Script

Motivation

- **1995**: Javascript taucht auf
- **2008**: Erster Wendepunkt in der Performanceentwicklung - Just in Time Compiler: Kompilierung und Optimierung während der Runtime
- Die Steigung hat dann mit NodeJS abgenommen, da jetzt auch mehr Performance für die komplexer werdenden Applikationen nötig werden
- **2017** bis heute: Haben wir einen weiteren Wendepunkt der Performance im Web durch WebAssembly erreicht? Diese Frage können wir hoffentlich im Laufe der Präsentation klären

Definition

- Einfach: WebAssembly definiert einen Bytecode, der im Browser ausgeführt werden kann. WebAssembly Bytecode kann das Compilation target von verschiedensten Sprachen sein, was uns erlaubt Code, welcher in andere Sprachen als JavaScript geschrieben ist, im Browser auszuführen.
- Wir stehen also vor einer Welt, in der wir unsere Software über den Browser zum Enduser delivern können mit near native performance, ohne Installation
- WebAssembly ist seit 2019 ein W3C Standard, was dieses Ziel noch viel realistischer macht
- Genauer: Definition von der offiziellen **webassembly.org** Website
- Wir wollen die Definition der Website noch etwas genauer betrachten um die einzelnen relevanten Komponenten run um WebAssembly zu verstehen

Binary Instruction Format

- Format mit Maschineninstruktionen, welches aus 1en und 0en besteht und nach der dekodierung von einer CPU ausgeführt werden kann
- Binaries können grundlegend zweier Natur sein: Maschinencode oder Bytecode, welcher von einer virtuellen Maschine ausgeführt werden kann
- Im Falle WebAssembly redet man von einem Bytecode
- Binaries können verschiedene Instruction Set Architectures als Ziel haben. Euch ist da wahrscheinlich x86, ARM oder RISC-V bekannt
- WASM verwendet allerdings virtuelle instruktionen für eine konzeptionelle Maschine, um am Ende mit möglichst vielen der Instruction Set Architectures kompatibel zu sein
- Man muss sich das WebAssembly Binary Instruction Format also eher als überschneidung mehrerer Instruction Formate vorstellen, welches nicht einfach so auf ein einzelnes gemapped werden kann

Stack-Based Virtual Machine

- Eine Virtuelle Maschine, welches einen Stack anstatt Registers für Operationen verwendet
- Bekannte SVM sind die JVM (Java Virtual Machine) und die CLR (Common Language Runtime, .NET)
- Will man zwei Zahlen in einer Stack-Based-VM addieren, pusht man die beiden Zahlen mit der add Instruktion auf den Stack, die beiden Zahlen werden gepopped und mit ihrer Summe replaced
- Wenn der Browser den WebAssembly Code zum jeweiligen Maschinencode übersetzt, werden allerdings wieder Register verwendet. WASM spezifiziert keine Register, was dem Browser maximale Flexibilität die Register für diese Maschine möglichst Effizient zu allokieren

Portable Compilation Target

- Naja, WebAssembly wurde dazu designed auf verschiedenen Betriebssystemen und Architekturen, im Web oder Nativ zu laufen. Das erklärt eigentlich schon das "portabel"
- Beim kompilieren kann als target WebAssembly ausgewählt werden
- Offener Standard
- Geringe Anforderungen an WASM execution environments:
  - 8-bit Bytes
  - Adressierbare Bytes
  - Little Endian
  - ...

Deployment on the Web Platform

- Web Platform kann in zwei Teilen beschrieben werden
  - VM/Engine, welche den Code ausführt. Beispiele dafür sind V8 in Chrome oder SpiderMonkey in Firefox
  - Web APIs welche weitere Browser- oder Gerätefunktionen freischalten (DOM, WebGL, Web Audio API etc.)
- In der Vergangenheit konnte nur JavaScript im Browser ausgeführt werden
- Seit 2017 haben alle großen Browserengines WASM support eingeführt. Stand von Dezember 2021 war, dass 95% aller installierten Browser WebAssembly supporten.
- Das bedeutet, dass jetzt WebAssembly und JavaScript im Browser ausgeführt werden kann
- Interessant ist, wie wir auch in den Demos sehen werden, dass die Sprachen sich nicht gegenseitig ausschließen, sondern ergänzen. Wir können JavaScript code aus WASM callen und auch WASM code aus JavaScript callen.

Definition

- Wenn wir uns jetzt nochmal die Definition anschauen ist vermutlich klar, was unter WebAssembly zu verstehen ist

Use Cases

- Im Browser:
  - Schnellere Ausführung von Sprachen, welche zuvor zum Web Cross-Compiled werden mussten
  - VR and AR, da diese sehr viel Performance benötigen
  - Simulationen/Emulationen (DOSBox, QEMU, ...)
  - Remote Desktop Solutions
  - Spiele
  - Cloud IDEs, wie Felix ja auch schon angesprochen hat
- Außerhalb des Browsers:
  - Serverseitige Anwendungen
  - Spieleplattformen (wasm binaries lassen sich super verteilen)
- Projekte, welche gerade WASM verwenden:
  - Blazor
  - AutoCAD
  - Figma
  - Jitsi für deren virtuelle Hintergründe
  - Zoom für ihr video encoding
  - D3wasm (Doom 3 im Browser, WebAssembly Demo)
  - go-app (Frontend mit Hilfe von WASM in Go schreiben)

In der WebAssembly spezifikation ist nicht nur das Binary Instruction Format definiert, sondern auch eine Assemblyartige Sprache, mit dem man WebAssembly Module schreiben kann. Und dabei handelt es sich um WebAssemblyText.

WebAssembly Text

- Eine Menschenlesbare Text-Repräsentation von WASM-Binaries
- Konzipiert um in Editoren oder im Browser-Inspektor betrachtet zu werden
- .wat Dateien können mit Hilfe des WebAssembly Binary Toolkits (WABT) zu .wasm kompiliert werden
- Wer sich da noch mehr Informieren möchte, oder auch allgemein zu WebAssembly, die MDN Webdocs sind dafür meiner Ansicht nach die beste Quelle
- Im folgenden ein kleines Beispiel:
  - Wir sehen hier eine simples WASM module mit einer add function, welche zwei Parameter nimmt und diese addiert
  - Module: ist einfach eine stateless WASM Binary mit einer gewissen Menge an Code. Angelehnt an ES2015 Module.
  - WebAssembly Text verwendet S-Expressions, welche ein altes Format ist um Baumstrukturen darzustellen

    - Function signature
    - Function body:
        * Stack Machine - Push zwei Values auf den Stack und addiere mit der i32 add Instruktion
        * i32 add: Poppe die zwei Values und Pushe das Ergebnis
    - Export block: “add” ist der identifier, welcher von JavaScript verwendet wird und $add ist intern, um die zu exportierende Funktion festzustellen
- Diesen nennen wir jetzt ein exportiertes WASM Module mit einer add function
- Unsere exportierte Funktion könnte jetzt so aus JavaScript aufgerufen werden. Diese Funktion kompiliert und instanziiert ein WebAssemembly Module direkt von einer gestreamten Quelle. Das ist der Effizienteste Weg, ein WASM-Modul zu laden, da hier eben streaming verwendet wird und so eben nicht alles auf einmal heruntergeladen werden muss. Will man das machen gibt es eine seperate compile und instantiate methode. Dieses Code snippet, werden wir in den Demos noch öfter sehen

Dann würde ich sagen haben wir erstmal über die grundlegende Spezifikation und Definition von WebAssembly geredet und schauen uns jetzt mal ein paar Demos an, wie wir konkret mit WebAssembly umgehen können.

Demo - Prequisitories

- Kleiner Disclaimer. Die Demo wird primär in Go stattfinden. Teile sind auch in Rust. Der Syntax ist aber sehr trivial, weswegen wir uns dazu entschieden haben Go zu verwenden.

Unterstützte Sprachen

- Es gibt allerdings noch viel mehr Sprachen mit WASM support. Hier eine kleine Aufzählung:
    - C
    - C++
    - Go
    - Rust
    - Zig
    - C#
    - AssemblyScript, ein Typescript-Dialekt ist, der mit WebAssembly im Hinterkopf designed wurde.
- Es gibt noch viel mehr, aber die sind entweder nicht so interessant oder noch nicht bereit für production usage

Dann würde ich sagen, ab zur Demo. Wir beginnen mal mit einem simplen Hello World!

Demo - File Structure

- Unsere Demos werden die Folgende struktur haben. Das ist so ziemlich das minimalste finale Setup, das wir brauchen.
- Wir haben einen assets Ordner mit einer `index.html`, der kompilierten `main.wasm` und der `wasm_exec.js` mit dem JavaScript glue code
- Dann haben wir einen `cmd` ordner mit einem `server` und einen `wasm` Ordner mit unserer zu kompilierenden Applikation

Demo - cmd/wasm/main.go

- Die erste Demo wir ein simples Hello World programm. Es soll uns Hello World in der Browser-Konsole printen
- Die `main.go` Datei im wasm folder ist, wie man sehen kann ein simples Hello World in Go. Keine Besonderheiten

Demo - Compilation

- Diese datei kompilieren wir jetzt zu WebAssembly. Dafür spezifizieren wir bei der kompilierung einfach das target WASM mit den Optionen `GOOS=js` und `GOARCH=wasm`
- Jetzt sollten wir die kompilierte `main.wasm` im assets Ordner finden

Demo - Javascript Glue Code

- Es ist eine gewisse Menge an JavaScript Code nötig, um das WebAssembly Modul, welches wir gerade erstellt haben im Browser zu laden.
- Dieser Code ist nötig, da WASM nicht auf Web APIs zugreifen kann. Daher muss WASM JavaScript callen, welche dann den API call machen. Dafür ist der Glue Code da. In e.g. Emscripten implementiert der glue code Zugriff auf SDL, OpenGL, OpenAL und teile von POSIX.
- Dieser Code ist in der Regel immer bereitgestellt
- In Go erhalten wir den Glue Code aus der Go installation (`wasm_exec.js`)
- Ganz wichtig hierbei ist, dass sich dieser mit den Go Versionen ändert. Wenn also das Program so nicht bei euch läuft, versucht mal die `wasm_exec.js` durch eure eigene auszutauschen.

Demo - assets/index.html

- Die `index.html` included die `wasm_exec.js` Datei, welches der glue code ist
- Im script laden wir das WebAssembly Module wie auch schon im WebAssembly Text Beispiel
- Das importObject enthält dabei die Werte, die in die neue Instanz einfließen sollen, also Funktionen oder `WebAssembly.Memory` Objekte

Demo - cmd/server/main.go

- Hier schreiben wir uns einen kleinen WebServer, welcher unseren `assets` Ordner served

Demo - Execection

- Diesen WebServer können wir ausführen und das Ergebnis im Browser betrachten
- In der Console können wir unser Hello World betrachten, welches aus unserem WASM Module entspringt
- Im Inspektor können wir auch unsere WASM-Datei entdecken und auch den Code in WebAssembly Text-Repräsentation

Die nächsten Demos werden jetzt auf die interoperability abzielen, die ich schon vorher erwähnt hattte. Wir können WebAssembly von JavaScript callen und JavaScript von WebAssembly callen. In der foldenden Demo werde ich nun die Exports präsentieren, die es erlauben WebAssembly-Module in JavaScript zu callen.

Demo Exports

- In dieser zweiten Demo wollen wir eine WASM funktion exportieren und diese dann in JavaScript nutzen.
- Hierfür verändern wir die main.go
- Wir definieren eine eine add function
- Jetzt erstellen wir einen Wrapper mit Hilfe des `syscall/js` packages. Das Package erlaubt die Interaktion mit JavaScript. Dieses Package ist nur legitim, wenn das Compilation-Target WASM ist. Deshalb zeigt das Go LSP dies auch als Fehler an. Func ist eine gewrappete Go function, welche von JavaScript gecalled werden kann
- FuncOf returned eine function, welche von JS gecalled werden kann
- Mit Global erhalten wir ein JavaScript Global Object
- Mit Set setzen wir die value zu einem property string
- Außerdem verändern wir die `index.html` um ein wenig Interaktion mit dem DOM zu haben. Sonst sieht der Code wieder gleich aus
- Jetzt können wir wieder den Server starten und das Ergebnis betrachten
- Wir haben nun ein simples Interface, mit welchem wir unsere im WASM Module definierte `add` function callen können.
- Genauso können wir jetzt add in der Konsole aufrufen

Demo Imports

- Die letzte Demo zeigt uns noch, wie wir auf JavaScript aus unserem WASM Module zugreifen können.
- Hierfür erstellen wir uns eine kleine JavaScript library
  - add function
  - hello function
  - env variable
  - config object
- Die `main.go` greift auf JavaScript mit dem `syscall/js` package zu.
  - Callen add function mit 2 Parametern
  - Printen das Ergebnis. Müssen aber den Typ mitgeben angeben, da JavaScript dynamische Typen hat und es daher theoretisch alles sein könnte.
  - Callen hello
  - Greifen auf die `name` Variable zu
  - Ändern die `name` Variable und schauen den veränderten Wert an
  - Setzen einen key und eine value im `config` Object
  - Betrachten die value zum `key`
- In der `index.html` müssen wir zusätzlich nur die `lib.js` Datei includen

Und schon können wir das Ergebnis im Browser betrachten.

Additional Information

- Es gibt noch weitere Informationen die beim Umgang mit WASM interessant sind
- WASM-Funktionen unterstützen nur Integer und Floats. Mit anderen Datentypen kann über den Linear Memory gearbeitet werden
- Linear Memory - Ein kontinuierlicher Buffer von Bytes, auf welche von JavaScript und WebAssembly zugegriffen werden kann
- DOM Manipulation wird wie schon erwähnt nicht unterstützt. Es muss also über JavaScript laufen. Dafür können aber packages wie `syscall/js` verwendet werden.
- Rust verwendet für WASM das `wasm-bindgen` package. AssemblyScript hat diese features natively, da die Sprache mit WebAssembly im Hinterkopf erstellt wurde

Terminology

- Module und Memory sollten aus den Demos halbwegs klar geworden sein
- Table ist einfach ein typisierter array von Referenzen zu e.g. Funktionen oder anderen Informationen, welche aus Sicherheitsgründen nicht einfach so im Linear Memory gespeichert werden sollten
- Instance: Module inklusive Memory, Table und Imports

Evaluation

- Nachdem wir wissen was WebAssembly ist, wofür wir es brauchen und wie wir damit arbeiten können, können wir mal schauen, was WASM macht, um die von ihnen selbstgesteckten Ziele, welche auf ihrer Website stehen zu erreichen
- Wir fokussieren in dieser Präsentation mal auf “Efficient and fast” und “Safe”, beziehungsweise auf die Security
- Zuerst betrachten wir aber, was WebAssembly so schnell macht, denn WebAssembly verspricht uns near native speed beim ausführen von Code

Efficient and fast

- Zwei Grafiken die die Zeitverteilung von Prozessen bei der ausführung von JavaScript und WASM im Browser betrachten
- Oben JavaScript
- Unten WASM
- Im folgenden werden wir die einzelnen Schritte evaluieren und vergleichen

Efficient and fast - Parsing

- JavaScript code wird zu einem Abstract Syntax Tree geparsed. Jeder der in Algorithmen und Datenstrukturen war oder schonmal einen Interpreter geschrieben hat, kennt das schon
- Dann wird dieser Code zu einer Art Bytecode geparsed
- In diesem Schritt kann man dann auch anfangen WebAssembly zu betrachten, denn wir starten bereits mit diesem Bytecode
- Dieser muss allerdings noch dekodiert und validiert werden

Efficient and fast - Compilation + Optimization

- JavaScript wird während der Ausführung mit einem JIT Compiler kompiliert. Da Javascript dynamic typing besitzt kann es sein, dass einzelnen Schritte mehrmals durchgeführt werden müssen, bis die richtigen Typen gefunden wurden
- WebAssembly spart in diesem Schritt Zeit, da wir die Typen bereits kennen und der Code bereits mit e.g. LLVM optimiert wurde, als die WebAssembly Binary erstellt wurde

Efficient and fast - Reoptimization

- Dieser Schritt findet nur im JavaScript Teil statt
- Die JIT Annahmen, die vorher getroffen worden sind können auch falsch sein e.g. wenn sich Variablen in einem Loop in verschiedenen Iterationen ändern. Die Annahmen könnte gewesen sein, dass dies nicht der Fall ist und der compilierte Code daher einfach mehrmals genutzt werden kann

- Tritt so ein Fehler auf, wir ein “rollback” auf einen älteren Zustand durchgeführt
- WebAssembly benötigt keine Reoptimizerungen, da Typen explizit sind und keine daher keine Optimierungen in diesem Schritt vorgenommen werden müssen

Efficient and fast - Execution

- Die Meisten Entwickler wissen nicht über JIT internals bescheid, daher ist es auch schwer speziell für ihn optimalen Code zu schreiben.
- Im Gegensatz dazu ist WASM als compiler target designed und daher auch dementsprechend performant
- WASM kann in nahezu native speed ausgeführt werden (native - wenn man den Code so bei sich laufen lassen würde). Im Browser können die Programme trotzdem mit 60-70% der Geschwindigkeit ausgeführt werden (aus einem C++ Beispiel)
  - Schlechte register allokierung (Erinerrung: WASM keine Vorgaben)
  - Reservierte Register
  - Schlechte Wahl der Instruktionen
  - Stack overflow checks
  - Indirect function call checks
- Meistens ist trotzdem ein speedup von bis zu 800% im vergleich zu JavaScript möglich

Efficient and fast - Garbage Collection

- JavaScript benutzt Garbage collection, um speicher von unbenutzen Variablen freizugeben
- Wenn man nicht weiß, wann der GC arbeitet, kann es zu unvorhersebarer performance kommen (eher marginale unterschiede)
- WASM hat keine GC, spart sich also diesen Schritt. Es ist aber tatsächlich in der Planung garbage collection hinzuzufügen

Evaluation

- Jetzt haben wir also die Performance von JavaScript und WASM verglichen
- Jetzt sollten wir uns anschauen, was WASM für Security unternimmt

Security

- Bytecode, der in einer Sandbox ausgeführt wird
- WebAssembly-Module werden in der Browser Engine/VM ausgeführt
- Es gelten also die gleichen Sicherheitsbeschränkungen wie auch schon für JavaScript Code
  - Same-Origin Policy

  - Auf Dateisystem oder Hardware kann auch nur mit Permission zugegriffen werden

- Außerdem kann WASM nicht aufs DOM zugreifen
- Aus Securitysicht relevant sind Bytecode und dessen Ausführung. Eigentlich kann man sogar noch weiter reduzieren auf nur die Ausführung, denn der Bytecode kommt i. d. R. von einer nicht vertrauenswürdigen Quelle. Daher müssen wir annehmen, dass dieser komprommitiert ist -> Schutzmaßnahmen müssen also im Browser stattfinden
- Einige Sprachen die WASM als Compilation-Target vorweisen erlauben den Zugriff auf beliebige Speicheradressen. Ein Angreifer könnte das ja potentiell nutzen, um Informationen wie Passwörter oder Authentifizierungstokens auszulesen, sofern er den Speicherort kennt. -> WebAssembly Modulen werden seperate Speicherbereiche zugewiesen
  - Dieser Speicher ist eine dedizierte Untermenge des Heaps der JavaScript VM und wird durch einen ArrayBuffer realisiert. Dadurch gelten für den gesamten Speicher eines WebAssembly Modules die gleichen Beschränkungen wie für normale JS Objekte.
  - Da sich der komplette Heap des WebAssembly-Moduls im ArrayBuffer steckt, weiß die JavaScript Umgebung, wie groß der Heap des Modules ist und wo genau er liegt. Dadurch kann bei jedem Speicherzugriff exakt geprüft werden, ob sich der Zugriff innerhalb des ArrayBuffers bewegt -> Unerlaubte Speicherzugriffe unterbinden
  - Dadurch haben wir zwar etwas Overhead, aber dieser Overhead ist notwendig, da die Ausführung sonst viel zu gefährlich wäre
- Genauso gefährlich wären Zugriffe auf den Execution-Stack. Dieser enthält lokale Variablen und die Rücksprungadressen. WebAssembly verhindert einen Schreibzugriff auf den Execution-Stack, indem er außerhalb des Speichers des WebAssembly-Moduls gespeichert wird -> Schutz des Execution-Stacks
- Generell Sprünge im Code können eine Gefahr darstellen. WebAssembly springt ausschließlich bei Funktionsaufrufen Speicheradressen an. Diese Adressen werden erst dynamisch zur Laufzeit berechnet. -> Tabellen, um Manipulation zu verhindern
  - Statt Zieladresse im call-Befehl verwendet der call Befehl zwei Parameter. Einen Index und eine Funktionssignatur. Der Index zeigt in die Tabelle mit Funktionspointern, die auch außerhalb des WebAssembly-Speichers gespeichert wird, sodass das Überschreiben nicht möglich ist.
  - Prüfung ob Funktionssignatur mit der an Index X übereinstimmt
    * Nur wenn ja, wird die Funktion an der Adresse aufgerufen
    * Wenn nein, wird das Modul sofort gestoppt
- Unsichere Features weglassen, aber Kompatibilität zu C/C++ behalten

  - Alle Funktionen und Datentypen beim Laden deklarieren, auch wenn dynamisch gelinkt wird
    * Dadurch ist Control-Flow-Integrity gewährleistet
  - Funktionsaufrufe über Tabellen
  - Execution-Stack geschützt
  - Branches müssen zu gültigen Zielen in der Funktion führen
- JavaScript Exceptions um abnormales Verhalten an die Laufzeitumgebung zu melden -> Traps
- Trotz dessen will ich euch nicht vorgaukeln, dass WASM die perfekt sichere Lösung ist. Es gab natürlich auch mal Sicherheitslücken. Crypto Mining, Spectre Angriffe und ausbrüche aus der Sandbox sind schon vorgekommen. Es gibt auch interessante side-channel-attacks, aber ich glaube das ist eher was für eine Master oder Doktorarbeit, so wie ich das aus dem Arikel herauslesen konnte.

Jetzt wo wir schon einiges noch ein paar `advanced concepts`. Beginnen wir mit den WebAssembly runtimes.

WebAssembly Runtimes

- WebAssembly überall ausführen, also auch außerhalb des Browsers
- WASM Runtimes sind einfach low level virtual stack machines
- Bekannte Beispiele sind Wasmer, Wasmtime und die WebAssembly Micro Runtime
- Vielleicht kennen manche von euch auch noch Lucit, aber Lucit wurde für `wasmtime` deprecated

Demo - wasmtime

- Installation von rust
- Installation von wasmtime
- WASM target zu Rust compiler hinzufügen
- Compilieren der rust datei zu WebAssembly
- Ausführen der wasm binary

Demo - wasmer

- go verwendet eine wasm binary, welche zuvor in Rust geschrieben wurde
- go run main.go

Wenn wir nun schon bei dem lokalen ausführen von WASM Binaries sind, brauchen wir eigentlich noch einen Weg, um mit dem System zu interargieren.

TODO: WASI überarbeiten ## WASI

- Die folgenden zwei Konzepte sind highly experimental. Daher habe ich auch keine Demos dazu, da diese im Moment noch nicht viel Nutzen werden

- Zitat von Lin Clark
- Mit dem System interagieren, wenn wir WASM außerhalb des Browsers ausführen
- Normalerweise übernimmt die Standard library einer programmiersprache diese Aufgabe und stellt das system interface für die jeweilige Sprache bereit
- Zum Beispiel könnte für die `printf` Function auf einer Windows Maschine die Windows API genutzt werden. Auf einer Linux oder Mac Maschine stattdessen könnte POSIX verwendet werden
- WASM weiß nicht, auf welchem System die Binary ausgeführt wird, daher kann kein vorgefertigtes OS interface verwendet werden
- Realisiert wird das ganze wie bei CloudABI's capability-orientiert, weswegen WASI auch gut in das Sanboxing Modell von WASM passt. WASI hat keine Möglichkeit nach außen zu kommunizieren ohne von außen mitgegebene capabilities e.g. File descriptors.
- E.g. Statt open, openat, hier wird dann ein FileDescriptor benötigt.
  - Es gibt aber trotzdem auf lipreopen basierte open functions
- Es gibt eine system call wrapper layer, welche calls zur eigentlicheen WASI Implementation macht, welche diese calls dann auf die Umgebung mapped
- Ziel damit ist eben portabilität und sandboxing auf system interface level
  - Geplant ist eine full-featured libc implementation

WAGI

- HTTP handlers mit Hilfe von STDIN, STDOUT und Umgebungsvariablen
- Damit können WASM microservices und web apps geschrieben werden
- Grundlegende problem welches gelöst werden soll ist, dass eine WASM binary kein Server ist. Sie kann nicht aktiv listenen, noch läuft sie die ganze Zeit. Also wie bekommen wir HTTP requests
- WAGI-Server verwenden. Dies leitet die HTTP request weiter, indem es die Binary mit STDIN und Umgebungsvariablen mit den nötigen Information versorgt. Die HTTP Response kommt dann über STDOUT und kann wieder von WAGI-Server gehandled werden

Conclusion

- Da WASM keinen direkten DOM-Zugriff hat und daher alle DOM interaktionen über JavaScript laufen müssen, haben wir einen extremen Overhead in dieser Hinsicht.
- Normale Websiten sind daher in JS schneller als in JS + glue code + WASM
- "Right tool for the right job"
- Wenn wirklich performance benötigt wird und nicht aller Gain den Overhead verschwindet, dann lohnt sich WebAssembly allerdings extrem, weswegen auch schon so viele Firmen auf WebAssembly in gewissen Teilen ihrer Anwendungen schwören

- Portability and interoperability
- Here to stay
  - Offener Standard, der Unterstützt und Gefördert wird e.g. von der ByteCode Allegiance, die sich gegründet, um WASM weiterzuentwickeln (ARM, Google, Intel, Microsoft und Mozilla)
  - Krustlet, was die Entwicklung weitertreibt und auch nochmal einen kleine Verbindung zu Felix seinem Teil darstellt, da Krustlet im Endeeffekt erlaubt WASM-Modules statt container images auf Kubernetes Clustern laufen zu lassen

Conclusion

- Um abzuschließen nochmal kurz zur Grafik vom Anfang
- Ich glaube sehr wohl, dass wir einen weiteren Wendepunkt in der performance im Web erreicht haben
- Und ich glaube auch, dass es in Zukunft nicht mehr so sein wird, dass WASM JS benötigt, aber JS WASM nicht unbedingt benötigt, sondern dass auch JavaScript WASM benötigen wird, da wie auch schon bei NodeJS einfach die Anforderungen an Web Applikationen steigen werden und daher nicht mehr auf den Performancezuwachs durch WASM verzichtet werden kann.

Webshop

- Zum Abschluss noch mein Web Shop
- Web Shop mit go-app erstellt, einem Go package um web apps zu erstellen
- HTML kann deklarativ in Go geschrieben werden