Engineer

WebAssembly

ett portabelt binärt kodformat som körs i en sandlåda

Vad det är

WebAssembly, oftast förkortat Wasm, är ett binärt format för körbar kod. Kärnspecifikationen kallar det ”a safe, portable, low-level code format designed for efficient execution and compact representation” och beskriver det som en virtuell instruktionsuppsättning: instruktionerna för en tänkt processor, som en webbläsare eller en annan värd först kontrollerar och sedan översätter till maskinkod för den verkliga. Kortformen är en sammandragning och inte en akronym, och därför skriver specifikationen Wasm och inte WASM.

En `.wasm`-fil är oftast resultatet av en kompilator för något annat språk. Huvudmålet är enligt specifikationen själv snabba program på webben, men formatet gör inga antaganden om webben, så det kan också köras som en fristående virtuell maskin eller inuti andra program. Standarden är skiktad därefter: kärnspecifikationen definierar instruktionerna, formaten, valideringen och exekveringen och ingen särskild miljö. Separata dokument definierar JavaScript-API:t, som kompilerar, länkar och anropar moduler, och webb-API:t, som hämtar dem över nätverket.

Version 1.0 blev en W3C-rekommendation den 5 december 2019. Version 2.0, som i allt väsentligt var klar tre år före sin efterföljare, lade till vektorinstruktioner, massoperationer på minnet, flera returvärden och enkla referenstyper. Den 17 september 2025 meddelade projektet att version 3.0 nu är den gällande standarden och att den redan fanns i de flesta stora webbläsare. Samma meddelande nämner Java, OCaml, Scala, Kotlin, Scheme och Dart bland de språk som nu kompileras till Wasm med hjälp av dess skräpsamling.

Ordförrådet är litet. Kodens enhet är en modul: den rymmer funktioner, tabeller, minnen och globala variabler, importerar det den behöver, varje import namngiven med ett modulnamn och ett elementnamn, och exporterar det den

erbjuder under egna namn. Det finns fyra taltyper, 32- och 64-bitars heltal och flyttal, skrivna `i32`, `i64`, `f32` och `f64`, plus en enda 128-bitars vektortyp och ogenomskinliga referenser. Heltalen har inget eget tecken; varje operation läser dem som antingen med eller utan tecken, och ett 32-bitars heltal fungerar också som booleskt värde. Minnesadresser använder `i32` eller, för ett 64-bitars minne, `i64`. Instruktionerna hämtar sina operander från en implicit stack och lägger tillbaka resultaten på den, och kontrollflödet är strukturerat: block, loopar och villkor ligger inuti varandra, och ett hopp kan bara gå till något av dem.

Byteadresserade data ligger i ett linjärt minne, en sammanhängande, skrivbar följd av byte som börjar i en angiven storlek och kan växa medan programmet körs. En läsning eller skrivning utanför den aktuella storleken når inget annat. Den utlöser en trap, som stoppar exekveringen omedelbart, och en trap kan inte hanteras inuti Wasm, bara rapporteras vidare till värden runt omkring. Exemplet nedan använder det textformat som specifikationen definierar för att människor ska kunna läsa det.

Använd begreppet i ett beslut

Använd ”Vad är WebAssembly?” i ett konkret beslut. Vi kan knyta begreppet till systemet ni bygger eller köper. Gränssnitt som laddar snabbt, fungerar i mobilen och beter sig likadant i alla webbläsare dina kunder råkar använda.

Se våra tjänster Hör av er →

Granska antagandena bakom ordet

Använd förklaringen och det länkade arbetet för att formulera frågor till projektet. Vi kan undersöka antagandena tillsammans innan ett verktyg väljs. Styrde full stack-utvecklingen av GIS-kartor — kartutveckling i full stack, från databas till webbläsare

Styrde full stack-utvecklingen av GIS-kartor Hör av er →

Varför det spelar roll

Hastighet är det första skälet. Designmålen kräver prestanda nära maskinkod och ett format som kan avkodas, valideras och kompileras i ett enda snabbt pass, just in time eller i förväg. Binärformatet ska vara mindre än typiska text- och maskinkodsformat, och det går att strömma: arbetet kan börja innan den sista byten har kommit fram. Binärformatet håller till och med varje funktions typdeklaration i en annan sektion än dess kropp, så att kropparna kan kompileras parallellt och medan de strömmar in. Ett team som redan har ett vältestat bibliotek i ett annat språk kan kompilera det för webbläsaren i stället för att skriva om det i JavaScript.

Portabilitet är det andra. Samma modul körs på alla moderna arkitekturer, på datorer såväl som mobila enheter, och i varje värd som bäddar in en motor: en webbläsare, en fristående körmiljö eller ett annat program. Specifikationen gynnar inget särskilt språk, ingen särskild programmeringsmodell och ingen särskild objektmodell, så formatet är ett gemensamt mål och inte ett enskilt språks privata körmiljö.

Avgränsning är det tredje. En modul har ingen omgivande åtkomst till datorn den körs på. Filer, nätverket, sidan och klockan ligger utom räckhåll om inte värden ger den en funktion för dem som import, så det är värden som avgör exakt vad koden får röra. På webben påpekar webb-API:t att Wasm kan köras överallt där JavaScript kan, har i stort sett samma hotmodell och inte kan exponera mer än JavaScript redan kunde. Version 3.0 vidgade vad som ryms innanför den gränsen: minnen som adresseras med 64 bitar, på webben begränsade till 16 gigabyte, flera minnen i en modul, skräpsamling för språk som förväntar sig det, och undantag som inte längre behöver fly ut till värden, något som meddelandet kallar varken portabelt eller effektivt.

Så ser det ut i praktiken

Här är en modul i textformatet med en enda exporterad funktion som adderar två 32-bitars heltal:

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

Funktionens typ säger att den tar två `i32` -värden och returnerar ett. Kroppen är tre instruktioner på stacken: `local.get 0` lägger på den första parametern, `local.get 1` den andra, och `i32.add` tar av båda och lägger på summan, som blir liggande som resultatet. `(export "add")` ger funktionen det namn som omvärlden ser. Översatt till binärformatet börjar filen med de fyra byten `0x00 0x61 0x73 0x6D` , en nolla följd av bokstäverna asm, och sedan binärformatets version, som fortfarande är 1.

I en webbläsare läser JavaScript-API:t in den:

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

`instantiateStreaming` kompilerar modulen medan svaret fortfarande är på väg och levererar den kompilerade modulen och en instans av den. Instansens exporter är vanliga JavaScript-funktioner. Det andra anropet visar vad typerna betyder: heltalsaddition i Wasm slår runt modulo 2 upphöjt till 32, och JavaScript-API:t läser ett `i32` -resultat som ett tal med tecken, så ett mer än det största positiva värdet kommer tillbaka som det mest negativa. Inget kastas. Det gör det däremot vid en trap: läser en funktion utanför sitt minne slutar anropet i ett `WebAssembly.RuntimeError` .

En modul som behöver något från sin värd säger det. Med `(import "env" "log" (func (param i32)))` i modulen behöver instansieringen ett andra argument, `{ env: { log: console.log } }` , och modulen kan anropa `log` som en av sina egna funktioner. Utelämna det andra argumentet så misslyckas instansieringen med ett `TypeError` ; skicka ett `env` -objekt utan en anropbar `log` så misslyckas den med ett `WebAssembly.LinkError` , i båda fallen innan något av modulens kod har körts. Import och export är hela gränssnittet, med deklarerade typer som omfattar tal och referenser; en adress i modulens minne är en `i32` , eller en `i64` för ett 64-bitars minne.

En portföljsida om att bygga en native macOS-klient i Swift 6 beskriver det andra sättet att återanvända kod från ett annat språk. I det arbetet, som gjordes i augusti 2026, länkades en Rust-kärna in i en Swift-app som ett statiskt arkiv och nåddes genom sitt C-gränssnitt, som sidan behandlar som kontraktet. En tunn wrapper äger varje pekare och varje sträng som korsar gränsen, och allt som gränssnittet returnerar har en ägarskapsregel, där ett misstag ger en läcka eller en krasch utan någon kompilatorvarning. Ingen Wasm är inblandad där, och det är just det som gör jämförelsen användbar: samma problem med en smal gräns mellan två språk, löst med native-länkning i stället för en importlista och ett minne i en sandlåda.

Var det går fel

Promiset avvisas innan något har kompilerats. En saknad `Content-Type` , eller något annat värde än `application/wasm` oavsett skiftläge, misslyckas med ett `TypeError` , och specifikationen påpekar att extra parametrar inte är tillåtna, inte

ens ett tomt `application/wasm;` . En status som inte är ok misslyckas på samma sätt, och det gör också ett svar från ett annat ursprung när CORS inte gör det till samma ursprung. Rätta serverns medietyp och headers. `WebAssembly.instantiate` på byte som du själv har hämtat hoppar över kontrollen av medietypen men förlorar den strömmade kompileringen.

En vy av minnet slutar fungera. JavaScript ser en moduls minne genom `memory.buffer` och de arrayer som byggs på den. För minne som inte är delat har bufferten som standard fast längd. När minnet växer kopplar JavaScript-API:t loss den bufferten med fast längd och ersätter den med en ny, så arrayer byggda på den gamla bufferten är avskurna från minnet. Det händer efter `memory.grow()` anropat från JavaScript och också efter en `memory.grow` -instruktion inuti modulen, under ett anrop som utifrån ser harmlöst ut. Skapa vyer från `memory.buffer` på nytt efter varje anrop som kan få minnet att växa, i stället för att behålla en under hela sidans livstid.

Att vänta sig att Wasm når sidan på egen hand. En modul har ingen omgivande åtkomst till något, så varje effekt på sidan, nätverket eller lagringen går genom en funktion som JavaScript ger den som import. Typerna biter vid samma gräns: en `i64` korsar som en `BigInt` , inte som ett tal, och en funktion med en `v128` -parameter eller ett `v128` -returvärde kastar ett `TypeError` varje gång JavaScript anropar den. Utforma import- och exportlistan som ett gränssnitt med enkla tal, och håll vektorerna på Wasm-sidan.

Att ta sandlådan för ett säkert program. Sandlådan skyddar värden mot modulen. Den skyddar inte modulen mot sig själv. Specifikationen slår fast att den inte kan hindra ett osäkert språk som kompilerats till Wasm från att förstöra sin egen minneslayout inuti det linjära minnet: ett buffertöverskridande i kompilerad kod skriver över modulens egna data, och bara en åtkomst bortom slutet av hela minnet utlöser en trap. Behåll källspråkets egna kontroller och tester, och behandla data som modulen tar emot utifrån som opålitliga.

Att skicka ut binärfilen utan att skydda den. Registreringen av `application/wasm` säger att formatet saknar skydd för integritet och sekretess, och att sådant skydd, där det behövs, måste komma utifrån, till exempel genom HTTPS. Liksom för JavaScript begränsas vissa sätt att läsa in och köra den genom same-origin-policyn eller Content Security Policy. Och eftersom Wasm blir maskinkod på verklig hårdvara varnar kärnspecifikationen för att det kan vara sårbart för sidokanalsattacker på hårdvarunivå. Servera moduler över HTTPS, och vakta vem som får ändra en modul lika noga som vem som får ändra ett skript.

Läs mer

- WebAssembly Specification — kärnspecifikationen för version 3.0: struktur, validering, exekvering och binär- och textformatet.
- Introduction — definitionen, designmålen, vad varje dokument täcker och säkerhetsöverväganden.
- WebAssembly JavaScript Interface — kompilering och instansiering från JavaScript, hur värden korsar gränsen, `Memory.grow` och de tre felklasserna.
- WebAssembly Web API — `instantiateStreaming`, kontrollerna av svaret och registreringen av `application/wasm`.
- Wasm 3.0 Completed — meddelandet från den 17 september 2025 och vad version 3.0 lägger till.
- WebAssembly Core Specification 1.0 — W3C-rekommendationen från den 5 december 2019.

Ordlista

Välj utifrån era faktiska förutsättningar

Beskriv arbetsflödet, befintliga gränssnitt och användarna som behöver förändringen. Vi kan definiera en testbar leverans och en överlämning som gör att teamet kan underhålla lösningen.

Se vad det kostar Hör av er →

Gör begreppen praktiska

Fråga hur ”Vad är WebAssembly?” gäller i er organisation. Ett konkret exempel från ert nuvarande system räcker för att börja samtalet.

Se våra tjänster

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

Ta med dig den här sidan

Visitkort

Omsorgsfullt skapad med kärlek av D.

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

Upphovsrätt © 2025 – nu · Engineer ApS · Alla rättigheter förbehållna.

Utskrivet från https://engineer.company/sv/glossary/webassembly/