WASI

Introduction

Pour l'exécution côté serveur, on entend bien sûr une exécution très éloignée du navigateur. Nous allons nous concentrer dans ce chapitre sur l'exécution en dehors de celui-ci.

Nous pouvons mettre en évidence deux méthodes d'exécution hors navigateur :

- Avec Node.js, abordé dans un autre chapitre ;
- Et avec la méthode qui nous intéresse, le WebAssembly, qui pourrait s'approcher de la JVM si nous nous limitons à un point de vue conceptuel.

Nous parlerons de la spécification WASI, qui a pour objectif de rendre le WebAssembly un **runtime** portable, c'est-à-dire exécutable sur n'importe quel environnement.

Nous parlerons ensuite bien sûr de ces **runtime** qui respectent la spécification WASI et qui nous permettent de développer des applications serveur compilées en wasm.

WASI qu'est-ce que c'est ?

Tout d'abord, il est important de dire que les personnes travaillant sur les spécifications WASI sont un sous-groupe des personnes travaillant sur le WebAssembly; ce ne sont peut-être pas les mêmes personnes, mais elles gardent un contact étroit.

Les promesses de WASI seront les mêmes que celles du WASM :

- Sécurité
- Polyglotte (Go, Rust, C+C++, etc.)
- Rapide
- Léger

Les fondations de cette spécification, d'un point de vue sécurité, font que les modules respectant les spécifications WASI ne peuvent pas :

- Accéder au système d'exploitation
- Obtenir l'accès à la mémoire que le « host » n'a pas fournie

- Faire des appels sur le réseau
- Lire ou écrire dans des fichiers

WASI est une spécification qui permet d'accéder, en toute sécurité et en toute isolation, au système sur lequel tourne le module WASM.

Nous pouvons alors dire qu'un module WebAssembly s'exécute complètement isolé de la fonctionnalité native du système hôte sous-jacent. Cela signifie que, par défaut, les modules Wasm sont conçus pour effectuer uniquement des calculs purs.

En conséquence, l'accès aux ressources de niveau "OS" telles que les descripteurs de fichiers, les sockets réseau, l'horloge système et les nombres aléatoires n'est pas possible depuis WASM. Cependant, il existe de nombreux cas où un module Wasm doit faire plus que du calcul pur ; il doit interagir avec la fonctionnalité native de l’OS.

Nous constatons que pour créer et déployer des applications serveur, nous avons besoin d'un runtime qui respecte les spécifications WASI. Nous allons donc examiner en détail les trois plus populaires pour vous aider à mieux comprendre leurs avantages et inconvénients et à choisir celui qui convient le mieux à vos besoins.

WasmEdge

WasmEdge est un runtime WebAssembly léger, performant et extensible pour les applications cloud native, edge et décentralisées. Développé par Second State, il alimente les applications sans serveur, les fonctions embarquées, les microservices, les contrats intelligents et les appareils IoT.

Comparé aux conteneurs Linux, WasmEdge peut être jusqu'à 100 fois plus rapide au démarrage, 20% plus rapide à l'exécution et consommer jusqu'à 1/100 de la taille d'une application similaire en conteneur Linux. Cela signifie qu'il est possible de créer des applications plus rapides, plus légères et plus faciles à gérer. WasmEdge offre aux développeurs une plus grande flexibilité et des performances optimisées pour leurs applications, avec des temps de démarrage plus rapides et une consommation de mémoire réduite.

Ce runtime a été optimisé pour l'edge computing et ses principaux cas d'utilisation sont :

- Applications Jamstack, avec un front-end statique et un backend serverless de type FaaS (Function as a Service)
- Automobile
- IOT

WasmEdge fournit également un ensemble d'extensions pour des cas d'utilisation plus avancés, tels que :

- Tensorflow

- Ethereum
- Network sockets

WasmEdge a également décidé d'implémenter le support des sockets en se basant sur la spécification actuelle des WASI-sockets.

Un autre avantage de WasmEdge est qu'il s'agit d'un projet sandbox officiel hébergé par la CNCF.

Wasmtime

Le premier avantage de ce runtime est qu'il est développé par la Bytecode Alliance, qui, comme vous le verrez, est une organisation composée de membres très compétents dans le domaine de l'informatique.

Il permet l'exécution du code WebAssembly en dehors du Web et peut être utilisé à la fois comme un utilitaire de ligne de commande ou comme une bibliothèque intégrée dans une application plus large.

Wasmtime prend en charge un riche ensemble d'API pour interagir avec l'environnement hôte via le standard WASI.

Wamstime utilise le compilateur Cranelift pour tous les cas d'utilisation, ce qui, par rapport à Wasmer, peut entraîner une perte de performance dans certains cas d'utilisation. N'oublions pas que, par rapport aux autres runtimes, celui-ci est développé par la Bytecode Alliance, qui ne tire aucun profit du projet.

Wasmer

La première version de Wasmer est sortie en 1 janvier 2021. D'un point de vue utilisation, Wasmer s'utilise comme une bibliothèque que l'on intègre dans n'importe quel langage de programmation supporté.

Wasmer vous permet d'exécuter des modules WebAssembly de manière autonome ou intégrée dans un grand nombre de langages. Wasmer est conçu pour fournir trois caractéristiques clés :

1. Permettre aux programmes de s'exécuter dans n'importe quel langage de programmation.
2. Permettre à des binaires extrêmement portables de fonctionner sans modification sur n'importe quel système d'exploitation.
3. Agir comme un pont sécurisé pour les modules Wasm afin d'interagir avec les fonctionnalités natives du système d'exploitation, via des interfaces binaires d'application (ABI) telles que WASI.

Pour le premier cas, ils offrent des intégrations pour plusieurs langages, ce qui permet à un module Wasm d'être exécuté à partir de n'importe quel langage de programmation.

Pour le second cas, ils offrent leur Runtime autonome pour exécuter des binaires Wasm sur n'importe quelle plateforme et chipset.

Pour illustrer ses capacités, les développeurs de Wasmer ont compilé et exécuté une version non modifiée du serveur web Nginx, en utilisant des appels WASI pour interagir avec le système hôte.

Pour résumer les caractéristiques principales de Wasmer :

- Pluggabilité : Compatible avec les différents frameworks de compilation, tout ce dont vous avez besoin (par exemple, Cranelift).
- Vitesse/Sécurité : Capable d'exécuter Wasm à une vitesse presque native dans un cadre complètement sandboxé.
- Universalité : Fonctionne sur n'importe quelle plateforme (Windows, Linux, etc.) et n'importe quel chipset.
- Support : Conforme aux normes de la suite de tests WebAssembly avec une large base de soutien de la communauté des développeurs et des contributeurs.

Gros point positif pour ce runtime : il met à disposition un gestionnaire de paquet nommé wapm qui est destiné à héberger des modules WebAssembly incluant des binaires et des bibliothèques universelles.

Bytecode alliance

Pour faire simple, WASM/WASI sont les spécifications du W3C et la Bytecode Alliance s'occupe de leur implémentation. La Bytecode Alliance est un projet visant à créer un environnement d'exécution multiplateforme et multipériphérique basé sur les avantages du WebAssembly. Ses membres fondateurs sont Mozilla, Fastly, Intel et Red Hat, et ils ont été rejoints par d'autres grands noms, tels qu'Amazon, ARM, Docker, Google, Microsoft et d'autres. Ils visent à créer de nouvelles fondations pour nos logiciels en se basant sur les standards WebAssembly et WebAssembly System Interface (WASI).

Les membres de cette alliance contribuent à plusieurs projets open source de l'alliance Bytecode, notamment :

- **Wasmtime**, un runtime léger et performant ;
- **Lucet**, un environnement d'exécution avec compilation hors ligne pour WebAssembly et WASI axé sur les applications à faible latence et à haute concurrence ;
- **WebAssembly Micro Runtime** (WAMR), un environnement d'exécution WebAssembly basé sur un interpréteur pour les périphériques embarqués ;
- **Cranelift**, un générateur de code multiplateforme misant sur la sécurité et la performance, écrit en Rust.

Lequel choisir ?

Le choix va beaucoup dépendre de votre cas d'utilisation et de quand vous allez devoir le faire. En effet, le WebAssembly et encore plus le WASI sont des standards avec des implémentations très récentes. Il faudra donc effectuer un benchmark des performances des runtimes en fonction de vos besoins et des points clés de votre application.

Pourquoi utiliser un runtime serveur WASI plutôt qu'un runtime classique ?

- La portabilité de votre application vous permettra d'exécuter le même code sur n'importe quel système d'exploitation, y compris sur des puces très légères.
- La sécurité du standard WASI vous permettra d'exécuter votre code dans un environnement sandboxé et donc sûr pour l'hôte.
- La taille du binaire compilé sera beaucoup plus petite qu'un conteneur Docker, par exemple.
- La rapidité de démarrage et d'exécution est sans égale :
  - Le WebAssembly est 100 fois plus rapide au démarrage qu'un conteneur très léger basé sur Linux.
  - Le WebAssembly est aussi 10 à 50 % plus rapide à l'exécution.

Pourquoi ne pas toujours le faire si seuls des avantages en découlent ?

- Le WebAssembly côté serveur est une technologie très récente et, par conséquent, elle évolue constamment.
- Si vous n'avez pas de problématique de portabilité, comme l'edge computing par exemple.
- Si les problématiques de sécurité ne sont pas essentielles pour vous.
- En raison de sa jeunesse, le WebAssembly ne bénéficie pas d'une grande communauté, contrairement à Docker par exemple. Il se peut donc que vous n'ayez pas de réponse à certaines de vos questions lors d'un problème rencontré.
- Aujourd'hui, les spécifications WASI ne permettent pas de tout faire. Il est par exemple impossible pour l'instant de faire du vrai multi-threading côté serveur.

Aller plus loin

Ces runtimes ne sont bien évidemment pas seuls et certains runtimes beaucoup plus spécialisés sont apparus, nous vous laisserons jeter un coup d'œil si cela vous intéresse.

“https://hacks.mozilla.org/2019/03/standardizing-wasi-a-webassembly-system-interface/”

Ressources

https://wasi.dev/

Revision #1
Created 2023-05-05 13:42:26 UTC by Noé Larrieu-Lacoste
Updated 2023-05-05 13:43:29 UTC by Noé Larrieu-Lacoste