niedriger Ebene, ausgelegt auf effiziente Ausführung und kompakte Darstellung. Sie beschreibt es als virtuellen Befehlssatz: die Anweisungen eines gedachten Prozessors, die ein Browser oder ein anderer Host zunächst prüft und dann in Maschinencode für den tatsächlichen Prozessor übersetzt. Die Kurzform ist ein zusammengezogenes Wort, kein Akronym. Deshalb schreibt die Spezifikation Wasm und nicht WASM.

Eine `.wasm`-Datei ist gewöhnlich das Ergebnis eines Compilers für eine andere Sprache. Das Hauptziel sind laut Spezifikation schnelle Anwendungen im Web. Das Format setzt jedoch nichts voraus, was nur im Web vorhanden wäre, und kann daher auch als eigenständige virtuelle Maschine oder innerhalb anderer Programme laufen. Dazu passt der Aufbau der Standards: Die Kernspezifikation definiert Anweisungen, Formate, Validierung und Ausführung, aber keine bestimmte Umgebung. Eigene Dokumente beschreiben die JavaScript-API, die Module kompiliert, verknüpft und aufruft, sowie die Web-API, die sie über das Netz lädt.

Die erste Fassung wurde am 5. Dezember 2019 eine W3C Recommendation. Version 2.0 war im Wesentlichen schon drei Jahre vor ihrem Nachfolger fertig. Sie ergänzte Vektoranweisungen, Sammeloperationen auf dem Speicher, Funktionen mit mehreren Rückgabewerten und eine erste einfache Form von Referenztypen. Am 17. September 2025 erklärte das Projekt Version 3.0 zum nun geltenden Standard und teilte mit, dass die meisten großen Browser sie bereits auslieferten. Dieselbe Ankündigung nennt Java, OCaml, Scala, Kotlin, Scheme und Dart als Sprachen, die dank der Speicherbereinigung inzwischen Wasm als Ziel unterstützen.

Der Wortschatz ist überschaubar. Die Einheit des Codes ist ein Modul. Es enthält Funktionen, Tabellen, Speicher und globale Variablen, importiert das Benötigte jeweils unter einem Modulnamen und einem Eintragsnamen und exportiert sein Angebot unter eigenen Namen. Es gibt vier Zahlentypen: Ganzzahlen und Gleitkommazahlen mit jeweils 32 oder 64 Bit, geschrieben `i32`, `i64`, `f32` und `f64`. Hinzu kommen ein einziger 128-Bit-Vektortyp und opake Referenzen. Ganzzahlen haben für sich genommen kein Vorzeichen; erst die jeweilige Operation interpretiert sie mit oder ohne Vorzeichen. Eine 32-Bit-Ganzzahl dient auch als Wahrheitswert. Speicheradressen verwenden `i32` oder, bei einem 64-Bit-Speicher, `i64`. Anweisungen nehmen ihre Operanden von einem impliziten Stapel und legen ihre Ergebnisse darauf ab. Der Kontrollfluss ist strukturiert: Blöcke, Schleifen und Bedingungen sind verschachtelt, und ein Sprung kann nur eines dieser Konstrukte zum Ziel haben.

Byteweise adressierte Daten liegen in einem linearen Speicher: einem einzigen zusammenhängenden, beschreibbaren Bytefeld, das mit einer angegebenen Größe beginnt und während der Ausführung wachsen kann. Ein Lese- oder Schreibzugriff außerhalb seiner aktuellen Größe erreicht nichts anderes. Er löst einen Trap aus, einen sofortigen Laufzeitabbruch. Wasm selbst kann einen solchen Abbruch nicht behandeln, sondern ihn nur an den umgebenden Host melden. Das folgende Beispiel verwendet das für Menschen lesbare Textformat der Spezifikation.

Das Konzept für eine Entscheidung nutzen

Nutzen Sie „Was ist WebAssembly?“ für eine konkrete Entscheidung. Wir können helfen, das Konzept mit dem System in Verbindung zu bringen, das Sie entwickeln oder einkaufen. Oberflächen, die schnell laden, auf dem Smartphone funktionieren und sich in allen Browsern Ihrer Kunden gleich verhalten.

Unsere Leistungen entdecken Kontakt aufnehmen →

Die Annahmen hinter dem Begriff prüfen

Nutzen Sie die Erklärung und die verlinkten Arbeiten, um Fragen für Ihr Projekt zu formulieren. Wir können die Annahmen gemeinsam mit Ihnen prüfen, bevor ein Werkzeug ausgewählt wird. Full-Stack-GIS-Kartenentwicklung geleitet — Full-Stack-Kartenentwicklung von der Datenbank bis zum Browser

Full-Stack-GIS-Kartenentwicklung geleitet Kontakt aufnehmen →

Warum es wichtig ist

Der erste Grund ist Geschwindigkeit. Die Entwurfsziele verlangen eine Leistung nahe an nativem Code und ein Format, das sich in einem einzigen schnellen Durchlauf dekodieren, validieren und kompilieren lässt, unabhängig davon, ob die Kompilierung zur Laufzeit oder im Voraus erfolgt. Das Binärformat soll kompakter sein als Text und nativer Maschinencode und lässt sich als Datenstrom

verarbeiten: Die Arbeit kann beginnen, bevor das letzte Byte angekommen ist. Sogar die Typdeklaration jeder Funktion steht in einem anderen Abschnitt als ihr Rumpf. Dadurch lassen sich die Funktionsrümpfe parallel und schon während der Übertragung kompilieren. Ein Team mit einer gut getesteten Bibliothek in einer anderen Sprache kann diese für den Browser kompilieren, statt sie in JavaScript neu zu schreiben.

Der zweite Grund ist Portabilität. Dasselbe Modul läuft auf jedem modernen Prozessor, in Desktoprechnern wie in Mobilgeräten, und in jedem Host, der eine Engine einbettet: in einem Browser, einer eigenständigen Laufzeitumgebung oder einem anderen Programm. Die Spezifikation bevorzugt weder eine Sprache noch ein Programmier- oder Objektmodell. Das Format ist damit ein gemeinsames Übersetzungsziel und keine private Laufzeitumgebung für eine einzelne Sprache.

Der dritte Grund ist Abschottung. Ein Modul hat keinen unmittelbaren Zugriff auf den Rechner, auf dem es läuft. Dateien, Netz, Seite und Uhr bleiben unerreichbar, solange der Host ihm keine entsprechende Funktion als Import übergibt. Damit legt der Host genau fest, worauf der Code zugreifen darf. Für das Web hält die Web-API fest, dass Wasm überall dort laufen kann, wo JavaScript läuft, im Wesentlichen dasselbe Bedrohungsmodell hat und nichts zugänglich machen kann, was JavaScript nicht ohnehin zugänglich wäre. Version 3.0 erweiterte, was innerhalb dieser Grenze möglich ist: mit 64 Bit adressierbare Speicher, im Web auf 16 Gigabyte begrenzt, mehrere Speicher in einem Modul, Speicherbereinigung für Sprachen, die sie voraussetzen, und Ausnahmen, die nicht mehr an den Host weitergereicht werden müssen. Die Ankündigung bezeichnet dieses bisherige Weiterreichen als weder portabel noch effizient.

Wie es in der Praxis aussieht

Dieses Modul im Textformat exportiert genau eine Funktion, die zwei 32-Bit-Ganzzahlen addiert:

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

Der Funktionstyp legt fest, dass die Funktion zwei `i32`-Werte entgegennimmt und einen zurückgibt. Ihr Rumpf besteht aus drei Stapelanweisungen: `local.get 0` legt den ersten Parameter auf den Stapel, `local.get 1` den zweiten, und `i32.add` nimmt beide herunter und legt ihre Summe darauf. Sie bleibt als Ergebnis liegen. `(export "add")` gibt der Funktion den Namen, den die Außenwelt sieht. Im Binärformat beginnt die Datei mit den vier Bytes `0x00 0x61 0x73 0x6D`: einer Null, gefolgt von den Buchstaben asm. Danach folgt die Version des Binärformats, die weiterhin 1 ist.

Im Browser lädt die JavaScript-API das Modul:

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

`instantiateStreaming` kompiliert das Modul, während die Antwort noch eintrifft, und liefert beim Erfüllen des Promise das kompilierte Modul und eine Instanz davon. Die Exporte der Instanz sind gewöhnliche JavaScript-Funktionen. Der zweite Aufruf zeigt, was die Typen bedeuten: Ganzzahladdition in Wasm rechnet modulo 2 hoch 32, und die JavaScript-API liest ein `i32`-Ergebnis mit Vorzeichen. Eins mehr als der größte positive Wert kommt deshalb als die kleinste darstellbare negative Zahl zurück. Dabei wird keine Ausnahme ausgelöst. Bei einem Trap dagegen schon: Liest eine Funktion außerhalb ihres Speichers, endet der Aufruf mit einem `WebAssembly.RuntimeError`.

Ein Modul gibt an, wenn es etwas von seinem Host braucht. Enthält es `(import "env" "log" (func (param i32)))`, braucht die Instanziierung ein zweites Argument, `{ env: { log: console.log } }`. Dann kann das Modul `log` wie jede eigene Funktion aufrufen. Fehlt das zweite Argument, scheitert die Instanziierung mit einem `TypeError`. Wird ein `env`-Objekt ohne aufrufbares `log` übergeben, scheitert sie mit einem `WebAssembly.LinkError`. In beiden Fällen ist noch kein Code des Moduls gelaufen. Importe und Exporte bilden die gesamte Grenze. Ihre deklarierten Typen umfassen Zahlen und Referenzen; eine Adresse im Modulspeicher ist ein `i32` oder, bei einem 64-Bit-Speicher, ein `i64`.

Ein Portfolioeintrag über einen nativen macOS-Client in Swift 6 beschreibt den anderen Weg, Code aus einer anderen Sprache wiederzuverwenden. Bei dieser Arbeit im August 2026 wurde ein Rust-Kern als statisches Archiv in eine Swift-

Anwendung eingebunden und über seine C-Schnittstelle angesprochen, die der Eintrag als verbindlichen Vertrag behandelt. Eine schlanke Hüllschicht verwaltet jeden Zeiger und jede Zeichenkette, die diese Grenze überschreiten. Für alles, was die Schnittstelle zurückgibt, ist die Zuständigkeit für den Speicher festgelegt; ein Fehler führt zu einem Speicherleck oder Absturz, ohne Warnung des Compilers. Wasm kommt dabei nicht zum Einsatz. Gerade das macht den Vergleich nützlich: Dasselbe Problem einer schmalen Grenze zwischen zwei Sprachen wird durch natives Linken gelöst statt durch eine Importliste und abgeschotteten Speicher.

Wo es schiefgeht

Das Promise wird abgewiesen, bevor etwas kompiliert wird. Fehlt `Content-Type` oder hat es einen anderen Wert als `application/wasm`, ohne Beachtung der Groß- und Kleinschreibung, folgt ein `TypeError`. Die Spezifikation erlaubt keine zusätzlichen Parameter, nicht einmal ein leeres `application/wasm;`. Auch ein nicht erfolgreicher Status führt zum Fehler. Dasselbe gilt für eine Antwort von einer anderen Origin, die nicht durch CORS freigegeben ist. Korrigieren Sie Medientyp und Header des Servers. `WebAssembly.instantiate` auf selbst abgerufenen Bytes umgeht die Prüfung des Medientyps, verliert aber die Kompilierung während der Übertragung.

Eine Sicht auf den Speicher wird unbrauchbar. JavaScript sieht den Speicher eines Moduls über `memory.buffer` und die darauf aufgebauten Arrays. Bei nicht geteiltem Speicher hat dieser Puffer standardmäßig eine feste Länge. Wächst der Speicher, trennt die JavaScript-API diesen Puffer ab und ersetzt ihn durch einen neuen. Arrays auf dem alten Puffer verlieren dadurch den Zugriff auf den Speicher. Das geschieht sowohl nach einem Aufruf von `memory.grow()` aus JavaScript als auch nach einer `memory.grow`-Anweisung im Modul, während eines von außen harmlos wirkenden Aufrufs. Erzeugen Sie nach jedem Aufruf, der den Speicher vergrößern könnte, neue Sichten auf `memory.buffer`, statt eine Sicht für die ganze Lebensdauer der Seite aufzubewahren.

Wasm soll selbst auf die Seite zugreifen. Ein Modul hat keinen unmittelbaren Zugriff auf seine Umgebung. Jede Wirkung auf Seite, Netz oder Speicher läuft daher über eine Funktion, die JavaScript als Import übergibt. An derselben Grenze werden auch die Typen entscheidend: Ein `i64` wird als `BigInt` übergeben, nicht als JavaScript-Wert vom Typ Number. Eine Funktion mit einem `v128`-Parameter oder -Ergebnis löst bei jedem Aufruf aus JavaScript einen `TypeError` aus. Entwerfen Sie die Import- und Exportliste als Schnittstelle mit einfachen Zahlen und belassen Sie Vektoren auf der Wasm-Seite.

Die Sandbox wird mit einem sicheren Programm verwechselt. Sie schützt den Host vor dem Modul, aber nicht das Modul vor sich selbst. Die Spezifikation hält fest, dass eine nach Wasm kompilierte unsichere Sprache weiterhin ihre eigene Speicherstruktur innerhalb des linearen Speichers beschädigen kann. Ein Pufferüberlauf im kompilierten Code überschreibt die eigenen Daten des Moduls. Nur ein Zugriff jenseits des gesamten Speichers löst einen Trap aus. Behalten Sie die Prüfungen und Tests der Ausgangssprache bei und behandeln Sie Daten von außen als nicht vertrauenswürdige Eingaben.

Die Binärdatei wird ohne Schutz ausgeliefert. Laut Registrierung von `application/wasm` enthält das Format keinen Schutz für Integrität oder Vertraulichkeit. Wo dieser nötig ist, muss er von außen kommen, etwa durch HTTPS. Wie bei JavaScript begrenzen Content Security Policy und Same-Origin-Policy einige Wege des Ladens und Ausführens. Weil Wasm auf echter Hardware zu Maschinencode wird, warnt die Kernspezifikation außerdem vor möglichen Seitenkanalangriffen auf Hardwareebene. Liefern Sie Module über HTTPS aus und schützen Sie das Recht, ein Modul zu ändern, ebenso streng wie das Recht, ein Skript zu ändern.

Weiterführende Lektüre

- WebAssembly Specification — die Kernspezifikation für Version 3.0: Aufbau, Validierung, Ausführung sowie Binär- und Textformat.
- Introduction — Definition, Entwurfsziele, Geltungsbereich der Dokumente und Sicherheitsfragen.
- WebAssembly JavaScript Interface — Kompilieren und Instanziieren aus JavaScript, Typübergänge, `Memory.grow` und die drei Fehlerklassen.
- WebAssembly Web API — `instantiateStreaming`, die Prüfungen der Antwort und die Registrierung von `application/wasm`.
- Wasm 3.0 Completed — die Ankündigung vom 17. September 2025 und die Neuerungen von Version 3.0.
- WebAssembly Core Specification 1.0 — die W3C Recommendation vom 5. Dezember 2019.



Glossar

Für Ihre tatsächlichen Rahmenbedingungen auswählen

Beschreiben Sie den Arbeitsablauf, die vorhandenen Schnittstellen und die Nutzer, die die Veränderung benötigen. Wir können eine prüfbare Lieferung definieren und die Übergabe so gestalten, dass Ihr Team die gelieferten Bestandteile warten kann.

Preise ansehen Kontakt aufnehmen →

Die Begriffe praktisch nutzbar machen

Fragen Sie, wie „Was ist WebAssembly?“ auf Ihre Organisation zutrifft. Ein konkretes Beispiel aus Ihrem bestehenden System reicht aus, um das Gespräch zu beginnen.

Unsere Leistungen entdecken

welcome@engineer.company +45 50 10 61 84 Kopenhagen, Dänemark

Nehmen Sie diese Seite mit

Kontaktkarte

Mit Sorgfalt und Liebe gestaltet von D.

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

Copyright © 2025 – heute · Engineer ApS · Alle Rechte vorbehalten.

Gedruckt von https://engineer.company/de/glossary/webassembly/