Engineer

WebAssembly

un format binaire portable de code exécuté en bac à sable

Ce que c’est

WebAssembly, généralement abrégé en Wasm, est un format binaire de code exécutable. Sa spécification centrale le qualifie de « format de code sûr, portable et de bas niveau, conçu pour une exécution efficace et une représentation compacte » et le décrit comme un jeu d’instructions virtuel : les instructions d’un processeur imaginaire, qu’un navigateur ou un autre hôte vérifie d’abord, puis traduit en code machine pour le processeur réel. La forme courte est une contraction, pas un sigle ; c’est pourquoi la spécification écrit Wasm et non WASM.

Un fichier `.wasm` est généralement le résultat d’un compilateur pour un autre langage. Selon la spécification, l’objectif principal est de permettre des applications rapides sur le Web. Le format ne fait toutefois aucune hypothèse sur le Web : il peut aussi fonctionner comme une machine virtuelle autonome ou à l’intérieur d’autres programmes. Le standard est organisé en couches en conséquence : la spécification centrale définit les instructions, les formats, la validation et l’exécution, sans imposer d’environnement particulier. Des documents distincts définissent l’API JavaScript qui compile, lie et appelle les modules, ainsi que l’API Web qui les charge par le réseau.

La première version est devenue une recommandation du W3C le 5 décembre 2019. La version 2.0, achevée pour l’essentiel trois ans avant celle qui lui a succédé, a ajouté des instructions travaillant sur des vecteurs, des opérations en bloc sur la mémoire, des fonctions ayant plusieurs valeurs de retour et une première forme simple de types référence. Le 17 septembre 2025, le projet a annoncé la version 3.0 comme le standard désormais en vigueur, en précisant qu’elle était déjà livrée dans la plupart des grands navigateurs. La même annonce cite Java, OCaml, Scala, Kotlin, Scheme et Dart parmi les langages qui ciblent désormais Wasm grâce à son ramasse-miettes.

Le vocabulaire est réduit. L'unité de code est un module : il contient des fonctions, des tables, des mémoires et des variables globales, importe ce dont il a besoin, chaque import étant désigné par un nom de module et un nom d'élément, et exporte ce qu'il propose sous ses propres noms. Il existe quatre types numériques, des entiers et des nombres à virgule flottante de 32 et 64 bits, notés `i32` , `i64` , `f32` et `f64` , ainsi qu'un unique type vectoriel de 128 bits et des références opaques. Les entiers n'ont pas de signe intrinsèque ; chaque opération les interprète comme signés ou non signés, et un entier de 32 bits sert aussi de booléen. Les adresses mémoire utilisent `i32` ou, pour une mémoire de 64 bits, `i64` . Les instructions prennent leurs opérandes sur une pile implicite et y déposent leurs résultats. Le flux de contrôle est structuré : blocs, boucles et conditions s'imbriquent, et un branchement ne peut cibler que l'une de ces constructions.

Les données adressées par octet résident dans une mémoire linéaire, un tableau continu d'octets accessible en écriture, qui commence avec une taille déclarée et peut croître pendant l'exécution du programme. Une lecture ou une écriture au-delà de sa taille actuelle n'atteint rien d'autre. Elle déclenche un trap qui arrête immédiatement l'exécution ; un trap ne peut pas être traité à l'intérieur de Wasm, seulement signalé à l'hôte qui l'entoure. L'exemple ci-dessous utilise le format textuel que la spécification définit pour la lecture humaine.

Utilisez le concept pour prendre une décision

Utilisez « Qu'est-ce que WebAssembly ? » pour éclairer une décision réelle. Nous pouvons vous aider à relier ce concept au système que vous construisez ou achetez. Des interfaces rapides, utilisables sur téléphone et cohérentes dans tous les navigateurs de vos clients.

Découvrir nos prestations Contactez-nous →

Vérifiez les hypothèses derrière le terme

Appuyez-vous sur l'explication et les réalisations liées pour formuler les questions de votre projet. Nous pouvons examiner les hypothèses avec vous avant le choix d'un outil. Développement cartographique GIS full-stack — cartographie full-stack, de la base au navigateur

Développement cartographique GIS full-stack Contactez-nous →

Pourquoi c'est important

La vitesse est la première raison. Les objectifs de conception demandent des performances proches du code natif et un format qu'un seul passage rapide peut décoder, valider et compiler, que la compilation se fasse à la volée ou à l'avance. Le binaire vise une représentation plus compacte que le texte et que le code machine natif, et il peut être traité au fil de sa réception : le travail peut commencer avant l'arrivée du dernier octet. Le format binaire place même la déclaration de type de chaque fonction dans une section distincte de son corps,

afin que les corps puissent être compilés en parallèle et pendant leur réception. Une équipe disposant déjà d'une bibliothèque bien testée dans un autre langage peut la compiler pour le navigateur au lieu de la réécrire en JavaScript.

La portabilité est la deuxième raison. Le même module fonctionne sur tout processeur moderne, sur ordinateur ou sur mobile, et dans tout hôte intégrant un moteur : navigateur, environnement d'exécution autonome ou autre programme. La spécification ne privilégie aucun langage, modèle de programmation ou modèle objet particulier. Le format constitue donc une cible commune plutôt que l'environnement d'exécution privé d'un langage.

Le cloisonnement est la troisième raison. Un module ne dispose d'aucun accès implicite à l'ordinateur qui l'exécute. Les fichiers, le réseau, la page et l'horloge restent hors d'atteinte, sauf si l'hôte lui fournit une fonction pour y accéder sous forme d'import. L'hôte décide ainsi exactement de ce que le code peut toucher. Sur le Web, l'API Web précise que Wasm peut fonctionner partout où JavaScript le peut, avec essentiellement le même modèle de menace, sans pouvoir exposer davantage que JavaScript. La version 3.0 a élargi ce qui tient dans cette frontière : des mémoires adressables sur 64 bits, limitées sur le Web à 16 Go, plusieurs mémoires dans un module, un ramasse-miettes pour les langages qui en attendent un et des exceptions qui n'ont plus à sortir vers l'hôte, une sortie que l'annonce ne juge ni portable ni efficace.

À quoi cela ressemble en pratique

Voici un module au format textuel avec une seule fonction exportée, qui additionne deux entiers de 32 bits :

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

Le type de la fonction indique qu'elle prend deux valeurs `i32` et en renvoie une. Son corps comporte trois instructions sur la pile : `local.get 0` empile le premier paramètre, `local.get 1` empile le second et `i32.add` les dépile tous deux, puis empile leur somme, laissée sur la pile comme résultat. `(export "add")` donne à la

fonction le nom que voit le monde extérieur. Converti au format binaire, le fichier commence par les quatre octets `0x00 0x61 0x73 0x6D`, un zéro suivi des lettres asm, puis par la version du format binaire, qui est toujours 1.

Dans un navigateur, l'API JavaScript le charge :

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

`instantiateStreaming` compile le module pendant que la réponse arrive encore, et sa promesse se résout avec le module compilé et une instance de celui-ci. Les exports de l'instance sont des fonctions JavaScript ordinaires. Le second appel montre ce que signifient les types : l'addition entière dans Wasm se fait modulo 2 à la puissance 32, et l'API JavaScript interprète un résultat `i32` comme signé. Une unité au-delà de la plus grande valeur positive revient donc comme la valeur la plus négative. Aucune exception n'est levée. Un trap en provoquerait une : si une fonction lit au-delà de sa mémoire, l'appel se termine par une `WebAssembly.RuntimeError`.

Un module qui a besoin de quelque chose de son hôte le déclare. Avec `(import "env" "log" (func (param i32)))` dans le module, l'instanciation exige un second argument, `{ env: { log: console.log } }`, et le module peut appeler `log` comme l'une de ses propres fonctions. Omettez le second argument et l'instanciation échoue avec une `TypeError` ; fournissez un objet `env` sans `log` appelable et elle échoue avec une `WebAssembly.LinkError`, dans les deux cas avant l'exécution de la moindre instruction du module. Imports et exports constituent toute la frontière, avec des types déclarés comprenant des nombres et des références ; une adresse dans la mémoire du module est un `i32`, ou un `i64` pour une mémoire de 64 bits.

Une étude de cas sur la création d'un client macOS natif en Swift 6 décrit l'autre manière de réutiliser du code d'un autre langage. Dans ce travail réalisé en août 2026, un noyau Rust était lié à une application Swift sous forme d'archive statique et accessible par son interface C, présentée dans l'étude comme le contrat. Une fine couche d'encapsulation assume la propriété de chaque pointeur et de chaque chaîne qui franchit la frontière. Tout ce que renvoie l'interface obéit à une règle de propriété ; une erreur entraîne une fuite mémoire

ou un plantage, sans avertissement du compilateur. Wasm n'intervient pas dans ce travail, ce qui rend la comparaison utile : le même problème d'une frontière étroite entre deux langages, résolu par une liaison native plutôt que par une liste d'imports et une mémoire cloisonnée.

Où les problèmes surviennent

La promesse est rejetée avant toute compilation. L'absence de `Content-Type`, ou toute valeur autre qu'`application/wasm` sans distinction de casse, provoque une `TypeError`. La spécification précise que les paramètres supplémentaires sont interdits, même sous la forme `application/wasm;` avec un paramètre vide. Le même échec se produit si le statut n'indique pas une réussite ou si la réponse provient d'une autre origine sans autorisation CORS. Corrigez le type de média et les en-têtes du serveur. `WebAssembly.instantiate` sur des octets récupérés par vos soins évite la vérification du type de média, mais perd la compilation au fil de la réception.

Une vue de la mémoire cesse de fonctionner. JavaScript voit la mémoire d'un module par `memory.buffer` et les tableaux construits dessus. Pour une mémoire non partagée, ce tampon a une longueur fixe par défaut. Lorsque la mémoire croît, l'API JavaScript détache ce tampon de longueur fixe et le remplace par un nouveau ; les tableaux construits sur l'ancien tampon sont donc coupés de la mémoire. Cela arrive après un appel à `memory.grow()` depuis JavaScript, mais aussi après une instruction `memory.grow` dans le module, pendant un appel qui paraît inoffensif de l'extérieur. Recréez les vues à partir de `memory.buffer` après tout appel susceptible de faire croître la mémoire, au lieu d'en conserver une pendant toute la vie de la page.

On attend de Wasm qu'il atteigne la page par lui-même. Un module n'a d'accès implicite à rien : tout effet sur la page, le réseau ou le stockage passe par une fonction que JavaScript fournit en import. Les types imposent leurs contraintes à la même frontière : un `i64` la franchit comme `BigInt`, pas comme une valeur de type Number, et une fonction ayant un paramètre ou un résultat `v128` lève une `TypeError` chaque fois que JavaScript l'appelle. Concevez la liste d'imports et d'exports comme une interface, avec des nombres simples, et gardez les vecteurs du côté Wasm.

Le bac à sable est pris pour un programme sûr. Le bac à sable protège l'hôte du module. Il ne protège pas le module de lui-même. La spécification précise que le bac à sable ne peut empêcher un langage non sûr compilé vers Wasm de

corrompre sa propre organisation mémoire dans la mémoire linéaire : un débordement de tampon dans le code compilé écrase les données du module, et seul un accès au-delà de la fin de toute la mémoire déclenche un trap. Conservez les contrôles et les tests du langage source, et traitez les données reçues de l'extérieur par le module comme des entrées non fiables.

Le binaire est distribué sans protection. L'enregistrement d' `application/wasm` précise que le format n'inclut aucune protection d'intégrité ni de confidentialité et que, lorsqu'elles sont nécessaires, ces protections doivent venir de l'extérieur, par HTTPS par exemple. Comme pour JavaScript, certaines façons de le charger et de l'exécuter sont limitées par la politique de sécurité du contenu ou par la politique de même origine. Puisque Wasm devient du code machine sur du matériel réel, la spécification centrale avertit aussi de sa vulnérabilité potentielle aux attaques par canaux auxiliaires au niveau matériel. Servez les modules par HTTPS et contrôlez qui peut les modifier aussi strictement que pour un script.

Pour aller plus loin

- Spécification WebAssembly — la spécification centrale de la version 3.0 : structure, validation, exécution et formats binaire et textuel.
- Introduction — la définition, les objectifs de conception, le périmètre de chaque document et les considérations de sécurité.
- Interface JavaScript de WebAssembly — compilation et instanciation depuis JavaScript, passage des valeurs, `Memory.grow` et les trois classes d'erreurs.
- API Web de WebAssembly — `instantiateStreaming` , les contrôles de la réponse et l'enregistrement d' `application/wasm` .
- Wasm 3.0 achevé — l'annonce du 17 septembre 2025 et les ajouts de la version 3.0.
- Spécification centrale WebAssembly 1.0 — la recommandation du W3C du 5 décembre 2019.

Glossaire

Choisissez selon vos contraintes réelles

Présentez le processus de travail, les interfaces existantes et les utilisateurs qui ont besoin du changement. Nous pouvons définir une livraison testable et une transmission permettant à votre équipe d'assurer la maintenance des éléments livrés.

Consulter les tarifs Contactez-nous →

Rendez la terminologie concrète

Demandez comment « Qu'est-ce que WebAssembly ? » s'applique à votre organisation. Un exemple concret tiré de votre système actuel suffit pour engager la discussion.

Découvrir nos prestations

welcome@engineer.company +45 50 10 61 84 Copenhague, Danemark

Emportez cette page avec vous

Fiche de contact

Créé avec soin et amour par D.

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

Droit d'auteur © 2025 – aujourd'hui · Engineer ApS · Tous droits réservés.

Imprimé depuis https://engineer.company/fr/glossary/webassembly/