Engineer

WebAssembly

et portabelt binært kodeformat, der kører i en sandkasse

Hvad det er

WebAssembly, som regel forkortet til Wasm, er et binært format til kørbar kode. Kernespecifikationen kalder det »a safe, portable, low-level code format designed for efficient execution and compact representation« og beskriver det som et virtuelt instruktionssæt: instruktionerne til en tænkt processor, som en browser eller en anden vært først kontrollerer og derefter oversætter til maskinkode til den rigtige. Den korte form er en sammentrækning og ikke et akronym, og derfor skriver specifikationen Wasm og ikke WASM.

En `.wasm`-fil er som regel resultatet af en oversætter til et andet sprog. Hovedmålet er ifølge specifikationen selv hurtige programmer på nettet, men formatet gør ingen antagelser om nettet, så det kan også køre som en selvstændig virtuel maskine eller inde i andre programmer. Standarden er lagdelt derefter: kernespecifikationen definerer instruktionerne, formaterne, valideringen og afviklingen og intet bestemt miljø. Særskilte dokumenter definerer JavaScript-API'et, der oversætter, sammenkæder og kalder moduler, og web-API'et, der henter dem over netværket.

Den første version blev en W3C-anbefaling 5. december 2019. Version 2.0, der i det væsentlige var færdig tre år før sin efterfølger, tilføjede vektorinstruktioner, masseoperationer på hukommelsen, flere returværdier og enkle referencetyper. Den 17. september 2025 meddelte projektet, at version 3.0 nu er den gældende standard, og at den allerede var med i de fleste store browsere. Samme meddelelse nævner Java, OCaml, Scala, Kotlin, Scheme og Dart blandt de sprog, der nu oversættes til Wasm ved hjælp af dets garbage collector.

Ordforrådet er lille. Kodens enhed er et modul: det rummer funktioner, tabeller, hukommelser og globale variabler, importerer det, det har brug for, hver import navngivet med et modulnavn og et elementnavn, og eksporterer det, det

tilbyder, under sine egne navne. Der er fire taltyper, 32- og 64-bit heltal og kommatal, skrevet `i32`, `i64`, `f32` og `f64`, plus en enkelt 128-bit vektortype og uigennemsigtige referencer. Heltallene har intet fortegn i sig selv; hver operation læser dem som enten med eller uden fortegn, og et 32-bit heltal tjener også som boolsk værdi. Hukommelsesadresser bruger `i32` eller, for en 64-bit hukommelse, `i64`. Instruktionerne tager deres operander fra en implicit stak og lægger resultaterne tilbage på den, og kontrolflowet er struktureret: blokke, løkker og betingelser ligger inden i hinanden, og et spring kan kun gå til en af dem.

Byteadresserede data ligger i en lineær hukommelse, et sammenhængende, skrivbart array af bytes, som starter i en erklæret størrelse og kan vokse, mens programmet kører. En læsning eller skrivning uden for den aktuelle størrelse når ikke noget andet. Den udløser en trap, der stopper afviklingen med det samme, og en trap kan ikke håndteres inde i Wasm, kun meldes videre til værten omkring det. Eksemplet nedenfor bruger det tekstformat, som specifikationen definerer, for at mennesker kan læse det.

Brug begrebet i en beslutning

Brug »Hvad er WebAssembly?« i en konkret beslutning. Vi kan knytte begrebet til det system, I bygger eller køber. Brugerflader, der indlæses hurtigt, virker på en telefon og opfører sig ens i alle de browsere, jeres kunder nu bruger.

Se vores ydelser Kontakt os →

Undersøg antagelserne bag ordet

Brug forklaringen og det tilknyttede arbejde til at formulere spørgsmål til jeres projekt. Vi kan undersøge antagelserne sammen, før et værktøj vælges. Har styret full-stack-udviklingen af GIS-kort — full stack-kortløsning, fra database til browser

Har styret full-stack-udviklingen af GIS-kort Kontakt os →

Hvorfor det betyder noget

Hastighed er den første grund. Designmålene kræver en ydelse tæt på native kode og et format, der kan afkodes, valideres og oversættes i ét hurtigt gennemløb, just in time eller på forhånd. Binærformatet skal være mindre end typiske tekst- og maskinkodeformater, og det kan streames: arbejdet kan begynde, før den sidste byte er ankommet. Binærformatet holder endda hver funktions typeerklæring i en anden sektion end dens krop, så kroppene kan oversættes parallelt og mens de strømmer ind. Et team, der allerede har et velafprøvet bibliotek i et andet sprog, kan oversætte det til browseren i stedet for at skrive det om i JavaScript.

Portabilitet er den anden. Det samme modul kører på alle moderne arkitekturer, på computere såvel som mobile enheder, og i enhver vært, der indlejrer en motor: en browser, et selvstændigt runtime-miljø eller et andet program. Specifikationen favoriserer intet bestemt sprog, ingen bestemt programmeringsmodel og ingen bestemt objektmodel, så formatet er et fælles mål og ikke ét sprogs private runtime.

Afgrænsning er den tredje. Et modul har ingen omgivende adgang til den computer, det kører på. Filer, netværket, siden og uret er uden for rækkevidde, medmindre værten giver det en funktion til dem som import, så det er værten, der bestemmer præcis, hvad koden må røre. På nettet bemærker web-API'et, at Wasm kan køre overalt, hvor JavaScript kan, har i det væsentlige samme trusselsmodel og ikke kan afsløre mere, end JavaScript allerede kunne. Version 3.0 udvidede, hvad der kan ligge inden for den grænse: hukommelser, der adresseres med 64 bit, på nettet begrænset til 16 gigabyte, flere hukommelser i ét modul, en garbage collector til sprog, der forventer en, og undtagelser, der ikke længere skal flygte ud til værten, hvad meddelelsen kalder hverken portabelt eller effektivt.

Sådan ser det ud i praksis

Her er et modul i tekstformatet med en enkelt eksporteret funktion, der lægger to 32-bit heltal sammen:

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

Funktionens type siger, at den tager to `i32`-værdier og returnerer én. Kroppen er tre instruktioner på stakken: `local.get 0` lægger den første parameter på, `local.get 1` den anden, og `i32.add` tager begge af og lægger deres sum på, hvor den bliver liggende som resultatet. `(export "add")` giver funktionen det navn, omverdenen ser. Oversat til binærformatet begynder filen med de fire bytes `0x00 0x61 0x73 0x6D`, et nul efterfulgt af bogstaverne asm, og derefter binærformatets version, som stadig er 1.

I en browser indlæser JavaScript-API'et det:

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

`instantiateStreaming` oversætter modulet, mens svaret stadig er på vej, og leverer det oversatte modul og en instans af det. Instansens eksporter er almindelige JavaScript-funktioner. Det andet kald viser, hvad typerne betyder: heltalsaddition i Wasm løber rundt modulo 2 i 32. potens, og JavaScript-API'et læser et `i32` -resultat som et tal med fortegn, så én mere end den største positive værdi kommer tilbage som den mest negative. Der bliver ikke kastet noget. Det gør der til gengæld ved en trap: læser en funktion uden for sin hukommelse, ender kaldet i en `WebAssembly.RuntimeError` .

Et modul, der har brug for noget fra sin vært, siger det. Med `(import "env" "log" (func (param i32)))` i modulet skal instansieringen have et andet argument, `{ env: { log: console.log } }` , og modulet kan kalde `log` som en af sine egne funktioner. Udelad det andet argument, og instansieringen fejler med en `TypeError` ; giv et `env` -objekt uden en kaldbar `log` , og den fejler med en `WebAssembly.LinkError` , i begge tilfælde før noget af modulets kode har kørt. Import og eksport er hele grænsefladen, med erklærede typer, der omfatter tal og referencer; en adresse i modulets hukommelse er en `i32` eller en `i64` for en 64-bit hukommelse.

En porteføljeside om at bygge en native macOS-klient i Swift 6 beskriver den anden måde at genbruge kode fra et andet sprog på. I det arbejde, der blev udført i august 2026, blev en Rust-kerne linket ind i et Swift-program som et statisk arkiv og nået gennem sin C-grænseflade, som siden behandler som kontrakten. En tynd indpakning ejer hver pointer og hver streng, der krydser grænsen, og alt, hvad grænsefladen returnerer, har en ejerskabsregel, hvor en fejl giver en lækage eller et nedbrud uden nogen advarsel fra oversætteren. Der er ingen Wasm i spil dér, og det er netop det, der gør sammenligningen nyttig: det samme problem med en smal grænse mellem to sprog, løst med native linkning i stedet for en importliste og en hukommelse i en sandkasse.

Hvor det går galt

Promiset afvises, før noget bliver oversat. En manglende `Content-Type` eller en hvilken som helst anden værdi end `application/wasm` , uanset store og små bogstaver, fejler med en `TypeError` , og specifikationen bemærker, at ekstra

parametre ikke er tilladt, heller ikke et tomt `application/wasm;` . En status, der ikke er ok, fejler på samme måde, og det gør et svar fra en anden oprindelse også, når CORS ikke gør det til samme oprindelse. Ret serverens medietype og headers. `WebAssembly.instantiate` på bytes, man selv har hentet, springer kontrollen af medietypen over, men mister den streamede oversættelse.

En visning af hukommelsen går død. JavaScript ser et moduls hukommelse gennem `memory.buffer` og de arrays, der er bygget oven på den. For ikke-delt hukommelse har bufferen som standard fast længde. Når hukommelsen vokser, frakobler JavaScript-API'et den buffer med fast længde og erstatter den med en ny, så arrays bygget på den gamle buffer er skåret fra hukommelsen. Det sker efter `memory.grow()` kaldt fra JavaScript og også efter en `memory.grow` -instruktion inde i modulet, under et kald, der udefra ser harmløst ud. Opret visninger fra `memory.buffer` igen efter hvert kald, der kan få hukommelsen til at vokse, i stedet for at beholde én, så længe siden lever.

At forvente, at Wasm selv når ud til siden. Et modul har ingen omgivende adgang til noget, så enhver virkning på siden, netværket eller lageret går gennem en funktion, som JavaScript giver det som import. Typerne bider ved den samme grænse: en `i64` krydser som en `BigInt` , ikke som et tal, og en funktion med en `v128` -parameter eller -returværdi kaster en `TypeError` , hver gang JavaScript kalder den. Design import- og eksportlisten som en grænseflade med enkle tal på, og hold vektorerne på Wasm-siden.

At tage sandkassen for et sikkert program. Sandkassen beskytter værten mod modulet. Den beskytter ikke modulet mod sig selv. Specifikationen fastslår, at den ikke kan forhindre et usikkert sprog oversat til Wasm i at ødelægge sit eget hukommelseslayout inde i den lineære hukommelse: et bufferoverløb i oversat kode overskriver modulets egne data, og kun en adgang ud over enden af hele hukommelsen udløser en trap. Behold kildesprogets egne kontroller og tests, og behandl data, som modulet modtager udefra, som upålidelige.

At sende binærfilen ud uden at beskytte den. Registreringen af `application/wasm` siger, at formatet ikke indeholder nogen beskyttelse af integritet eller fortrolighed, og at den, hvor den er nødvendig, må komme udefra, for eksempel gennem HTTPS. Som med JavaScript er nogle måder at indlæse og køre den på begrænset gennem same-origin-politikken eller Content Security Policy. Og fordi Wasm bliver til maskinkode på rigtig hardware, advarer

kernespecifikationen om, at det potentielt er sårbart over for sidekanalangreb på hardwareniveau. Server moduler over HTTPS, og vogt over, hvem der må ændre et modul, lige så nøje som over, hvem der må ændre et script.

Læs mere

- WebAssembly Specification — kernespecifikationen for version 3.0: struktur, validering, afvikling og binær- og tekstformatet.
- Introduction — definitionen, designmålene, hvad hvert dokument dækker, og sikkerhedsovervejelserne.
- WebAssembly JavaScript Interface — oversættelse og instansiering fra JavaScript, hvordan værdier krydser grænsen, `Memory.grow` og de tre fejlklasser.
- WebAssembly Web API — `instantiateStreaming`, kontrollerne af svaret og registreringen af `application/wasm`.
- Wasm 3.0 Completed — meddelelsen fra 17. september 2025 og det, version 3.0 tilføjer.
- WebAssembly Core Specification 1.0 — W3C-anbefalingen fra 5. december 2019.

Ordliste

Vælg ud fra jeres faktiske vilkår

Beskriv arbejdsgangen, de eksisterende grænseflader og de brugere, der har brug for ændringen. Vi kan definere en testbar leverance og en overdragelse, der gør teamet i stand til at vedligeholde løsningen.

Se, hvad det koster Kontakt os →

Gør begreberne praktiske

Spørg, hvordan »Hvad er WebAssembly?« gælder i jeres organisation. Et konkret eksempel fra jeres nuværende system er nok til at begynde samtalen.

Se vores ydelser

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

Tag denne side med dig

Visitkort

Lavet med kærlighed af D.

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

Copyright © 2025 – nu · Engineer ApS · Alle rettigheder forbeholdes.

Udskrevet fra https://engineer.company/da/glossary/webassembly/