Engineer

WebAssembly

et portabelt binært kodeformat som kjører i en sandkasse

Hva det er

WebAssembly, som regel forkortet til Wasm, er et binært format for kjørbar kode. Kjernespesifikasjonen kaller det «a safe, portable, low-level code format designed for efficient execution and compact representation» og beskriver det som et virtuelt instruksjonssett: instruksjonene til en tenkt prosessor, som en nettleser eller en annen vert først kontrollerer og deretter oversetter til maskinkode for den virkelige. Kortformen er en sammentrekning og ikke et akronym, og derfor skriver spesifikasjonen Wasm og ikke WASM.

En `.wasm` -fil er som regel resultatet av en kompilator for et annet språk. Hovedmålet er ifølge spesifikasjonen selv raske programmer på nettet, men formatet gjør ingen antakelser om nettet, så det kan også kjøre som en frittstående virtuell maskin eller inne i andre programmer. Standarden er lagdelt deretter: kjernespesifikasjonen definerer instruksjonene, formatene, valideringen og kjøringen og ikke noe bestemt miljø. Egne dokumenter definerer JavaScript-API-et, som kompilerer, lenker og kaller moduler, og web-API-et, som henter dem over nettverket.

Den første versjonen ble en W3C-anbefaling 5. desember 2019. Versjon 2.0, som i hovedsak var ferdig tre år før etterfølgeren, la til vektorinstruksjoner, masseoperasjoner på minnet, flere returverdier og enkle referansetyper. Prosjektet kunngjorde 17. september 2025 at versjon 3.0 nå er den gjeldende standarden, og at den allerede var med i de fleste store nettlesere. Den samme kunngjøringen nevner Java, OCaml, Scala, Kotlin, Scheme og Dart blant språkene som nå kompileres til Wasm ved hjelp av søppeltømmingen.

Ordforrådet er lite. Kodens enhet er en modul: den inneholder funksjoner, tabeller, minner og globale variabler, importerer det den trenger, hver import navngitt med et modulnavn og et elementnavn, og eksporterer det den tilbyr,

under egne navn. Det finnes fire talltyper, 32- og 64-bits heltall og flyttall, skrevet `i32`, `i64`, `f32` og `f64`, i tillegg til én 128-bits vektortype og ugjennomsiktige referanser. Heltallene har ikke noe fortegn i seg selv; hver operasjon leser dem enten med eller uten fortegn, og et 32-bits heltall fungerer også som boolsk verdi. Minneadresser bruker `i32` eller, for et 64-bits minne, `i64`. Instruksjonene henter operandene sine fra en implisitt stakk og legger resultatene tilbake på den, og kontrollflyten er strukturert: blokker, løkker og betingelser ligger inni hverandre, og et hopp kan bare gå til en av dem.

Byteadresserte data ligger i et lineært minne, en sammenhengende, skrivbar rekke av byte som starter i en oppgitt størrelse og kan vokse mens programmet kjører. En lesing eller skriving utenfor den gjeldende størrelsen når ikke noe annet. Den utløser en trap, som stopper kjøringen umiddelbart, og en trap kan ikke håndteres inne i Wasm, bare meldes videre til verten rundt. Eksempelet nedenfor bruker tekstformatet som spesifikasjonen definerer for at mennesker skal kunne lese det.

Bruk begrepet i en beslutning

Bruk «Hva er WebAssembly?» i en konkret beslutning. Vi kan knytte begrepet til systemet dere bygger eller kjøper. Grensesnitt som laster raskt, fungerer på mobilen og oppfører seg likt i alle nettlesere kundene dine måtte bruke.

Se tjenestene våre Ta kontakt →

Undersøk antakelsene bak ordet

Bruk forklaringen og det tilknyttede arbeidet til å formulere spørsmål for prosjektet. Vi kan undersøke antakelsene sammen før et verktøy velges. Styrte fullstack-utviklingen av GIS-kart — fullstack-kartutvikling, fra database til nettleser

Styrte fullstack-utviklingen av GIS-kart Ta kontakt →

Hvorfor det er viktig

Fart er den første grunnen. Designmålene krever ytelse nær native kode og et format som kan dekodes, valideres og kompileres i én rask gjennomgang, just in time eller på forhånd. Binærformatet skal være mindre enn typiske tekst- og maskinkodeformater, og det kan strømmes: arbeidet kan begynne før den siste byten har kommet fram. Binærformatet holder til og med hver funksjons typedeklarasjon i en annen seksjon enn kroppen, slik at kroppene kan kompileres parallelt og mens de strømmer inn. Et team som allerede har et godt testet bibliotek i et annet språk, kan kompilere det for nettleseren i stedet for å skrive det om i JavaScript.

Portabilitet er den andre. Den samme modulen kjører på alle moderne arkitekturer, på datamaskiner så vel som mobile enheter, og i enhver vert som bygger inn en motor: en nettleser, et frittstående kjøremiljø eller et annet program. Spesifikasjonen favoriserer ikke noe bestemt språk, noen bestemt programmeringsmodell eller noen bestemt objektmodell, så formatet er et felles mål og ikke ett språks private kjøremiljø.

Avgrensning er den tredje. En modul har ingen omgivende tilgang til datamaskinen den kjører på. Filer, nettverket, siden og klokken er utenfor rekkevidde med mindre verten gir den en funksjon for dem som import, så det er verten som bestemmer nøyaktig hva koden får røre. På nettet påpeker web-API-et at Wasm kan kjøre overalt der JavaScript kan, har i hovedsak den samme trusselmodellen og ikke kan avsløre mer enn JavaScript allerede kunne. Versjon 3.0 utvidet hva som får plass innenfor den grensen: minner som adresseres med 64 bit, på nettet begrenset til 16 gigabyte, flere minner i én modul, søppeltømming for språk som forventer det, og unntak som ikke lenger må rømme ut til verten, noe kunngjøringen kaller verken portabelt eller effektivt.

Slik ser det ut i praksis

Her er en modul i tekstformatet med én eksportert funksjon som legger sammen to 32-bits heltall:

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

Funksjonens type sier at den tar to `i32`-verdier og returnerer én. Kroppen er tre instruksjoner på stakken: `local.get 0` legger på den første parameteren, `local.get 1` den andre, og `i32.add` tar av begge og legger på summen, som blir liggende som resultatet. `(export "add")` gir funksjonen navnet omverdenen ser. Oversatt til binærformatet begynner filen med de fire bytene `0x00 0x61 0x73 0x6D`, en null fulgt av bokstavene asm, og deretter binærformatets versjon, som fortsatt er 1.

I en nettleser laster JavaScript-API-et den inn:

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

`instantiateStreaming` kompilerer modulen mens svaret fortsatt er på vei, og gir tilbake den kompilerte modulen og en instans av den. Instansens eksporter er vanlige JavaScript-funksjoner. Det andre kallet viser hva typene betyr: heltallsaddisjon i Wasm går rundt modulo 2 i 32. potens, og JavaScript-API-et leser et `i32` -resultat som et tall med fortegn, så én mer enn den største positive verdien kommer tilbake som den mest negative. Ingenting blir kastet. Det blir det derimot ved en trap: leser en funksjon utenfor minnet sitt, ender kallet i en `WebAssembly.RuntimeError` .

En modul som trenger noe fra verten sin, sier fra. Med `(import "env" "log" (func (param i32)))` i modulen trenger instansieringen et andre argument, `{ env: { log: console.log } }` , og modulen kan kalle `log` som en av sine egne funksjoner. Utelat det andre argumentet, så feiler instansieringen med en `TypeError` ; gi et `env` -objekt uten en kallbar `log` , så feiler den med en `WebAssembly.LinkError` , i begge tilfeller før noe av modulens kode har kjørt. Import og eksport er hele grensesnittet, med deklarerte typer som omfatter tall og referanser; en adresse i modulens minne er en `i32` , eller en `i64` for et 64-bits minne.

En porteføljeside om å bygge en native macOS-klient i Swift 6 beskriver den andre måten å gjenbruke kode fra et annet språk på. I det arbeidet, som ble gjort i august 2026, ble en Rust-kjerne lenket inn i en Swift-app som et statisk arkiv og nådd gjennom C-grensesnittet sitt, som siden behandler som kontrakten. En tynn wrapper eier hver peker og hver streng som krysser grensen, og alt grensesnittet returnerer, har en eierskapsregel, der en feil gir en lekkasje eller en krasj uten noen kompilatoradvarsel. Det er ingen Wasm med der, og det er nettopp det som gjør sammenligningen nyttig: det samme problemet med en smal grense mellom to språk, løst med native lenking i stedet for en importliste og et minne i en sandkasse.

Hvor det går galt

Promiset avvises før noe blir kompilert. En manglende `Content-Type` , eller en hvilken som helst annen verdi enn `application/wasm` uavhengig av store og små bokstaver, feiler med en `TypeError` , og spesifikasjonen påpeker at ekstra parametere ikke er tillatt, heller ikke en tom `application/wasm;` . En status som

ikke er ok, feiler på samme måte, og det gjør også et svar fra en annen opprinnelse når CORS ikke gjør det til samme opprinnelse. Rett opp serverens medietype og headere. `WebAssembly.instantiate` på byte du har hentet selv, hopper over kontrollen av medietypen, men mister den strømmede kompileringen.

En visning av minnet slutter å virke. JavaScript ser en moduls minne gjennom `memory.buffer` og arrayene som er bygd på den. For minne som ikke er delt, har bufferen som standard fast lengde. Når minnet vokser, kobler JavaScript-API-et fra den bufferen med fast lengde og erstatter den med en ny, så arrayer bygd på den gamle bufferen er koblet fra minnet. Det skjer etter `memory.grow()` kalt fra JavaScript og også etter en `memory.grow` -instruksjon inne i modulen, under et kall som ser harmløst ut utenfra. Lag visninger fra `memory.buffer` på nytt etter hvert kall som kan få minnet til å vokse, i stedet for å beholde én så lenge siden lever.

Å forvente at Wasm når ut til siden på egen hånd. En modul har ingen omgivende tilgang til noe, så enhver virkning på siden, nettverket eller lagringen går gjennom en funksjon som JavaScript gir den som import. Typene biter ved den samme grensen: en `i64` krysser som en `BigInt` , ikke som et tall, og en funksjon med en `v128` -parameter eller -returverdi kaster en `TypeError` hver gang JavaScript kaller den. Utform import- og eksportlisten som et grensesnitt med enkle tall på, og hold vektorene på Wasm-siden.

Å ta sandkassen for et trygt program. Sandkassen beskytter verten mot modulen. Den beskytter ikke modulen mot seg selv. Spesifikasjonen slår fast at den ikke kan hindre et utrygt språk kompilert til Wasm i å ødelegge sitt eget minneoppsett inne i det lineære minnet: et bufferoverløp i kompilert kode overskriver modulens egne data, og bare en tilgang forbi slutten av hele minnet utløser en trap. Behold kildespråkets egne kontroller og tester, og behandle data modulen får utenfra, som upålitelige.

Å sende ut binærfilen uten å beskytte den. Registreringen av `application/wasm` sier at formatet ikke har noen beskyttelse av integritet eller konfidensialitet, og at slik beskyttelse, der den trengs, må komme utenfra, for eksempel gjennom HTTPS. Som med JavaScript er noen måter å laste inn og kjøre den på begrenset gjennom same-origin-policyen eller Content Security Policy. Og fordi Wasm blir maskinkode på ekte maskinvare, advarer kjernespesifikasjonen om at det potensielt er sårbart for sidekanalangrep på maskinvarenivå. Server moduler over HTTPS, og pass på hvem som får endre en modul, like nøye som på hvem som får endre et skript.

Les mer

- WebAssembly Specification — kjernespesifikasjonen for versjon 3.0: struktur, validering, kjøring og binær- og tekstformatet.
- Introduction — definisjonen, designmålene, hva hvert dokument dekker, og sikkerhetshensynene.
- WebAssembly JavaScript Interface — kompilering og instansiering fra JavaScript, hvordan verdier krysser grensen, `Memory.grow` og de tre feilklassene.
- WebAssembly Web API — `instantiateStreaming`, kontrollene av svaret og registreringen av `application/wasm`.
- Wasm 3.0 Completed — kunngjøringen fra 17. september 2025 og hva versjon 3.0 legger til.
- WebAssembly Core Specification 1.0 — W3C-anbefalingen fra 5. desember 2019.

Ordliste

Velg ut fra deres faktiske rammer

Beskriv arbeidsflyten, eksisterende grensesnitt og brukerne som trenger endringen. Vi kan definere en testbar leveranse og en overlevering som gjør teamet i stand til å vedlikeholde løsningen.

Se hva det koster Ta kontakt →

Gjør begrepene praktiske

Spør hvordan «Hva er WebAssembly?» gjelder i organisasjonen deres. Et konkret eksempel fra dagens system er nok til å starte samtalen.

Se tjenestene våre

welcome@engineer.company +45 50 10 61 84 København, Danmark

Ta med deg denne siden

Visittkort

Laget med kjærlighet av D.

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

Opphavsrett © 2025 – nå · Engineer ApS · Alle rettigheter forbeholdt.

Skrevet ut fra https://engineer.company/nb/glossary/webassembly/