Webassembly

Sicherere und performantere Codeausführung im Webbrowser

Skriptsprachen im Web sind unsicher und langsam. Durch Sandboxing schafft Webassembly eine maschinennahe, einheitliche Laufzeitumgebung auf Basis eines Bytecodes. Dieser lässt sich aus vielen verschiedenen Programmiersprachen generieren.

Definition

Webassembly ist eine Assembler-ähnliche Sprache in einem kompakten Binärformat, auch Bytecode genannt. Das Gestaltungsziel von Webassembly war es, viele Unzulänglichkeiten, die Javascript grundsätzlich innewohnen, zu beseitigen. Diese Unzulänglichkeiten sind Performanz, Sicherheit und Interoperabilität.

Für Javascript existiert eine Vielzahl von Frameworks, Libraries und Tools. Diese Vielzahl ermöglicht zwar eine hohe Portabilität und Flexibilität; allerdings alles auf Kosten der Performanz. So muss der Webbrowser den ausgelieferten Code rechenintensiv interpretieren. Durch Kompilierung des Javascript-Codes in Webassembly-Bytecode entsteht hingegen ein deutlich schneller und einfacher zu interpretierender Code. Die Ausführungszeit des Codes sinkt und resultiert schlussendlich in einer erheblich gesteigerten **Performanz**. Weitere Leistungssteigerungen sind möglich, weil sich der Bytecode aufteilen und wahlweise auf dem Client oder auf dem Server ausführen lässt.

Die **Sicherheit** adressiert Webassembly durch ein spezielles Sicherheitsmodell. Dieses soll den Anwender einerseits vor fehlerhaften oder bösartigen Modulen schützen. Andererseits soll es nützliche Grundfunktionen und Schutzmaßnahmen für die Entwicklung sicherer Anwendungen bereitstellen. Der Webbrowser lädt den Webassembly-Code deshalb in Form von Modulen und führt diese aus. Eben jene Module lassen sich mit speziellen Compilern für die jeweilige Quellprogrammiersprache generieren. Im Webbrowser läuft jedes einzelne Modul in einer eigenen Sandbox mit seinem eigenen, separierten Speicherbereich. Der Datenaustausch erfolgt per Javascript.

Die **Interoperabilität** steigert Webassembly aufgrund der prinzipiell unbeschränkten Auswahl an Quellsprachen. Aus verschiedenen Programmiersprachen heraus lässt sich Webassembly-Code kompilieren. Dieser kann auf einer gemeinsamen Plattform miteinander kommunizieren und kann den Zwischenschritt über Webservices damit vollständig überspringen.

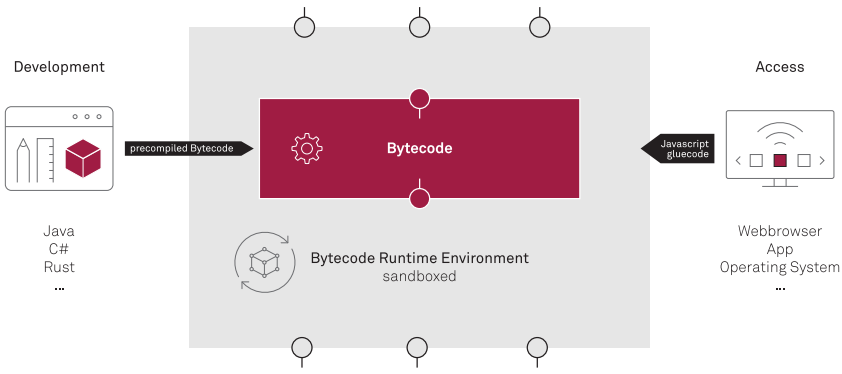
Als Quellsprachen eignen sich bereits Java, C++ oder C#, aber auch Rust oder Go, für die es längst passende Webassembly-Compiler gibt.

Referenzszenario

Das Unternehmen stellt auf webbasierte Betriebsinformationssysteme um. Um bereits vorhandenen Code wiederzuverwenden, überführt es diesen Code in Webassembly. Es trennt dazu den bereits vorhandenen Code fachlich und lagert ihn in einzelne Module mit eigenen Zugriffsparametern aus. Ein auf Javascript basierendes Frontend importiert die Module, der Webbrowser lädt sowie führt die Module aus und das Frontend bindet die Module unter Einsatz von Webassembly-Hooks an sich an.

Potenzial

Webassembly bietet die Möglichkeit, mit einfachen Mitteln bereits vorhandenen, ausentwickelten Code in eine moderne Webanwendung überführen. Dies ist insbesondere für fachlichen Codebestand interessant. Gleichzeitig lassen sich Performanz und Sicherheit bereits vorhandener Webanwendungen erhöhen. Code verschiedener Quellprogrammiersprachen lässt sich dank des Einsatzes von



Performanz

- schnelle Ergänzung zu Javascript hinsichtlich Lade- und Ausführungszeiten
- rechenintensive Applikationen im möglich

Sicherheit

- isolierende Sandbox-Umgebung
- integriert Fremdcode in separate Module

WSM

Interoperabilität

- wachsende Anzahl unterschiedlicher Quellsprachen
- erhöhter Bedarf nach Integrationsoptionen

Portabilität

- bestehenden Code in Webanwendungen übernehmen
- veraltete Frontend-Frameworks ersetzen

Webassembly zusammen und über Module integrieren.

Fachlich getrennte Module bieten eine hohe Wiederverwendbarkeit. Aufgrund der vielen unterstützten Quellprogrammiersprachen können Entwickler ihre gewohnte Programmiersprache weiterverwenden und sehr schnell Resultate erzielen - ohne den hohen Aufwand, sich erst einarbeiten zu müssen. Weil Webassembly-Module lokal auf den Clients laufen können, lassen sich rechenintensive Vorgänge auslagern und Infrastrukturkosten senken.

Reifegrad

Webassembly wurde 2015 vorgestellt; 2016 folgte ein erster Release Candidate. 2019 wurde es offiziell in das World Wide Web Consortium (W3C) eingebunden und ist damit die vierte Webtechnologie, die Code im Browser ausführt. Webassembly steckt zwar noch in der Entwicklungsphase, aber die entscheidenden Webbrowser-Entwickler arbeiten gemeinsam mit der Webassembly Community Group an der Weiterentwicklung.

Marktübersicht

Bereits jetzt kompilieren viele Programmiersprachen ihren Code nach Webassembly, etwa C, C++, Rust, C#, F#, Go, Kotlin, Swift, D, Pascal und Zig. Assemblyscript ist speziell auf die Bedürfnisse von Webassembly ausgelegt. Zudem stehen viele Compiler und Interpreter für Webassembly bereit, etwa Emscripten. Unterschiedliche Laufzeitumgebungen sind bereits auf dem Markt erhältlich. Wasmer containerisiert Webassembly-Code und Laufzeitumgebung, sodass sich dieser überall ausführen und in andere Programmiersprachen einbetten lässt. Auch Package Manager äquivalent zu NPM gibt es.

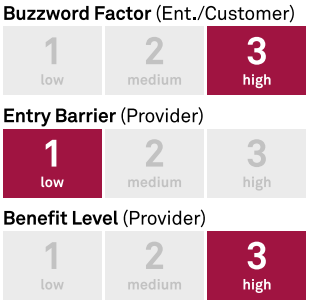
Blazor wiederum ist ein Framework, um interaktive, clientseitige Webbedienschnittstellen mit .NET zu erstellen. Es verzichtet auf den Einsatz von Javascript und führt die in C# geschrieben, nach Webassembly kompilierten Module direkt im Webbrowser aus.

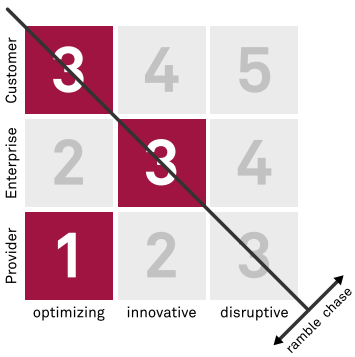
Alternativen

Google Native Client als Sandbox-Technik führt C- oder C++-Code in einem Webbrowser auf Basis von Chromium zusammen. Asmjs ist ein Subset von Javascript, entworfen, um C-Code in Webanwendungen zu nutzen. Es bildet den Aufbau für das Binärformat Webassembly. Um eine gute Performanz zu erreichen, kann lässt sich der Code in ein eigenes Backend auslagern. Aber auch Webworker zurückgreifen, die in einem eigenen Thread laufen, können den Punkt Performanz adressieren.

Fazit

- \+ Anwendungen sind schneller
- \+ WASM-Umgebungen sind sicherer
- \+ lässt sich sowohl im als auch außerhalb eines Webbrowsers ausführen
- \+ rechenintensive Prozesse auf Client auslagerbar
- \- erfordert einen Javascript-Loader
- \- komplexes Programmiermodells
- \- programmiert in Hochsprache
- \- beschränkende Javascript-to-WASM-Schnittstelle





https://msg.direct/techrefresh

Stand: Dezember 2021

msg systems ag

Robert-Bürkle-Straße 1 | 85737 Ismaning/München | Telefon: +49 89 96101-0 | Fax: +49 89 96101-1113 | www.msg.group | info@msg.group