Engineer

WebAssembly

переносний бінарний формат коду, що виконується в пісочниці

Що це таке

WebAssembly, або скорочено Wasm, — це бінарний формат виконуваного коду. Основна специфікація називає його «a safe, portable, low-level code format designed for efficient execution and compact representation» і описує як віртуальну систему команд: команди уявного процесора, які браузер чи інше середовище спершу перевіряє, а потім перекладає в машинний код справжнього. Коротка форма — це скорочення, а не абревіатура, тому специфікація пише Wasm, а не WASM.

Файл `.wasm` зазвичай є результатом роботи компілятора з якоїсь іншої мови. Головна мета, як каже сама специфікація, — швидкі застосунки у вебі, але формат не робить жодних припущень про веб, тож може працювати й як окрема віртуальна машина, і всередині інших програм. Стандарт поділено на шари відповідно до цього: основна специфікація визначає команди, формати, валідацію та виконання і не прив'язана до жодного середовища. Окремі документи визначають JavaScript API, що компілює, компонує й викликає модулі, і Web API, що завантажує їх мережею.

Перша версія стала рекомендацією W3C 5 грудня 2019 року. Версія 2.0, по суті завершена за три роки до наступної, додала векторні команди, групові операції з пам'яттю, кілька значень, що повертаються, і прості типи посилань. Версію 3.0 проєкт оголосив чинним стандартом 17 вересня 2025 року й повідомив, що вона вже працює в більшості основних браузерів. У тому самому оголошенні Java, OCaml, Scala, Kotlin, Scheme і Dart названо серед мов, які тепер компілюються у Wasm з опорою на його збирач сміття.

Словник невеликий. Одиниця коду — модуль: він містить функції, таблиці, пам'ять і глобальні змінні, імпортує те, чого потребує, причому кожен імпорт має назву модуля й назву елемента, і експортує те, що пропонує, під

власними назвами. Числових типів чотири: 32- і 64-бітові цілі числа та числа з рухомою комою, що записуються як `i32`, `i64`, `f32` і `f64`, а ще один 128-бітовий векторний тип і непрозорі посилання. Цілі числа самі не мають знака; кожна операція трактує їх як знакові або беззнакові, а 32-бітове ціле слугує ще й логічним значенням. Адреси в пам'яті використовують `i32` або, для 64-бітової пам'яті, `i64`. Команди беруть операнди з неявного стека й кладуть туди результати, а потік керування структурований: блоки, цикли й умови вкладаються одне в одне, і перехід може вести лише до однієї з цих конструкцій.

Дані з побайтовою адресацією лежать у лінійній пам'яті — суцільному масиві байтів, доступному для запису, що має початковий розмір і може зростати під час роботи програми. Читання чи запис поза її поточним розміром нікуди більше не дістається. Воно спричиняє пастку (trap), що негайно зупиняє виконання, і всередині Wasm пастку обробити не можна, про неї можна лише повідомити середовищу навколо. Приклад нижче написано в текстовому форматі, який специфікація визначає для того, щоб код могли читати люди.

Застосуйте поняття до рішення

Застосуйте «Що таке WebAssembly?» до конкретного рішення. Ми можемо пов'язати поняття із системою, яку ви створюєте або купуєте. Інтерфейси, що швидко завантажуються, працюють на телефоні й поводяться однаково в усіх браузерах ваших клієнтів.

Переглянути наші послуги Зв'язатися з нами →

Перевірте припущення за терміном

Використайте пояснення та пов'язані роботи, щоб сформулювати запитання для проєкту. Ми можемо перевірити припущення разом до вибору інструмента. Керувала full-stack-розробкою GIS-карт — full-stack-картографія — від бази даних до браузера

Керувала full-stack-розробкою GIS-карт Зв'язатися з нами →

Чому це важливо

Перша причина — швидкість. Цілі проєктування вимагають продуктивності, близької до нативного коду, і формату, який можна декодувати, перевірити й скомпілювати за один швидкий прохід, під час виконання або заздалегідь. Бінарний формат має бути меншим за типові текстові та машинні формати, і його можна передавати потоком: робота може початися ще до того, як надійде останній байт. Бінарний формат навіть зберігає оголошення типу кожної функції в іншому розділі, ніж її тіло, щоб тіла можна було компілювати паралельно й просто під час надходження. Команда, яка вже має добре перевірену бібліотеку іншою мовою, може скомпілювати її для браузера, а не переписувати на JavaScript.

Друга — переносність. Той самий модуль працює на всіх сучасних архітектурах, і на комп'ютерах, і на мобільних пристроях, та в будь-якому середовищі, що вбудовує рушій: у браузері, в окремому середовищі виконання чи в іншій програмі. Специфікація не віддає переваги жодній мові, моделі програмування чи об'єктній моделі, тож формат є спільною ціллю компіляції, а не приватним середовищем однієї мови.

Третя — ізоляція. Модуль не має неявного доступу до комп'ютера, на якому працює. Файли, мережа, сторінка й годинник для нього недосяжні, доки середовище не передасть йому функцію для них як імпорт, тож саме середовище вирішує, чого саме код може торкатися. У вебі Web API зазначає, що Wasm може виконуватися скрізь, де й JavaScript, має по суті ту саму модель загроз і не може розкрити більше, ніж уже може JavaScript. Версія 3.0 розширила те, що вміщується в ці межі: пам'ять із 64-бітовою адресацією, у вебі обмежена 16 гігабайтами, кілька блоків пам'яті в одному модулі, збирання сміття для мов, які на нього розраховують, і винятки, яким більше не треба тікати в середовище, що оголошення називає ні переносним, ні ефективним.

Як це виглядає на практиці

Ось модуль у текстовому форматі з однією експортованою функцією, що додає два 32-бітові цілі числа:

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

Тип функції каже, що вона приймає два значення `i32` і повертає одне. Тіло — це три команди на стеку: `local.get 0` кладе перший параметр, `local.get 1` — другий, а `i32.add` знімає обидва й кладе їхню суму, яка й лишається на стеку як результат. `(export "add")` дає функції назву, яку бачить зовнішній світ. У бінарному форматі файл починається з чотирьох байтів `0x00 0x61 0x73 0x6D`, тобто нуля й літер asm, а далі йде версія бінарного формату, досі перша.

У браузері модуль завантажує JavaScript API:

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

`instantiateStreaming` компілює модуль, поки відповідь ще надходить, і повертає скомпільований модуль та його екземпляр. Експорти екземпляра — звичайні функції JavaScript. Другий виклик показує, що означають типи: додавання цілих чисел у Wasm іде за модулем 2 у 32-му степені, а JavaScript API читає результат `i32` як знакове число, тож одиниця понад найбільше додатне значення повертається як найменше від'ємне. Жодного винятку не виникає. А от пастка його спричинить: якщо функція читатиме поза своєю пам'яттю, виклик завершиться помилкою `WebAssembly.RuntimeError` .

Модуль, якому щось потрібно від середовища, так і каже. Якщо в модулі є `(import "env" "log" (func (param i32)))` , створенню екземпляра потрібен другий аргумент, `{ env: { log: console.log } }` , і модуль може викликати `log` , як будь-яку власну функцію. Без другого аргументу створення екземпляра завершиться помилкою `TypeError` ; з об'єктом `env` , у якому немає функції `log` , — помилкою `WebAssembly.LinkError` , і в обох випадках раніше, ніж виконається хоч рядок коду модуля. Імпорт і експорт — це вся межа з оголошеними типами, що включають числа та посилання; адреса в пам'яті модуля — це `i32` або `i64` для 64-бітової пам'яті.

Сторінка портфоліо про створення нативного клієнта для macOS на Swift 6 описує інший спосіб повторно використати код з іншої мови. У тій роботі, виконаній у серпні 2026 року, ядро на Rust скомпонували із застосунком на Swift як статичний архів і викликали через його C-інтерфейс, який сторінка вважає контрактом. Тонка обгортка володіє кожним вказівником і кожним рядком, що перетинає межу, а для всього, що повертає інтерфейс, діє правило володіння, і помилка в ньому дає витік пам'яті або аварійне завершення без жодного попередження компілятора. Wasm там немає, і саме тому порівняння корисне: та сама задача вузької межі між двома мовами, але розв'язана нативним компонуванням, а не списком імпортів і пам'яттю в пісочниці.

Де виникають проблеми

Проміс відхиляється ще до компіляції. Відсутній `Content-Type` або будь-яке значення, крім `application/wasm` без огляду на регістр, дає `TypeError`, а специфікація зауважує, що додаткові параметри не дозволені, навіть порожній `application/wasm;`. Так само завершується статус, що не є успішним, і відповідь з іншого джерела, яку CORS не робить відповіддю з того самого джерела. Виправте тип медіа й заголовки на сервері. `WebAssembly.instantiate` з байтами, які ви завантажили самі, пропускає перевірку типу медіа, але втрачає потокову компіляцію.

Подання пам'яті перестає працювати. JavaScript бачить пам'ять модуля через `memory.buffer` і масиви, побудовані на ньому. Для пам'яті без спільного доступу цей буфер за замовчуванням має фіксовану довжину. Коли пам'ять зростає, JavaScript API від'єднує цей буфер фіксованої довжини й замінює його новим, тож масиви на старому буфері відрізано від пам'яті. Це стається після виклику `memory.grow()` з JavaScript, а також після команди `memory.grow` усередині модуля, під час виклику, який ззовні здається безневинним. Створюйте подання з `memory.buffer` наново після кожного виклику, що може збільшити пам'ять, а не тримайте одне на весь час життя сторінки.

Очікування, що Wasm сам дістанеться до сторінки. Модуль не має неявного доступу ні до чого, тож будь-яка дія зі сторінкою, мережею чи сховищем іде через функцію, яку JavaScript передає йому як імпорт. На тій самій межі дошкуляють і типи: `i64` перетинає її як `BigInt`, а не як число, а функція з параметром чи результатом `v128` кидає `TypeError` щоразу, коли JavaScript її викликає. Проєктуйте список імпортів і експортів як інтерфейс із простими числами, а вектори тримайте на боці Wasm.

Сприйняття пісочниці як безпечної програми. Пісочниця захищає середовище від модуля. Модуль від самого себе вона не захищає. Специфікація прямо каже, що не може завадити небезпечній мові, скомпільованій у Wasm, зіпсувати власне розташування даних у лінійній пам'яті: переповнення буфера в скомпільованому коді перезаписує власні дані модуля, а пастку спричиняє лише доступ за межі всієї пам'яті. Зберігайте перевірки й тести вихідної мови та вважайте дані, які модуль отримує ззовні, недовіреними.

Розповсюдження бінарного файлу без захисту. Реєстрація типу `application/wasm` каже, що формат не має жодного захисту цілісності чи конфіденційності, і якщо такий захист потрібен, його мають забезпечити ззовні, наприклад через HTTPS. Як і для JavaScript, деякі способи завантаження й виконання обмежуються політикою того самого джерела або Content Security Policy. А оскільки Wasm перетворюється на машинний код на справжньому залізі, основна специфікація попереджає, що він потенційно вразливий до атак через побічні канали на рівні апаратури. Віддавайте модулі через HTTPS і стежте за тим, хто може змінити модуль, так само пильно, як за тим, хто може змінити скрипт.

Що почитати далі

- WebAssembly Specification — основна специфікація версії 3.0: структура, валідація, виконання, бінарний і текстовий формати.
- Introduction — визначення, цілі проєктування, що охоплює кожен документ, і міркування безпеки.
- WebAssembly JavaScript Interface — компіляція й створення екземплярів із JavaScript, як значення перетинають межу, `Memory.grow` і три класи помилок.
- WebAssembly Web API — `instantiateStreaming`, перевірки відповіді та реєстрація `application/wasm`.
- Wasm 3.0 Completed — оголошення від 17 вересня 2025 року і що додає версія 3.0.
- WebAssembly Core Specification 1.0 — рекомендація W3C від 5 грудня 2019 року.

Словник

Обирайте за фактичними умовами

Опишіть робочий процес, наявні інтерфейси та користувачів, яким потрібна зміна. Ми можемо визначити перевірюваний результат і спосіб передачі, який дасть вашій команді змогу підтримувати його.

Дивитися, скільки це коштує Зв'язатися з нами →

Зробіть термінологію практичною

Запитайте, як «Що таке WebAssembly?» стосується вашої організації. Конкретного прикладу з поточної системи достатньо для початку розмови.

Переглянути наші послуги

welcome@engineer.company +45 50 10 61 84 Копенгаген, Данія

Візьміть цю сторінку із собою

Візитівка

Дбайливо зроблено з любов'ю — D.

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

Авторське право © 2025 – дотепер · Engineer ApS · Усі права захищено.

Надруковано з https://engineer.company/uk/glossary/webassembly/