HAW HAMBURG

BACHELORTHESIS
Malte Scheller

Evaluierung der Technologie WebAssembly in einer browserbasierten Anwendung

FAKULTÄT TECHNIK UND INFORMATIK
Department Informatik

Faculty of Computer Science and Engineering
Department Computer Science

HOCHSCHULE FÜR ANGEWANDTE WISSENSCHAFTEN HAMBURG
Hamburg University of Applied Sciences

Malte Scheller

Evaluierung der Technologie WebAssembly in einer browserbasierten Anwendung

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung
im Studiengang *Bachelor of Science Angewandte Informatik*
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Olaf Zukunft
Zweitgutachter: Prof. Dr. Stefan Sarstedt

Eingereicht am:

Malte Scheller

Thema der Arbeit

Evaluierung der Technologie WebAssembly in einer browserbasierten Anwendung

Stichworte

WebAssembly, Webanwendung, Evaluierung, browserbasiert, angular, wasm, Rust, AssemblyScript

Kurzzusammenfassung

Ziel dieser Arbeit war die Evaluierung der Technologie WebAssembly. Zu diesem Zweck wurde eine Webanwendung mit mehreren Benchmark-Funktionen entwickelt. Diese Benchmark-Funktionen wurden jeweils in JavaScript, Rust und AssemblyScript implementiert, so dass die Ausführungsgeschwindigkeit der verschiedenen Implementationen miteinander verglichen werden konnten. Diese Vergleiche haben gezeigt, dass WebAssembly in den meisten Fällen eine wesentlich bessere Performanz aufweisen kann.

Malte Scheller

Title of Thesis

Evaluation of the WebAssembly technology in a browser-based application

Keywords

WebAssembly, web application, evaluation, browser-based, angular, wasm, Rust, AssemblyScript

Abstract

The aim of this work was to evaluate the technology WebAssembly. For this purpose, a web application with several benchmark functions was developed. These benchmark functions were implemented in JavaScript, Rust and AssemblyScript respectively, so that the execution speed of the different implementations could be compared with each other. These comparisons have shown that in most cases WebAssembly can show a significantly better performance.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Listings

Glossar

Look and feel 1 Design-Aspekte auf Websites oder Anwendungen mit grafischer Oberfläche

Ajax 4 Konzept der asynchronen Datenübertragung zwischen einem Browser und dem Server

Engines 4 Eigenständiger Teil eines Programms, der meist im Hintergrund läuft

Cern 3 Das Cern ist eine Großforschungseinrichtung in der Nähe von Genf

CPU 9 Prozessor eines Computers. Er bildet die zentrale Recheneinheit.

Heap Corruption 13 Wenn der Inhalt eines Speicherorts aufgrund eines programmatischen Verhaltens geändert wird und dies die Absicht des ursprünglichen Programmierers überschreitet

Race Conditions 21 Tritt auf, wenn zwei oder mehr Threads auf gemeinsame Daten zugreifen können und sie versuchen, diese gleichzeitig zu ändern

DOM-Tree 30	Struktur der HTML-Tags, aus denen jede Website besteht
Subscription 49	Abonnement für ein bestimmtes Event, sodass jedes Mal eine Benachrichtigung erhalten wird, sobald dieses eintritt
Properties 49	Properties sind die Werte, die mit einem JavaScript-Objekt verbunden sind
View 38, 39, 44, 49, 60, 61, 64, 70	Grafische Ansicht eines Elements
Event 49, 62, 65	Konzept der „Event-driven Architecture“. Bei der Komponenten einer Software durch Ereignisse (Events) kommunizieren, die Erzeugt werden, sobald etwas passiert. Beispiel: „Benutzer klickt auf Button“
Workspace 58	Ein Arbeitsbereich
Angular-CLI 58	Ein Werkzeug mit Befehlszeilenschnittstelle, mit dem Angular-Anwendungen direkt von einer Konsole aus initialisieren, entwickelt und gewartet werden kann
BASH-Skripte 58	Kleine Programme, die meist mit Hilfe der Konsole ausgeführt werden
Booten 59	Starten eines Programms
Bauen 59	Erstellungsprozess, bei dem eine Anwendung erzeugt wird

Pattern 60 Ein bestimmter Lösungsweg, für ein wiederkehrendes Entwurfsproblem

deep copy 69 Die Kopie eines vorliegenden Objekts. „Deep“ da nicht nur die Referenzen eines Objekt kopiert werden, sondern die Werte auf denen referenziert wird

Selektoren 64 Ein HTML-Tag, welches in einer View platziert werden kann. Es ist beispielsweise einer bestimmten Angular-Komponente zugordnet und bewirkt, dass die entsprechende Komponente instanziiert wird

DOMHighResTimeStamp 69 Der Typ DOMHighResTimeStamp ist ein Double und wird verwendet, um einen Zeitwert in Millisekunden zu speichern. Dieser Typ kann zur Beschreibung eines diskreten Zeitpunkts oder eines Zeitintervalls (die Zeitdifferenz zwischen zwei diskreten Zeitpunkten) verwendet werden. [1]

SIMD 94 Single Instruction Multiple Data (SIMD) ist eine Rechnerarchitektur mit Vektorprozessoren, bei der viele Prozessoren ihre Befehle von einem Befehlsprozessor erhalten und gleichzeitig unterschiedliche Daten verarbeiten. [2]

1 Einleitung

1.1 Motivation

Websites sind heutzutage nicht mehr nur Internetauftritte mit statischem Inhalt, vielmehr haben sie sich zu komplexen Webanwendungen entwickelt, die den klassischen Desktopanwendungen im Look and Feel nahezu ebenbürtig sind. Der wohl größte Vorteil dieser Webanwendungen ist, dass sie unabhängig vom Betriebssystem, auf jedem Endgerät, auf dem ein Webbrowser installiert ist, ausgeführt werden können. Mit den von den meisten Entwicklern gut vertrauten Technologien der Webentwicklung lässt sich so beinahe für jeden Anwendungsfall Software schreiben, die mit derselben Codebasis mit jedem System kompatibel ist. Bisher bestand jedoch auf diesem Gebiet ein wesentlicher Nachteil. Aufgrund fehlender Performanz können diese Anwendungen nicht ernsthaft mit den nativen Anwendungen konkurrieren. Der Grund dafür ist, dass die in der Webentwicklung verwendete Programmiersprache JavaScript bei weitem nicht so performant ist, wie die nativen Sprachen.

1.2 Zielsetzung

Ziel dieser Arbeit ist es, die neue Technologie WebAssembly vorzustellen und zu evaluieren. Diese Technologie zählt seit 2019 zu einer der vier offiziellen Programmiersprachen der Webentwicklung [3] und wird als revolutionäre Entwicklung angesehen [4]. Mit ihr soll es möglich sein Webanwendungen in nahezu nativer Geschwindigkeit auszuführen. Es werden zuerst die bisher bestehenden Technologien der Webentwicklung aufgezeigt, sowie die dort existierenden Leistungsgrenzen und Engpässe. Anschließend wird eine Testanwendung entwickelt, die es ermöglicht die Leistungsunterschiede aufzuzeigen, die WebAssembly mit sich bringt.

Durch die mit dieser Anwendung ermittelten Ergebnisse wird dann evaluiert, ob und wann sich der Einsatz von WebAssembly lohnt.

1.3 Gliederung

In Kapitel 2 dieser Arbeit werden zunächst die Grundlagen behandelt. Es wird dabei ein grundlegendes Wissen über die relevanten Themen vermittelt werden. Mit Kapitel 3 folgt die Anforderungsanalyse. Dort werden die Anforderungen definiert, welche die zu entwickelnde Anwendung erfüllen muss. In Kapitel 4 werden die Entwurfsentscheidungen behandelt. Aufgezeigt werden die wichtigsten Entscheidungen, die aufgrund der definierten Anforderungen getroffen wurden. Kapitel 5 beschreibt die Implementierung dieser Entwurfsentscheidungen und gewehrt dabei durch ausgewählte Codebeispiele einen genaueren Einblick in die Anwendung. Die Evaluierung der Technologie WebAssembly folgt im 6. Kapitel. Dort werden die mit der Anwendung gemessenen Werte ausgewertet und dabei Schlussfolgerungen angestellt. Darüber hinaus wird WebAssembly in diesem Kapitel auf Basis der Erfahrungen, die während der Implementierung gemacht wurden, bewertet. Im 7. Kapitel erfolgt eine Zusammenfassung der Arbeit sowie ein Ausblick auf die zukünftige Entwicklung der Technologie WebAssembly. Basierend auf diesen Entwicklungen, werden zusätzlich Themen aufgezeigt, die in weiterführenden Arbeiten folgen könnten.

2 Grundlagen

In diesem Kapitel werden Grundlagen behandelt, die zum besseren Verständnis dieser Arbeit beitragen. Zunächst werden die bisher in der Webentwicklung verwendeten Sprachen behandeln. Anschließend werden die Grundlagen der Technologie WebAssembly erläutert, sowie die für diese Arbeit gewählten Programmiersprachen. Zum Ende dieses Kapitels, wird der Begriff Performanz näher beleuchtet, es wird erklärt, was Performanz in der Informatik überhaupt bedeutet und wie sie gemessen werden kann.

2.1 HTML, CSS, JavaScript

Bisher zählte das W3C[1] die drei Sprachen HTML[2], CSS[3] und JavaScript zum offiziellen Webstandard. HTML ist eine Auszeichnungssprache. Sie wurde 1990 und damit schon in einer frühen Epoche des Webs, von Tim Berners-Lee entwickelt [5]. So wie das Web selbst, wurde es am Cern entwickelt und sollte dazu dienen Dokumente, wie Forschungsergebnisse zwischen Mitarbeitern und verschiedenen Einrichtungen auszutauschen [6]. Es stellt dabei eine einfach zu verstehende Textauszeichnungssprache dar, mit der es ermöglicht wurde, die Inhalte dieser Dokumente wie Texte, Bilder oder Hyperlinks zu strukturieren. HTML bildet heute die Grundlage des World Wide Webs und wird von allen Webbrowsern bereits in der Version 5.2 unterstützt [7]. Um neben einer semantischen Strukturierung auch eine Formatierung der Dokumente zu ermöglichen, wurde die Stylesheet-Sprache CSS entwickelt. Mit CSS wurde es

[1] World Wide Web Consortium

[2] Hypertext Markup Language

[3] Cascading Style Sheets

möglich, neben der Strukturierung auch die Darstellung der Inhalte vorzugeben. So ist es möglich jedes einzelne Element eines HTML-Dokumentes zu referenzieren und z.B. seine Farbe oder Typografie zu modifizieren. So wie HTML, hat sich CSS durchgesetzt und existiert heute bereits in der Version 3 [8].

Um den statischen HTML-Dokumenten mehr Dynamik zu verleihen, entwickelte die Firma Netscape in Kooperation mit der Firma Sun Microsystems, die Skriptsprache JavaScript [9]. JavaScript sollte es dem Benutzer ermöglichen in größeren Umfang mit den vom Browser dargestellten Internetseiten zu interagieren. Schon in den frühsten Versionen war so z.B. die Überprüfung von Formulareingaben, das Verändern von Elementen, wie Texten, Bilder und diversen Animationen verfügbar [9, p. 457]. Das Potential dieser Sprache wurde auch von anderen Firmen erkannt, sodass diese schon bald Browser mit eigenen JavaScript-Engines auf den Markt brachten. Dies führte Mitte der 1990er zu einem jahrelangen Verdrängungswettbewerb zwischen Microsoft und Netscape, der als Browserkrieg bekannt wurde [10, p. 3]. JavaScript erlebte dadurch einen Entwicklungsboom und wurde vor allem durch Erscheinen der Technologie Ajax im Jahr 2005 und die daraus entstandenen Frameworks und Bibliotheken zu einer der meistgenutzten Programmiersprachen der Welt [10, p. 4].

Diese Entwicklung hat dazu geführt, dass JavaScript die einzige Programmiersprache ist, die als offizieller Standard in der Webentwicklung verwendet wird. Dies hat den großen Vorteil, dass der Code nur einmal geschrieben werden muss und auf allen Geräten, auf denen ein Browser installiert ist, ausgeführt werden kann. Ein entscheidender Nachteil ist, dass JavaScript ursprünglich als einfacher und kurzlebiger Code gedacht war, der Komponenten miteinander verbindet [11, p. 13]. Durch die fortlaufende Entwicklung des Webs allerdings, existieren nun Webanwendungen, die komplexe Prozesse ausführen und deren Quellcodes oftmals tausende Zeilen lang sind. Vor allem mit der Einführung von Single-Page-Anwendungen hat der Trend zu langlebigen, komplexen JavaScript-Algorithmen stark zugenommen [12, p. 6]. Das Internet, das ursprünglich nur zum Anzeigen einfacher Texte und Bilder gedacht war, hat sich zu hoch interaktiven Websites entwickelt. Diese Websites werden daher mittlerweile meist Webanwendungen genannt, da sie bereits dieselbe Funktionalität wie viele Desktopanwendungen aufweisen [12, p. 7].

Im Laufe der Zeit konnten etliche Optimierungen an JavaScript vorgenommen werden. Performante Engines, wie die V8 Engine von Google oder SpiderMonkey von Mozilla verleihen Entwicklern die Möglichkeit, rechenintensivere Prozesse, wie Machine Learning oder Videospiele in JavaScript zu implementieren [12, pp. 15-16]. Der größte Nachteil bleibt allerdings bestehen, JavaScript ist eine schwach typisierte, dynamische Interpretersprache. Interpretersprachen lassen sich dadurch charakterisieren, dass der von den Entwicklern geschriebene Code in derselben Form, in der er geschrieben wurde, an die Klienten übertragen und erst bei Ausführung von einem Interpreter in den für die CPU verständliche Maschinensprache übersetzt wird. Was zuvor ein Vorteil war, wirkt sich vor allem in komplexen Algorithmen negativ aus. Da Interpretersprachen erst zur Laufzeit (on-the-fly) in Maschinencode übersetzt werden, entsteht vor der Ausführung keine Wartezeit. Dies steht im Gegensatz zu Compilersprachen, die vor der Ausführung erst kompiliert werden müssen. Interpretersprachen haben jedoch dadurch das Problem, dass der geschriebene Code jedes Mal erneut übersetzt werden muss, wenn dieser ausgeführt werden soll. Enthält der Algorithmus z.B. eine Schleife, so wird der Code bei jeder Ausführung der Schleife erneut übersetzt. Um diesem Problem entgegenzuwirken, wurde die Just-in-time-Kompilierung entwickelt. Diese bewirkt eine erhebliche Verbesserung der Performanz.

2.1.1 Bisherige Optimierungen

Just-In-time-Compiler

Wie schon erwähnt, gibt es in der Programmierung im Allgemeinen zwei Möglichkeiten den geschriebenen Code in Maschinensprache zu übersetzen. So ist es möglich zur Übersetzung einen Compiler oder einen Interpreter zu verwenden.

Mit einem Interpreter geschieht die Übersetzung Zeile für Zeile, was auch on-the-fly genannt wird. Ein Compiler hingegen arbeitet ahead-of-time. Dies bedeutet, dass er das komplette Programm vor der Ausführung in Maschinencode übersetzt [12, p. 8].

Der Vorteil eines Interpreters ist, dass nicht erst der ganze Kompilierungsschritt durchlaufen werden muss bis der Code ausgeführt werden kann. Es entstehen keine Wartezeiten vor der

Ausführung. Für Webentwickler ist diese Eigenschaft schon immer besonders wichtig gewesen, deshalb ist JavaScript auch die Sprache der Wahl für diesen Bereich. Das steht im Gegensatz zum Compiler, der den Code vor der Ausführung erst komplett in Maschinencode übersetzt. Dies nimmt eine gewisse Zeit in Anspruch, ist dafür aber anschließend viel performanter, da der Compiler während des Kompilierens auch Optimierungen am Code vornehmen kann [13].

Der sogenannte Just-In-Time-Compiler sollte die besten Eigenschaften aus Compiler und Interpreter kombinieren. An den Stellen, an denen der Code mehrmals durchlaufen wird, beginnt der Browser, wie ein Compiler zu arbeiten. Dazu kommt ein sogenannter Monitor zum Einsatz, der den Code überwacht [14]. Wird eine Stelle im Code mehrmals durchlaufen, wird diese als „warm“ bezeichnet. Der Monitor merkt sich diese Stelle im Code. Sobald sie eine gewisse Anzahl an Durchläufen erreicht hat und alle darin enthaltenen Variablen denselben Typ behalten haben, kommt der Baseline-Compiler zum Einsatz. Der Baseline-Compiler übersetzt den Code zu einem effizienteren Maschinencode. Wird diese Stelle noch öfter durchlaufen, ohne, dass es Veränderungen gibt, kommt der Optimizing-Compiler zum Einsatz. Dieser nimmt nochmals diverse Optimierungen am Code vor, was die Ausführungsgeschwindigkeit der Anwendung nochmal um ein Vielfaches steigern kann [14].

Aufgrund der dynamischen Natur von JavaScript gibt es allerdings auch hier einen entscheidenden Nachteil. Da die Datentypen hier erst zur Laufzeit geprüft werden, kann nie ganz sicher vorausgesagt werden, welchen Typ eine Variable annehmen wird. So kann es vorkommen, dass in 1 von 100 Durchlaufen, eine Variable doch einen anderen Datentyp annimmt, als der Monitor angenommen hat. In so einem Fall muss eine Deoptimierung vorgenommen werden, die den Code wieder zurück zu JavaScript-Code übersetzt. Anschließend wird der Code erneut überwacht und möglicherweise erneut optimiert. Diese Prozesse nehmen viel Zeit in Anspruch, was sogar dazu führen kann, dass Anwendungen durch die JIT-Kompilierung langsamer laufen [14].

Um dieser dynamischen Natur entgegenzuwirken, wurde von Mozilla die Technologie asm.js entwickelt.

asm.js

Asm.js ist eine strikte Untermenge der Sprache JavaScript und ermöglicht die Ausführung von C oder C++ Code in Webanwendungen. Die Entwickler entwickeln nicht direkt in dieser Sprache, sondern implementieren ihre Programme zuvor in C oder C++ und konvertieren den Code anschließend mit einem speziellen Compiler zu asm.js. Durch weitere Techniken, die diese Technologie bietet, können die Programme mit deutlich besseren Leistungseigenschaften ausgeführt werden als Programme, die in purem JavaScript geschrieben wurden [12, pp. 4-5]. Zusätzlich wird der Code hier mit Statements versehen, durch die der ausführende Browser weiß, dass er an der entsprechenden Stelle Low-Level-Operationen[4] verwenden kann, anstatt teurere JavaScript-Operationen. Mit sogenannten Type-Hints kann in asm.js sogar der Typ einer Variable festgelegt werden. Dadurch wird der Laufzeitumgebung zugesichert, dass sich der Datentyp der Variable niemals ändern wird. Der Code muss so nicht mehr lange vom Monitor überwacht werden, sondern kann sofort zu Maschinencode kompiliert werden [12, pp. 4-5]. Dies führt zu einem Geschwindigkeitszuwachs gleich zu Beginn der Ausführung.

Aber auch asm.js bringt ein paar entscheidende Nachteile mit sich, die eine Trendwende in der Webentwicklung verhindert haben [12, p. 5]:

- Die zusätzlichen Einträge im Code führen zu sehr großen Dateien.
- Die asm.js-Dateien müssen trotzdem noch von der JavaScript-Engine gelesen und geparst werden
- JavaScript war nie dazu gedacht ein Compiler-Ziel zu sein, was zu weiteren Problemen führt. Um z.B. zusätzliche Features hinzufügen zu können, müssten die Entwickler die Sprache JavaScript modifizieren, was nicht ohne weiteres möglich ist

[4] maschinennahe Befehle

2.2 WebAssembly

Das Projekt WebAssembly wurde als Nachfolger von asm.js ins Leben gerufen. An der Entwicklung waren die Firmen der bekanntesten Browser-Engines[5] beteiligt [15]. Es wurde mit dem Ziel entwickelt, alle positiven Aspekte von asm.js aufzugreifen und gleichzeitig deren Mängel zu beheben. Im weiteren Verlauf der Arbeit, wird für WebAssembly oft die offizielle Abkürzung Wasm verwendet [12, p. 5].

WebAssembly ist ein binäres Befehlsformat für eine Stack-basierten, virtuelle Maschine. Es wurde speziell als portables Kompilierungsziel für Programmiersprachen konzipiert und ermöglicht die Bereitstellung im Web für Client- und Serveranwendungen [16]. Das Befehlsformat ist ein sicheres, assemblerähnliches Low-Level-Codeformat, was für eine effiziente Ausführung und kompakte Darstellung entwickelt wurde [17]. Hauptziel ist es, Hochleistungsanwendungen im Web zu ermöglichen. WebAssembly ist ein offener Standard, der von einer W3C Community Group entwickelt wurde und schon seit 2017 von allen modernen Browsern unterstützt wird. Seit dem 5. Dezember 2019, gehört WebAssembly außerdem als vierte Sprache zum offiziellen Webstandard [3].

Anders als asm.js, handelt es sich bei WebAssembly nicht um eine weitere Sprache, die zu JavaScript konvertiert wird. Ein WebAssembly-Programm besteht aus in Bytecode gespeicherten, virtuellen Instruktionen. Meist wird so ein Programm Wasm-Modul genannt und besteht aus nur einer *.wasm*-Datei. Obwohl nicht direkt in diesem Format entwickelt wird, kann dieser WebAssembly-Bytecode im weitesten Sinne sogar als Programmiersprache bezeichnet werden [11, p. 5].

[5] Chrome, Edge, Firefox, Safari

2.2.1 Erzeugen eines Wasm-Moduls

Um ein Wasm-Modul zu erzeugen und in einer Webanwendung nutzbar zu machen, muss der im Modul vorhandene Algorithmus zuvor in einer unterstützten Quellsprache geschrieben werden. Browser kommen auf einer Vielzahl von unterschiedlichen Geräten bzw. CPUs zum Einsatz. Da all diese Browser die Wasm-Module unterstützen und somit den Bytecode verstehen müssen, unterscheidet sich der Weg von diesem Punkt an von dem einer nativen Anwendung. Da jeder CPU-Typ seinen eigenen Dialekt von Maschinensprache versteht, müssen die entwickelten Anwendungen durch Compiler zu jedem dieser Dialekte kompiliert werden, damit sie von den entsprechenden CPUs ausgeführt werden können. Üblicherweise besteht so ein Compiler, wie in der Abbildung (Figure 1) zu sehen ist, aus einem Frontend und einem Backend. [13]

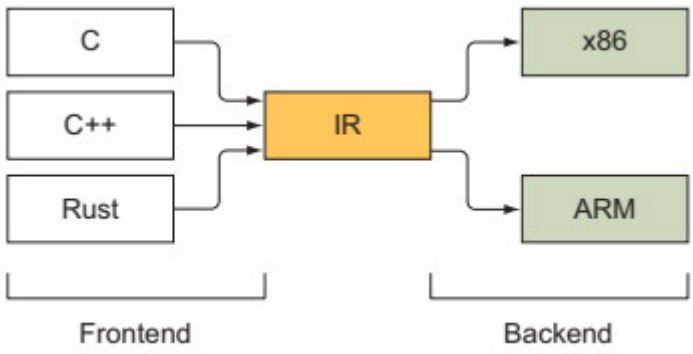

Figure 1 - Front-End und Back-End eines Compilers mit der IR als Zwischenschritt (Gallant, 2019)

Über diverse Zwischenschritte, in denen der geschriebene Code auch optimiert und in eine Zwischendarstellung, der sogenannten IR[6], gebracht wird, entsteht im Backend des Compilers, der speziell für den CPU-Typ verständliche Maschinencode. Da bei einem Wasm-Modul zuvor nicht bekannt ist, auf welcher Art CPU es ausgeführt werden soll, kann hier nicht direkt zu Maschinencode kompiliert werden. Stattdessen wird noch ein weiterer Schritt in den

[6] intermediate representation

Kompilierungsprozess eingefügt. Die IR wird hier von einem speziellen Compiler-Backend zu dem Wasm-Bytecode kompiliert (Figure 2) [13].

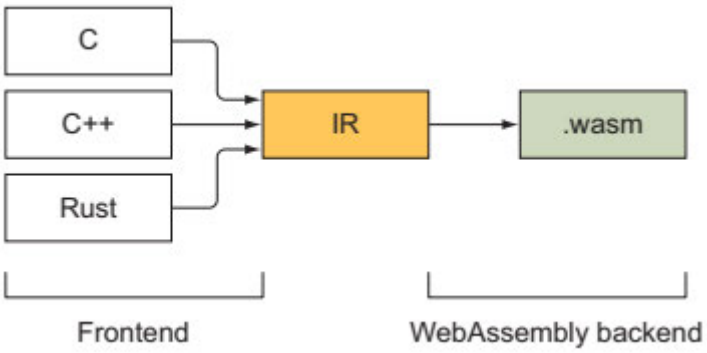

Figure 2 - Kompilieren der IR zu Wasm-Bytecode durch speziellen Compiler (Gallant, 2019)

Bei diesem Bytecode handelt es noch nicht um den finalen Maschinencode. Er stellt einen Satz virtueller Instruktionen dar, die ein Browser der WebAssembly unterstützt, verstehen kann. Diese Art von Code ist bereits optimiert und besitzt eine wesentlich geringere Abstraktion gegenüber dem Maschinencode. Diese Instruktionen bilden das Wasm-Modul und können anschließend von einem Browser geladen werden. Dort wird die Gültigkeit des Moduls verifiziert und letztendlich zu dem passenden Maschinencode des Gerätes kompiliert. [12, p. 9]

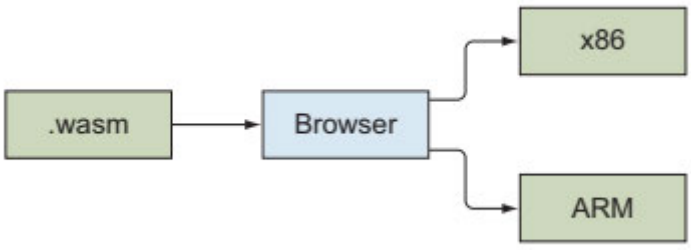

Figure 3 – Kompilieren des Bytecodes zu Maschinencode im Browser [12]

2.2.2 Vorteile

Das WebAssembly-Design bietet gegenüber JavaScript diverse beachtliche Vorteile. Wie bereits beschrieben, ist JavaScript eine dynamisch getypte Interpretersprache, die on-the-fly vom Interpreter interpretiert wird. WebAssembly ist eine statisch getypte Sprache, die ahead-of-time vom Compiler kompiliert wird. Das bedeutet, dass das Modul bereits bei Beginn der Ausführung im effizientem Maschinencode vorliegt. Die statische Typisierung ermöglicht eine optimierte Ausführung von Anfang an, da Teile des Codes nicht erst, wie bei JavaScript, einige Zeit überwacht werden müssen, bevor sie optimiert werden können.

Dadurch, dass WebAssembly bereits in einem Bytecode vorliegt, sind die Wasm-Module sehr viel kompakter, was die Übertragungs- und Downloadzeiten gegenüber JavaScript erheblich verringert. Die Module sind außerdem so strukturiert, dass sie parallel und in kurzer Zeit validiert werden können. Zusätzlich werden durch die sogenannte Streaming-Compilation Module bereits während des Downloadvorgangs kompiliert, was den Start der Ausführung nochmals beschleunigt. [12, p. 27]

Die Auswahl an verwendbaren Sprachen, die zu WebAssembly kompiliert werden können, nimmt stetig zu. Neben C und C++ können auch Rust, C#, Go, Kotlin, Swift und viele mehr verwendet werden [18]. Das ermöglicht nicht nur die Wiederverwendbarkeit von Code, es können nun auch nützliche Bibliotheken, die in diesen Sprachen geschrieben wurden, in Webanwendungen eingesetzt werden. Eine bisherige Hürde bei der Verwendung der Sprachen stellt der fehlende Garbage Collector dar. Die Speicherverwaltung muss in WebAssembly manuell vorgenommen werden. Das bringt zwar zusätzliche Performanz-Vorteile, schränkt aber die Auswahl der Sprachen ein [12, p. 14].

2.2.3 Nachteile

WebAssembly kann bereits in vielen Fällen durch bessere Performanz überzeugen, doch ist die Technologie nicht in jedem Anwendungsfall schneller als JavaScript. Es existieren bereits Performanz-Tests, bei denen WebAssembly schlechter abschneidet [19]. Das liegt daran, dass WebAssembly als maschinennahe Sprache nur vier primitive Datentypen besitzt [20]. Werden

Anwendungsfälle getestet, bei denen viele Operationen auf Strings ausgeführt werden müssen, schneidet WebAssembly meist schlechter ab. Das liegt daran, dass diese Strings zuvor in WebAssembly-kompatible Datentypen konvertiert werden müssen, bevor diese an das Modul gesendet werden können.

Des Weiteren wird WebAssembly in der JavaScript-Sandbox ausgeführt. Dies bringt zwar Vorteile in der Sicherheit mit sich, bremst aber die Module in der Ausführung aus. Durch die Ausführung in der Sandbox ist es WebAssembly außerdem nicht möglich, auf das Dateisystem des Hosts zuzugreifen. Dies kann in einigen Fällen die Arbeit mit den Modulen erschweren. [11, pp. 5-6]

Auch wenn WebAssembly mit einer Ausführungsgeschwindigkeit in nahezu nativer Geschwindigkeit wirbt, so hat es noch den Nachteil, dass in der momentanen Version 1.0 weder Multithreading noch SIMD unterstützt werden. Diese Technologien werden von den meisten nativen Sprachen unterstützt und bieten entscheidende Schübe in der Performanz.

2.2.4 Architektur

Bisher war es so, dass Technologien, die eine Erweiterung zu JavaScript schaffen wollten, in eigenen Sandbox-VMs laufen mussten. Die konnte zu diversen Problemen führen, vor allem in der Sicherheit. Zum ersten Mal überhaupt, wird die JavaScript-VM geöffnet, damit WebAssembly-Code in der gleichen VM ausgeführt werden kann. Dies hat viele Vorteile. Der größte davon ist vermutlich, dass die VM im Laufe der Jahre intensiv getestet und gegen Sicherheitslücken abgehärtet wurde. [11]

Wie bereits zu Anfang des Kapitels beschrieben, handelt es sich bei WebAssembly um ein binäres Befehlsformat, welches von einer Stack-basierten, virtuellen Maschine ausgeführt wird.

Stack-Machine

Handelsübliche PCs werden, genauso wie Laptops oder Smartphones, als Register Machines bezeichnet. In einer Register Machine referenzieren die Instruktionen der CPU direkt Register oder Datenspeicherorte der CPU. Diese Methode ist schnell und effizient, da die Daten dort direkt abgreifbar sind. Dadurch unterscheiden sie sich von Stack-basierten Maschinen bei denen erwartet wird, dass die Operanden einer Operationen direkt auf dem Stack liegen. Muss zum Beispiel eine Additions-Operation ausgeführt werden, so müssen, wie in Listing 1 gezeigt, die entsprechenden Parameter vorher auf den Stack gelegt und anschließend die Additionsoperation ausgeführt werden. Diese Operation weiß, wie viele Parameter sie benötigt und holt sich diese automatisch vom Stack. Dies bringt gewisse Vorteile mit sich, u.a. eine stark verringerte Binärgröße, eine effiziente Befehlscodierung und eine leichte Portierbarkeit. [13]

```
1 // hole Wert des ersten Parameters und packe ihn auf den Stack
2 get_local 0
3
4 // packe Konstante auf den Stack
5 i32.const 42
6
7 // addiere die ersten zwei Werte und packe das Ergebniss auf den Stack
8 i32.add
```

Listing 1 - Beispiel für das Addieren zweier Werte in einer Stackmachine

Es bleibt allerdings anzumerken, dass auch wenn WebAssembly als Stack-basierte Maschine spezifiziert ist, wird diese Funktionsweise dennoch nicht auf die physische Host-Maschine übertragen. Der Grund dafür ist, dass der Browser, der den Bytecode in Maschinencode übersetzt, selbst auf einer Register Machine arbeitet und somit auch Register verwendet. Dies ist dennoch vorteilhaft, denn da WebAssembly keine Register angibt, hat der Browser mehr Flexibilität die beste Registerzuweisung für die CPU zu verwenden, auf der die Anwendung ausgeführt wird [11, p. 8].

Die Handhabung des Kontrollflusses in WebAssembly unterscheidet sich ein wenig von anderen, weniger portablen Assemblersprachen. WebAssembly unternimmt große Anstrengungen, um sicherzustellen, dass der Kontrollfluss die Typsicherheit nicht außer Kraft setzen kann.

Damit sollen Angriffe, wie der Heap Corruption entgegengewirkt werden. Ein Aspekt, durch den dies ermöglich werden soll, ist der lineare Speicher [11, pp. 9-10].

Linearer Speicher

Anders als bei anderen Sprachen, verfügt WebAssembly weder über einen Heap noch über komplexe Objekte. In anderen Sprachen ist es möglich einfach neue Instanzen von Objekten auf dem Heap mit dem „new"-Operator zu erzeugen. Dadurch weiß der Compiler automatisch, ob ein Zeiger oder ein Wert übergeben wurde und wo er diesen anordnen muss, um ihn verfügbar zu machen. In WebAssembly existiert kein Konzept eines new-Operators, um Speicher auf Objektebene zu reservieren. WebAssembly verfügt über einen linearen Speicher. Dabei handelt es sich um einen fortlaufenden Block von Bytes, der intern im Modul deklariert wird. Dieser lineare Speicher besteht aus sogenannten Seiten, die jeweils eine Größe von 64KB haben. Dieser Speicher kann beliebig vergrößert und verkleinert werden. Da die Laufzeitumgebung über keinen „Garbage Collector" verfügt, ist der Entwickler selbst für die Zuteilung des Speichers verantwortlich. [11, pp. 10-11]

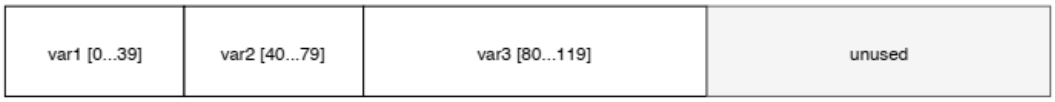

Figure 4 - Beispiel für die Strukturierung des linearen Speichers in WebAssembly (Gallant, 19)

Die Abbildung (Figure 4) zeigt eine mögliche Verwendung des linearen Speichers. Hier besteht die Anordnung aus Variablen und Offsets. Neben der Effizienz des direkten Speicherzugriffs gibt es einen weiteren entscheidenden Vorteil für die Sicherheit [11, p. 12] . Während der Host den linearen Speicher, der einem Wasm-Modul zur Verfügung gestellt wird, jederzeit lesen und beschreiben kann, kann das Wasm-Modul niemals auf den Speicher des Hosts zugreifen. Darüber hinaus prüft das WebAssembly-Framework bei jedem Speicherzugriff, ob der dieser innerhalb der Grenzen des Speichers stattfindet [12, p. 21].

Sektionen

Ein Wasm-Modul ist in verschiedene Abschnitte (Figure 5) unterteilt, die nachfolgend Sektionen genannt werden. Die Aufteilung in Sektionen bewirkt ebenfalls diverse Vorteile, vor allem in Bezug auf Performanz und Sicherheit. Prozesse, wie Validierung und Kompilierung können so parallel durchgeführt werden, wodurch das Modul schneller einsatzbereit ist. [12, p. 17]

Für die Erstellung und Strukturierung der Sektionen sind nicht die Entwickler verantwortlich. Der Compiler ist dafür verantwortlich, die Abschnitte nach Bedarf zu erstellen und sie auf der Grundlage des geschriebenen Codes in der richtigen Reihenfolge zu platzieren.

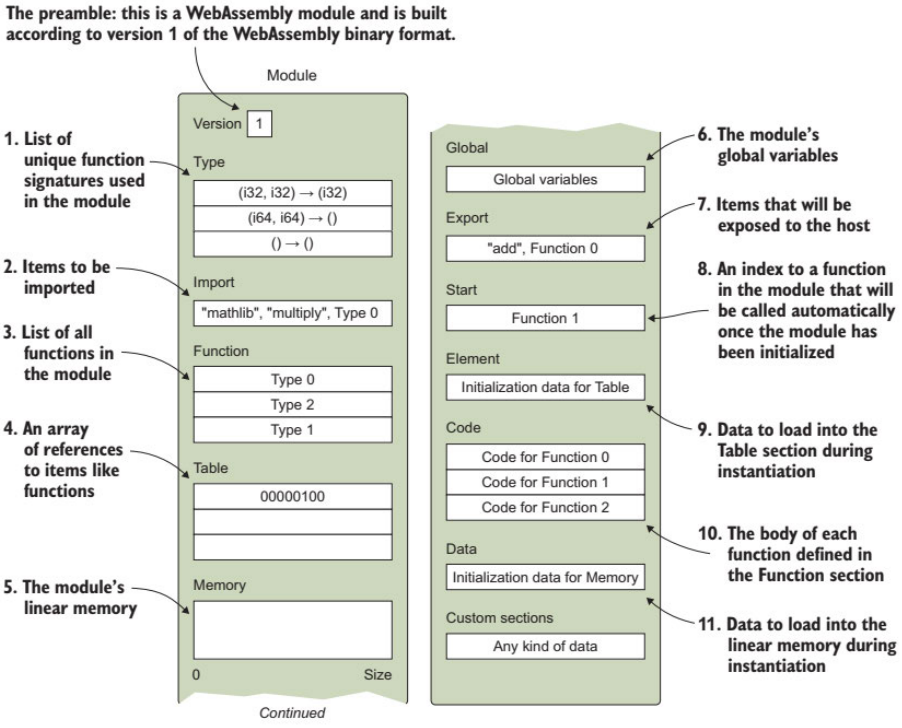

Figure 5 - Sektionen eines Wasm-Moduls - [12]

Das Modul beginnt mit der Preamble-Sektion, die angibt, dass es sich um ein Wasm-Modul handelt und in diesem Modul die Version 1 des Wasm-Bytecode-Formats verwendet wird.

Nach der Preamble-Sektion folgen die obligatorischen Known[7]- und die optionalen Custom-Sections[8]. Während die Known-Sections jeweils nur einmal vorkommen dürfen und eine feste Reihenfolge besitzen, können die Custom-Sections überall im Modul beliebig oft vorkommen. [12, p. 18]

Ein anschauliches Beispiel für die Known-Sections, sind die Sektionen *Type*, *Function* und *Code*. Die Type-Sektion enthält eine Liste der Signaturen aller Funktionen, die innerhalb des Moduls zum Einsatz kommen. Dabei kann jede Signatur nur einmal vorkommen, wodurch sich auch mehrere Funktionen dieselbe Signatur teilen können. Die Function-Sektion enthält die Liste aller Funktionen des Moduls, während die Code-Sektion die Liste mit den Bodies aller Funktionen enthält. Wie man in der Abbildung (Figure 6) erkennen kann, referenziert der Wert, der zweiten Funktion „Type 2“ in der Function-Sektion, die Signatur mit dem Index 2 in der Type-Sektion, während der Index der Funktion „Type 2“ denselben Index enthält, wie der Body in der Code-Sektion. Diese Art der Referenzierung gilt für alle Elemente in diesen Sektionen. Durch diese Aufteilung ist eine parallele Validierung des Moduls möglich, es ist so wesentlich schneller einsatzbereit. Funktionsdeklarationen sind von den Funktionskörpern getrennt, um eine parallele Streaming-Compilation jeder Funktion im Modul zu ermöglichen. [12, p. 20]

[7] bekannte Sektionen

[8] benutzerdefinierte Sektionen

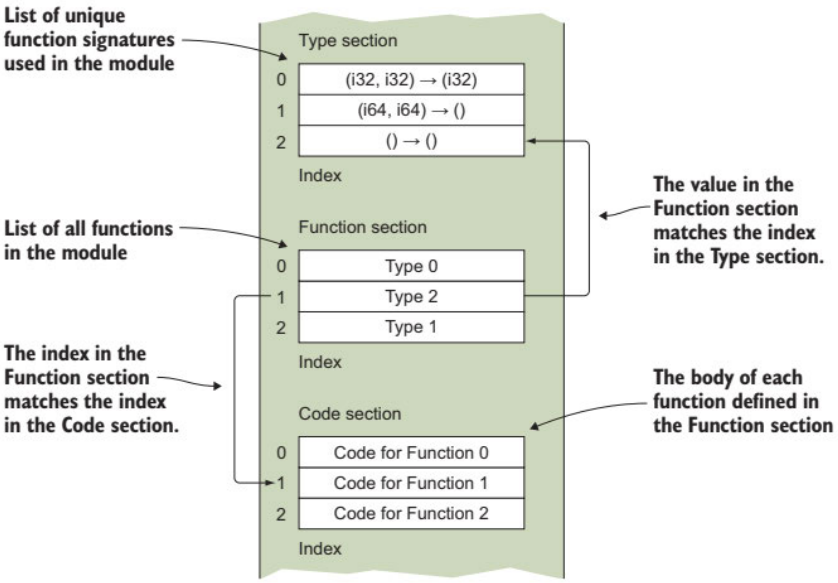

Figure 6 - Type-, Function- und Code-Section eines Wasm-Moduls - [12]

Die Table-Sektion stellt einen der zentralen Sicherheitsaspekte von WebAssembly bereit. Diese Sektion enthält ein Array mit Referenzen wie Funktionen, welche nicht direkt im linearen Speicher des Moduls liegen sollen.

Der Code, der im Modul ausgeführt wird, hat keinen direkten Zugriff auf die in der Tabelle gespeicherten Referenzen. Wenn der Code auf die Daten zugreifen möchte, auf die in dieser Sektion verwiesen wird, bittet er stattdessen das Framework, auf dem Element zu operieren, welches sich auf dem entsprechenden Index befindet. Das WebAssembly-Framework liest dann die bei diesem Index gespeicherte Adresse und führt die Aktion aus. [10, p. 20]

Ein weiterer zentraler Sicherheitsaspekt wird durch die Memory-Sektion gewährleistet. WebAssembly-Module haben, wie schon beschrieben, keinen direkten Zugriff auf den Speicher des ausführenden Systems. Stattdessen wird ihnen von der Laufzeitumgebung ein ArrayBuffer zugewiesen, welches vom Modul als linearer Speicher verwendet wird. Dieser Speicher ist vergleichbar mit einem Heap bei C++. Jedes Mal, wenn der Code auf den Speicher zugreift, prüft das Framework, ob dieser Zugriff innerhalb der Grenzen dieses Arrays liegt. [12, pp. 21-22]

Custom Sections

Diese Sektionen enthalten spezielle benutzerdefinierte Sektionen und tragen nicht direkt zur WebAssembly-Semantik bei. Sie können nützliche Metadaten zur Verfügung stellen, die von Implementierungen genutzt werden können, um die Benutzerfreundlichkeit zu verbessern oder Hinweise für die Kompilierung zu erhalten. [21] Von Haus aus existiert bisher nur eine dieser Sektionen, die Name-Sektion. Der Zweck dieser Sektion ist es, Definitionen in einem Modul mit ausgebbaren Namen zu versehen, die z. B. von einem Debugger verwendet werden können oder wenn Teile des Moduls in Textform ausgegeben werden sollen. [21]

Beispiel

Mit dem Toolkit WABT[9] ist es möglich Informationen zu einem vorliegenden Wasm-Modul zu erhalten. Der Konsolenbefehl *wasm-objdump* listet die Sektionen eines Moduls auf. Das folgende Listing (Listing 2) gibt einen Einblick in diese Sektionen.

```
1  #[wasm_bindgen]
2  pub fn sum(a: i32, b: i32) -> i32 {
3    a + b
4  }
```

```
1  Type[1]:
2   - type[0] (i32, i32) -> i32
3  Function[1]:
4   - func[0] sig=0 <sum>
5  Memory[1]:
6   - memory[0] pages: initial=17
7  Export[2]:
8   - memory[0] -> "memory"
9   - func[0] <sum> -> "sum"
10 Code[1]:
11  - func[0] size=7 <sum>
12 Custom:
13  - name: "producers"
```

Listing 2 - Beispiel mit Sektionen eines kleinen Wasm-Moduls

[9] https://github.com/WebAssembly/wabt

Im oberen Teil des Listings, ist eine in der Sprache Rust geschriebene Funktion zu sehen. Sie nimmt zwei Parameter entgegen und gibt die Summe dieser Parameter zurück. Im unteren Teil des Listings befindet sich Ausgabe des *wasm-objdump*-Befehls. Es zeigt die Sektionen des Wasm-Moduls, das aus dieser Funktion generiert wurde.

Man kann in dieser Ausgabe u.a. die zuvor beschriebenen Sektionen erkennen. In Zeile 2 und 3 befindet sich die Type-Sektion, welche die Signatur der in Rust geschriebenen Funktion enthält. Anzumerken ist hierbei, dass die Parameter zufällig dieselben Datentypen aufweisen. Rust verfügt über denselben primitiven Datentypen i32, wie WebAssembly. Die darauffolgende Sektion Function enthält die Liste aller Funktionen. In diesem Fall nur die eine. Die Memory-Sektion hält den linearen Speicher. Zu erkennen ist hier in Zeile 6, dass der Speicher eine initiale Größe von 17 Seiten hat. In der Export-Sektion befindet sich die Liste mit allen Objekt, die in die Host-Umgebung exportiert werden, sobald das Modul instanziiert ist. In diesem Fall ist das der Speicher und die Funktion. Die Code-Sektion enthält den Body der enthaltenen Funktion und ganz am Ende befindet sich noch eine vom Compiler generierte Custom-Section. Der Zweck der Sektion "producers" besteht darin, eine optionale, strukturierte Aufzeichnung aller verschiedenen Tools bereitzustellen, die zur Erstellung eines bestimmten WebAssembly-Moduls verwendet wurden. Dies soll der späteren Analyse dienen [22].

2.3 Rust

Rust ist eine multiparadigmische Systemprogrammiersprache, die von einer Open-Source-Community entwickelt wurde und unter anderem von Mozilla Research gesponsert wird. Sie ist vor allem auf Geschwindigkeit, Speichersicherheit und Parallelität konzentriert. So eignet sich die Sprache besonders gut zum Entwickeln von Spiele-Engines, Betriebssystemen, Dateisystemen, Browser-Komponenten und VR-Simulations-Engines. [23]

Rust ist eine relativ junge Sprache (2010) und enthält Elemente aus bewährten Systemprogrammiersprachen sowie modernem Programmiersprachendesigns. Sie verfügt über eine ausdrucksstarke und intuitive Syntax von Hochsprachen mit der Kontrolle und Leistung einer Low-Level-Sprache. Außerdem verhindert sie Segmentierungsfehler und garantiert Thread-Sicherheit.

[23] Syntaktisch ist die Sprache eher an C angelehnt. Funktionen beginnen mit dem Operator *fn* und auch Konstrukte wie *structs* sind hier vertreten. Rust bietet außerdem eine der ausgereiftesten Unterstützungen für WebAssembly an. Es existieren bereits diverse Bibliotheken, welche die Kommunikation mit JavaScript aus den Wasm-Modulen sehr vereinfachen. Neben der stark statischen Typisierung existieren weitere interessante Konzepte, die sich hervorragend mit WebAssembly vereinigen lassen. Vor allem die Umsetzung der Speichersicherheit ohne Garbage Collection und die Unveränderlichkeit der Variablen sind hier von großer Bedeutung.

2.3.1 Speichersicherheit durch Ownership

Die zentrale Eigenschaft, die Rust bisher einzigartig macht, ist das Ownership[10]-Konzept. Viele andere Sprachen, wie Java oder JavaScript benötigen eine Garbage Collection. Dieser läuft neben jedem ausführenden Programm und prüft nach einem gewissen Regelwerk, ob Variablen noch benötigt werden. Ist dies nicht der Fall, so kann der Wert der Variable gelöscht und der Speicher freigegeben werden. In Rust muss schon während der Programmierung auf die Speicherzuweisung geachtet werden. Ein Wert, wird hier vom Speicher gelöscht, sobald die Variable, welche die Referenz auf diesen Wert besitzt, ihren Scope[11] verlässt. Zusätzlich kann jeweils immer nur eine Variable die Referenz auf einen Wert besitzen. Dies ist auch die namensgebende Eigenschaft dieses Konzepts. Zusammenfassend lassen sich drei Regeln [24, p. 61] aufstellen, die bei der Programmierung zu beachten sind:

- Jeder Wert in Rust hat eine Variable, die als sein Besitzer bezeichnet wird.
- Es kann immer nur einen Besitzer geben.
- Wenn der Besitzer den Gültigkeitsbereich verlässt, wird der Wert gelöscht.

Dieses Konzept macht in Rust geschrieben Programme sehr effizient, da hier kein Garbage Collector benötigt wird, um Speichersicherheit zu gewährleisten. Wie bereits erwähnt, verträgt

[10] exklusiver Besitz

[11] Sichtbarkeitsbereich einer Variable

sich diese Eigenschaft auch hervorragend mit den WebAssembly-Konzepten, da in den Modulen ebenfalls keine Garbage Collection vorgesehen ist. [24, pp. 59-64]

2.3.2 Unveränderlichkeit von Variablen

Ein weiteres wichtiges Konzept der Sprache ist, dass Variablen in Rust standardmäßig unveränderlich sind. Dadurch werden die Entwickler gezwungen, sicheren und vor allem einfach zu parallelisierenden Code zu schreiben, denn unveränderliche Daten garantieren, dass Probleme wie Race Conditions nicht stattfinden können. [24]

2.4 AssemblyScript

Die Sprache AssemblyScript bildet wie asm.js eine Untermenge von JavaScript bzw. von TypeScript. Im Gegensatz zu asm.js wird allerdings direkt in dieser Sprache implementiert. Anschließend wird der Code mithilfe des Binaryen-Compilers[12] zu WebAssembly kompiliert und somit ein Modul erzeugt. Sie wurde ausschließlich zum Entwickeln von Wasm-Modulen konzipiert. Der AssemblyScript-Syntax unterscheidet sich kaum vom TypeScript-Syntax. Der Unterschied besteht hier hauptsächlich in der Typisierung. Während in TypeScript versucht wurde, die dynamische Natur von JavaScript zu erhalten, so ist dies hier nicht mehr möglich. AssemblyScript verwendet dieselben Datentypen wie WebAssembly und ist dabei statisch typisiert. Darüber hinaus handelt es sich hierbei nicht um eine Interpretersprache. Die AssemblyScript-Algorithmen werden schon vor der Ausführung zu WebAssembly-Bytecode kompiliert. [25]

AssemblyScript enthält außerdem einen Teil der JavaScript-Standardbibliothek, wodurch u.a. die Nutzung von Math, Arrays, Strings und Map ermöglicht wird. Die nahe Verwandtschaft zu TypeScript, ermöglicht darüber hinaus bereits bestehende Konzepte, aus Webanwendungen mit geringen Veränderungen in WebAssembly-Module umzuwandeln. Entwicklern aus dem

[12] https://github.com/WebAssembly/binaryen

Front-End-Bereich, die überwiegend mit TypeScript bzw. JavaScript vertraut sind, wird so außerdem eine gute Möglichkeit geboten, die Vorteile von WebAssembly zu nutzen. Dies ist auch der Grund, warum die Sprache für dieses Projekt gewählt wurde. [25]

Folgendes Listing (Listing 3) bildet zwei Funktionen ab. Beide Funktionen nehmen zwei Parameter entgegen und addieren diese anschließend. Die obere Funktion ist allerdings in TypeScript und die untere in AssemblyScript implementiert. Es zeigt sich, dass sich diese Algorithmen lediglich in den verwendeten Datentypen unterscheiden.

```
1 export function sum(a: number, b: number): number {
2   return a + b;
3 }
```

```
1 export function sum(a: i32, b: i32): i32 {
2   return a + b;
3 }
```

Listing 3 - Beispiel eines AssemblyScript-Programms

2.5 Performanz

In der Informatik beschreibt die Performanz ein Maß für die Rechenleistung von IT-Systemen [2]. Anders als in der Physik, in der die Leistung den Quotienten aus verrichteter Arbeit pro Zeitintervall beschreibt, gibt es in der Informatik keine festgeschriebene Definition [26]. Es existieren zwar diverse Möglichkeiten, um Leistungsmerkmale eines IT-Systems durch verschiedene Messwerte zum Ausdruck zu bringen, allerdings handelt es sich dabei meist um relative Größen, die von anderen Einflussfaktoren abhängen. Daher ist es hier notwendig, die Leistung in Relation zu einem Referenzsystem oder zu einer bestimmten Arbeitslast zu messen [27]. Typische Messwerte können hier u.a. sein:

- Anzahl der Rechenoperationen pro Sekunde, im Fall von Gleitkommazahlen in „FLOPS“ gemessen.
- Die Latenz, welche die Zeit zwischen Start und Ende eines Datentransfers angibt.
- Die Datentransfer- und Datenübertragungsrate.

Wenn es speziell um Anwendungen geht, dann wird unter Performanz im Allgemeinen verstanden, wie effektiv ein Softwaresystem in Bezug auf Zeitvorgaben und die Zuweisung von Ressourcen ist. Die traditionellen Leistungskennzahlen dafür sind Reaktionszeit, Durchsatz und Auslastung. Dabei beschreibt die Reaktionszeit die Zeit, die benötigt wird, um eine Anfrage zu beantworten. Dies kann sowohl eine Reaktion auf einen einzelnen Mausklick sein als auch der komplette Prozess eines Anwendungsfalls, welcher aus vielen Teilaufgaben besteht. Der Durchsatz beschreibt die Anzahl an Aufträgen, die ein System innerhalb eines bestimmten Zeitintervalls verrichten kann. Die Auslastung ist der Prozentsatz der Zeit, in der ein bestimmter Teil des Systems mit einem Auftrag beschäftigt ist. [28]

Je nach Art der Anwendung können diese Kennzahlen von verschiedenen Faktoren abhängen. Bei einer Webanwendung zum Beispiel können folgende Faktoren Einfluss auf die Performanz haben:

- Bearbeitungszeit im Back-End
- Übertragungsgeschwindigkeit
- Dateigrößen
- Im Front-End verwendete Technologien, wie z.B. Frameworks
- Codequalität
- Qualität und Strukturierung von HTML und CSS
- Eingesetzter Browser

Besonders wichtig für die Benutzererfahrung ist die Startzeit der Webanwendung. Direkten Einfluss darauf hat die Größe der Dateien, die wiederum von der eingesetzten Technologie sowie Programmiersprachen abhängig ist. Außerdem kann die Ladezeit je nach Komplexität des verwendeten Front-End-Frameworks variieren. Diese Faktoren sind daher auf keinen Fall zu vernachlässigen. [29]

Zusammengefasst lässt sich ableiten, dass die Evaluierung der Performanz eines Systems kein trivialer Prozess ist. Aus einer vermeintlich bewertenden Aussage wie „System X ist schneller als System Y“, lässt sich keine professionelle Schlussfolgerung ableiten. Handelt es sich bei diesen Systemen um Smartphones und bei der bewertenden Person um einen Standardanwender, so kann er damit meinen, dass System X eine schnellere Reaktionszeit hat, wie das schnelle

Öffnen einer App, handelt es sich allerdings bei den Systemen um Server und bei der Person um einen Administrator, so könnte es wiederum bedeuten, dass System X einen höheren Durchsatz hat. Um beim Messen der Performanz also professionellere Bewertungen durchführen zu können, setzt man seit langem auf eine bewährte und vor allem genormte Methode, die als Benchmark bezeichnet wird.

2.5.1 Benchmarks

Benchmarks sind Programme die speziell dafür entwickelt wurden, um eine standardisierte Leistungsbewertung von Computersystemen auf der Grundlage spezifischer Merkmale zu ermöglichen. Benchmarks sind so konzipiert, dass sie einen bestimmten Workload[13] auf einer Komponente oder einem System nachahmen und eine Messung der Auswirkungen erlauben.

Korrekte Benchmarks zeichnen sich dabei u.a. durch folgende Eigenschaften [30] aus:

- Repräsentativität: Benchmark-Performance-Metriken sollten von Industrie und Wissenschaft breit akzeptiert werden.
- Reproduzierbarkeit: Benchmark-Ergebnisse können verifiziert werden.
- Skalierbarkeit: Benchmark-Tests sollten über Systeme hinweg funktionieren, die eine Bandbreite an Ressourcen von niedrig bis hoch besitzen.
- Transparenz: Die Benchmark-Metriken sollten einfach zu verstehen sein.

Die Grundlagen der Performanzmessung sind in einer ISO Norm standardisiert. Dort wird der Ablauf eines Benchmarks in drei Schritten beschrieben [31]:

1. Stabilisierungsphase: System wird in einen stabilen Funktionszustand gebracht, der Workload wird geladen
2. Bewertungsintervall: Der Test wird wiederholt ausgeführt
3. Ergänzungslauf: Die Tests werden weiterhin durchlaufen, bis die geforderten Wiederholungen abgeschlossen wurden

[13] Arbeitslast

Anschließend können die Ergebnisse, welche in der zweiten und dritten Phase gemessenen wurden, verwendet werden, um verschiedene Werte zu errechnen, mit denen sich die Systeme besser vergleichen lassen. Typische Vergleichswerte sind hier z.B. die mittlere Ausführungsdauer oder der Durchsatz. Um die Reproduzierbarkeit der Ergebnisse zu gewährleisten, sollte zu jeder Messung außerdem eine ausführliche Beschreibung der Testumgebung erfolgen. Dazu zählen u.a. die eingesetzte Hardware sowie das Betriebssystem. Um aussagekräftigere Ergebnisse zu erhalten, sollten die Testdurchläufe mehrmals wiederholt werden. Dabei sollte der Workload bei jedem Durchlauf variieren, um so das System unterschiedlich zu belasten und ein realistischeres Szenario nachbilden zu können [32].

Es existieren mehrere Arten von Benchmarks, die sich in ihrer Komplexität und ihrer Aussagekraft teilweise stark voneinander unterscheiden. Einer der bekanntesten Vertreter ist der synthetische Benchmark. Synthetische Benchmarks sind meist kleine Programme oder Teile von Programmen, mit denen versucht wird, die Funktionalität eines echten Programms nachzubilden, um dann Aussagen über das Verhalten eines Systems treffen zu können. Viele der bekannten Browserhersteller haben eine Reihe dieser Benchmarks entwickelt und in Suites[14] bereitgestellt, um so die Leistung ihrer Browser unter Beweis stellen zu können. Zu diesen Suites zählen Octane[15], JetStream[16] oder Kraken[17]. In diesen Suites sind meist Benchmarks zu finden, mit denen versucht wird, Funktionen aus realen Anwendungen nachzubilden. Typische Vertreter sind dabei kryptographische Funktionen, Base64-Funktionen, RegExp-Funktionen, 3D-Rendering oder Hash-Funktionen [32].

Es ist heute allerdings allgemein bekannt, dass synthetische Benchmarks nicht die beste Wahl sind, um ein System zu bewerten. Einer der Gründe dafür ist, dass diese Benchmarks nicht den kompletten Umfang einer realen Anwendung nachbilden können. Somit kann auch nicht evaluiert werden, wie sich das System unter realen Bedingungen verhalten würde. So hat sich in dem Paper „How fast is WebAssembly?“ [33] gezeigt, dass bisherige Bewertungen von

[14] Sammlung mehrerer Benchmarks

[15] https://chromium.github.io/octane/

[16] https://browserbench.org/JetStream/

[17] https://krakenbenchmark.mozilla.org/kraken-1.1/driver

WebAssembly nur teilweise valide sind, da die dort verwendeten Benchmarks nicht das komplette Potential dieser Technologie ausschöpfen können.

Trotz dieser Tatsache eignen sich synthetische Benchmarks dennoch, um WebAssembly im Rahmen dieser Arbeit im Vergleich zu den anderen Technologien bewerten zu können. Vor allem ist es so leicht möglich kleine Komponenten mit speziellen Aufgaben in den hier eingesetzten Sprachen JavaScript, Rust und AssemblyScript zu realisieren, um diese dann anschließend in ihrer Leistung miteinander vergleichen zu können.

3 Anforderungsanalyse

In diesem Kapitel werden nun die Anforderungen definiert, die von der zu entwickelnden Anwendung erfüllt werden müssen. Die Anforderungen bestehen aus funktionellen sowie nicht-funktionellen Anforderungen. Neben der Bezeichnung erhalten die Anforderungen außerdem ein Kürzel zur besseren Referenzierung. Diese Kürzel bestehen aus einer fortlaufenden Nummerierung und den Buchstaben [FA] für funktionale Anforderungen und [NFA] für nicht-funktionale Anforderungen.

3.1 Funktionale Anforderungen

Die nachfolgend definierten funktionalen Anforderungen beschreiben die Funktionen der zu entwickelnden Anwendung. Die hier beschriebenen Funktionen müssen nachweisbar in der fertigen Anwendung umgesetzt sein.

[FA-1] Die Anwendung soll in allen gängigen Browsern ausführbar sein

WebAssembly wird von allen gängigen Browsern unterstützt, allerdings gibt es Unterschiede in den Implementierungen der Technologie. Damit die Performanz unabhängig von der ausführenden Umgebung getestet werden kann, soll die Anwendung in jeden dieser Browser ausführbar sein.

Die Anwendung soll mindestens auf den folgenden Browsern ausführbar sein:

- Google Chrome (ab Version 91)
- Google Chrome Mobil (ab Version 91)
- Microsoft Edge (ab Version 91)
- Mozilla Firefox (ab Version 89)
- Apple Safari (ab Version 14)

[FA-2] Ausführbarkeit der Benchmarks

Die Benchmarks in dieser Anwendung sollen gesammelt sowie separat ausführbar sein.

[FA-3] Konfigurierbarkeit

Um ein realistischeres Szenario nachbilden zu können, sollte es möglich sein, das System auf unterschiedliche Weise zu belasten. Es sollte daher möglich sein, die Art der Ausführung der Benchmarks in einem gewissen Ausmaß konfigurierbar zu machen. Dabei sollte vor allem die Größe des Workloads sowie die Anzahl der Durchläufe veränderbar sein.

[FA-4] Benchmarks

Basierend auf der in den Grundlagen beschriebenen Funktionsweise von WebAssembly, sollte eine angemessene Vielfalt von Benchmarks implementiert werden. Vor allem sollte bei der Wahl der Benchmarks versucht werden, die Stärken und Schwächen der Technologie näher zu beleuchten.

[FA-4.1] Rechenintensive Benchmarks

Aus den Anwendungsfällen [34] von WebAssembly ist bekannt, dass sich diese Technologie vor allem für rechenintensive Aufgaben eignet. Es sollten daher entsprechende Benchmarks entwickelt werden, mit denen sich diese Eigenschaft evaluieren lässt. Dabei sollte möglichst auf andere störende Prozesse, wie das Senden vieler Daten verzichetet werden.

[FA-4.2] Kryptographie

Kryptographie ist ein weiterer Anwendungsfall, der von den Entwicklern angegeben wird. Es soll mindestens ein Benchmark entwickelt werden, der eine kryptographische Funktion implementiert.

[FA-4.3] Kommunikation

Da die Wasm-Module keinen direkten Zugriff auf den Speicher des Hosts haben, ist das Übertragen von Daten noch relativ teuer. Es sollten Benchmarks entwickelt werden, mit denen evaluiert werden kann, welche Leistungseinbußen durch das Übertragen verschiedener Datentypen zu erwarten sind.

[FA-4.4] Verarbeitung von Strings

Es ist bekannt, dass WebAssembly über nur vier primitive Datentypen verfügt. Komplexe Datentypen wie Strings werden nicht direkt unterstützt. Es sollten daher Benchmarks entwickelt werden, mit denen evaluiert werden kann, ob WebAssembly dennoch für diesen Aufgabenbereich angewendet werden kann.

[FA-4.5] DOM-Manipulation

WebAssembly soll eher eine Ergänzung als ein Ersatz für die Sprache JavaScript sein. Das liegt u.a. daran, dass die Wasm-Module keinen direkten Zugriff auf den Host haben und es

somit nicht direkt möglich ist, den DOM-Tree zu manipulieren. Mit der Bibliothek „web-sys[18]“ für Rust soll dies dennoch möglich sein. Es soll ein Benchmark entwickelt werden, mit dem diese Möglichkeit evaluiert werden kann.

[FA-4.6] Bildmanipulation

Ein weiterer für WebAssembly angegebener Anwendungsfall ist die Bildmanipulation. Da diese Funktionalität sowohl rechenintensiv ist, als auch die Verarbeitung komplexer Datentypen voraussetzt, sollte ein Benchmark entwickelt werden, mit denen diese Eigenschaften evaluiert werden können.

[FA-5] Ergebnisse anzeigen

Nachdem der Benchmark-Prozess abgeschlossen ist, sollen sinnvolle Ergebnisse ausgegeben werden, mit denen es möglich ist die Technologien miteinander zu vergleichen.

[FA-6] Grafische Darstellung der Ergebnisse

Die Ergebnisse sollen in aussagekräftigen Grafiken in Form von Diagrammen dargestellt werden.

3.2 Nicht-Funktionale Anforderungen

Die nachfolgend definierten, nicht-funktionalen Anforderungen beschreiben, wie gut das zu entwickelnde System, die im vorherigen Unterkapitel definierten Funktionen umsetzen soll. Sie tragen somit maßgeblich zu Qualität der Anwendung bei.

[18] https://rustwasm.github.io/wasm-bindgen/web-sys/index.html

[NFA-1] Optimale Implementierung

Da die Benchmarks in verschiedenen Sprachen implementiert werden, kann nur schwer nachvollzogen werden, ob die Algorithmen im Hintergrund tatsächlich dieselben Operationen ausführen. Um für die Messungen dennoch möglichst faire Voraussetzungen zu schaffen, sollen die Algorithmen strukturell in einer möglichst gleichen Form vorliegen. Falls dies nicht möglich ist, dann soll zumindest eine Möglichkeit der Umsetzung gewählt werden, von der bekannt ist, dass es sich um eine performante Umsetzung handelt.

[NFA-2] Übersichtlichkeit

Die Übersichtlichkeit und eine einfache Bedienbarkeit sollten beim Design des grafischen Interfaces an erster Stelle stehen. Ein umfangreiches und grafisch aufwendiges Interface ist nicht nötig, da dies nicht zum Zweck der Anwendung beitragen würde.

[NFA-3] Fehlertoleranz

Wenn ein Benchmark einen Fehler erzeugt, sollte nicht die komplette Reihe aller Benchmarks abgebrochen werden, stattdessen sollte mit dem nachfolgenden Benchmark fortgefahren werden. Falls ein Fehler bei der Ausführung entstanden ist, sollte dies irgendwo in geeigneter Form ersichtlich sein.

[NFA-4] Erweiterbarkeit

WebAssembly ist eine noch sehr junge Technologie. Es ist zu erwarten, dass in Zukunft noch viele Features hinzukommen werden, mit der die Technologie erweitert und verbessert wird. Es ist also von Vorteil, wenn die Anwendung so realisiert wird, dass sie leicht um weitere Benchmarks erweitert werden kann.

4 Entwurfsentscheidungen

Im nun folgenden Kapitel werden die Entwurfsentscheidungen erläutert, die zur Umsetzung der im vorherigen Kapitel definierten Anforderungen getroffen wurden. Beschrieben werden die gewählten Technologien, die Benutzungsoberfläche und die Architektur, zu der die Struktur der Anwendung in verschiedenen Detailstufen und die wichtigsten Abläufe gehören. Außerdem wird kurz der Aufbau der Wasm-Module angeschnitten, bevor im Kapitel „Implementation“ näher auf die Logik eingegangen wird.

4.1 Technologien

Die zu entwickelnde Anwendung, soll eine Evaluierung von WebAssembly in Bezug auf die clientseitige Performanceverbesserung ermöglichen. Aus diesem Grund wird für dieses Projekt kein Back-End benötigt. Die Bereitstellung sämtlicher für das Projekt benötigten Daten, sowie deren Verarbeitung geschieht ausschließlich durch das Front-End. Zur Entwicklung dieser Anwendung wird das Front-End-Framework Angular[19] verwendet. Angular ist ein von Google entwickeltes Open-Source-Framework zur Erstellung skalierbarer, dynamischer Webanwendungen. Es enthält eine Sammlung gut integrierter Bibliotheken, die eine Vielzahl von Funktionen abdecken, einschließlich Routing, Formularverwaltung, Client-Server-Kommunikation und mehr [35]. Es unterstützt das Entwickeln übersichtlicher und schneller Anwendungen und wird kontinuierlich von einer großen Community weiterentwickelt. Zur Anwendung kommt in diesem Projekt die Version 11. Angular als komponentenbasiertes Framework erlaubt eine leichte Modularisierung, wodurch eine übersichtliche Struktur sowie eine bequeme Erweiterbarkeit der Anwendung begünstigt wird. Die für Angular verwendeten Sprachen sind HTML und CSS für die Strukturierung der Benutzungsoberfläche und TypeScript[20] für die Logik. TypeScript ist somit die Sprache, in der die Anwendung entwickelt wird. TypeScript ist eine

[19] https://angular.io/

[20] https://www.typescriptlang.org/

Open-Source-Sprache, aus dem Hause Microsoft. Sie bildet eine syntaktische Obermenge von JavaScript und fügt der Sprache eine optionale, statische Typisierung sowie Klassen, Vererbung und Module hinzu. Jeder geschrieben JavaScript-Code ist somit auch gültiger TypeScript-Code [36]. Dies ist für dieses Projekt von besonderer Bedeutung, da sich die in JavaScript geschriebenen Benchmarks so leicht in die Anwendung integrieren lassen. Zur Gestaltung der Benutzungsoberfläche wird außerdem das CSS-Framework Bootstrap[21] in der Version 4 verwendet. Es enthält auf HTML und CSS basierende Gestaltungsvorlagen, wie z.B. Typografie, Formulare, Buttons und ein praktisches Grid-System zur einfachen Anordnung der Elemente. Angular selbst sowie alle in diesem Projekt verwendeten Abhängigkeiten, werden als NPM-Pakete in der NPM-Registry[22] zur Verfügung gestellt. Aus diesem Grund wird in diesem Projekt der NPM-Paketmanager verwendet, welcher Bestandteil der Node.js-Laufzeitumgebung[23] ist. Die für die Auswertung der Benchmarks erforderlichen Diagramme, werden durch das „ngx-charts[24]“-Package bereitgestellt. Darüber hinaus existieren diverse weitere Abhängigkeiten, die für den Betrieb der Benchmarks benötigt werden und im weiteren Verlauf der Arbeit näher beleuchtet werden.

Neben den Abhängigkeiten der Benchmark-App, existieren weitere Abhängigkeiten, die in den Rust- und AssemblyScript-Projekten Verwendung finden. Für die AssemblyScript-Projekte wird ebenfalls NPM verwendet, mit dem u.a. der Compiler integriert werden kann, der benötigt wird, um den dortigen Code zu Wasm-Modulen zu kompilieren. In den Rust-Projekten kommt der Paketmanager „Cargo“ zum Einsatz. Die durch diesen Paketmanager bereitgestellten Abhängigkeiten werden Crates[25] genannt. Die wichtigste Crate ist hier die „wasm-bindgen[26]“. Damit werden Funktionen bereitgestellt, welche die Kommunikation zwischen den aus Rust generierten Wasm-Modulen und der Benchmark-App ermöglichen. Für das Generieren der

[21] https://getbootstrap.com/

[22] https://www.npmjs.com/

[23] https://nodejs.org/en/

[24] https://swimlane.gitbook.io/ngx-charts/

[25] Kisten

[26] https://docs.rs/wasm-bindgen/0.2.74/wasm_bindgen/

Wasm-Module wird das Tool „wasm-pack[27]“ verwendet. Beide Technologien sind Teil des Rust und WebAssembly Ökosystems und werden auch in den offiziellen Tutorials[28] verwendet.

4.2 Benutzungsoberfläche

Nachfolgend wird der Aufbau der Oberfläche (Figure 7) erläutert. Zur einfacheren Zuordnung der Beschreibungen, wurden Elemente der Oberfläche rot umrandet und nummeriert. Die Referenzierung eines Elements im Text erfolgt über eine eingeklammerte Nummer.

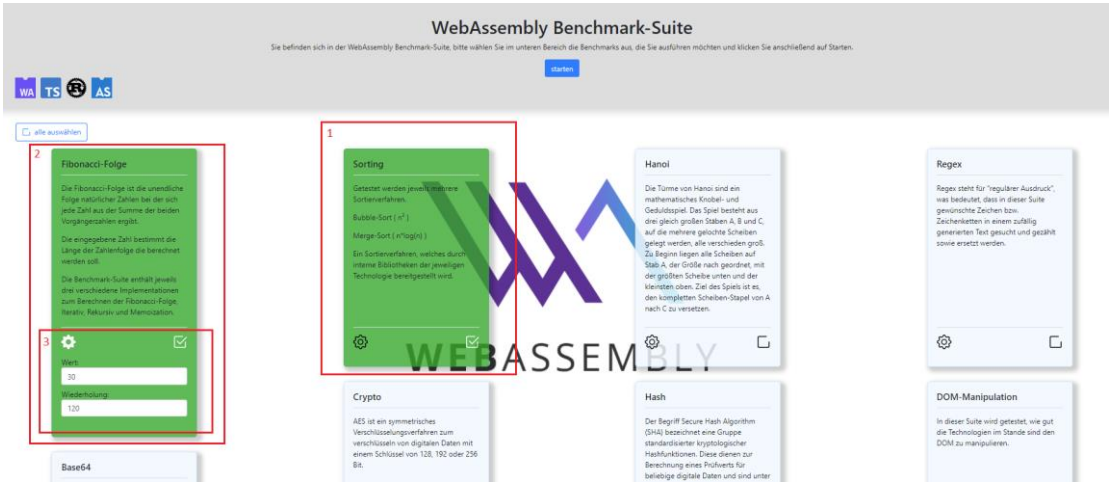

Figure 7 - Benutzungsoberfläche mit BenchCards, zwei davon sind markiert

Die Oberfläche ist im Wesentlichen in zwei Bereiche aufgeteilt. Im oberen, grau hinterlegten Bereich befindet sich neben Titel und kurzer Beschreibung der Anwendung, der Start-Button, welcher den Benchmark-Prozess startet. Im unteren, weißen Bereich der Anwendung, sind alle verfügbaren Benchmark-Suites als Karten-Felder (BenchCards) dargestellt. Jede dieser Karten enthält die Bezeichnung der Suite, sowie eine kurze Beschreibung. Jede Benchmark-Suite kann einzeln über eine Checkbox im unteren Bereich der entsprechenden Karte selektiert werden.

[27] https://rustwasm.github.io/wasm-pack/book/

[28] https://www.rust-lang.org/what/wasm

Ist eine Benchmark-Suite ausgewählt, färbt sich die Karte grün (1, 2), was zur besseren Übersicht der Anwendung beitragen soll. Über den Button „Alle auswählen“ oben links, können außerdem alle Benchmark-Suites auf einmal ausgewählt bzw. abgewählt werden. Jede Suite läuft über eine bereits festgelegte Voreinstellung. Durch Klick auf das Zahnradsymbol, das sich unten links auf jeder BenchCard befindet, ist es möglich diese Einstellungen bei Bedarf anzupassen (3). Die Art der möglichen Konfiguration variiert von Suite zu Suite. In den meisten Fällen handelt es sich allerdings um die Art des Workloads sowie Anzahl der Wiederholungen, die jeder Benchmark einer Suite durchlaufen soll.

Nachdem die gewünschten Benchmark-Suites ausgewählt und konfiguriert wurden, kann der Benchmark-Prozess über den Start-Button im oberen Bereich gestartet werden. Während der Ausführung verschwinden alle nicht ausgewählten BenchCards. Die ausgewählten BenchCards werden mit einer transparenten, grauen Ebene überdeckt, sodass die Buttons darauf nicht bedient werden können. Der Prozess-Fortschritt ist daran zu erkennen, dass auf der Karte des gerade laufenden Benchmarks eine Ladeanimation mit der Meldung „Wird ausgeführt…“ eingeblendet wird. Auf den Karten der abgeschlossenen Benchmarks wird die Meldung „Fertig!“ eingeblendet (Figure 8).

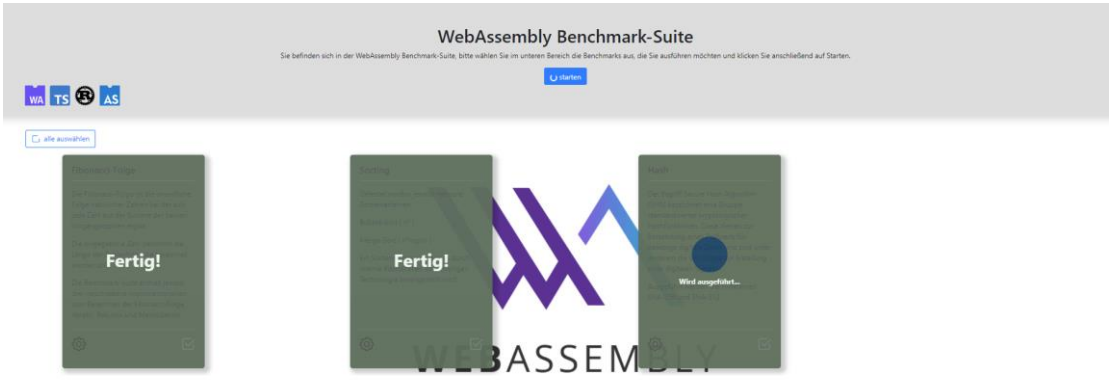

Figure 8 - Benutzungsoberfläche während des Benchmark-Prozesses

Nachdem alle Benchmarks durchgelaufen sind, verschwinden die BenchCards. Anstelle der Karten, werden im unteren Bereich nun die Auswertungen der Benchmarks in Form von Linien-Diagrammen und gemessenen Werten eingeblendet (Figure 9). Die Auswertung ist in

Bereiche unterteilt, wobei jeder Bereich die Auswertung einer Benchmark-Suite darstellt. Die einzelnen Bereiche enthalten jeweils den Titel der Benchmark-Suite, sowie die Auswertungen der darin enthaltenen Benchmarks. Die X-Achse der Diagramme stellt dabei die Iterationen dar und die Y-Achse die Zeit in Millisekunden, welche die verschiedenen Technologien für die Benchmarks benötigt haben. Die Ergebnisse der verwendeten Technologien können durch verschiedenfarbige Linien erkannt werden. Um welche Technologie es sich dabei handelt, kann in der Legende neben den Diagrammen abgelesen werden. Um die Anwendung zurückzusetzen damit sie anschließend erneut ausgeführt werden kann, muss der „Zurücksetzen"-Button im oberen Bereich angeklickt werden, der während der Auswertung anstelle des „Starten"-Buttons erscheint.

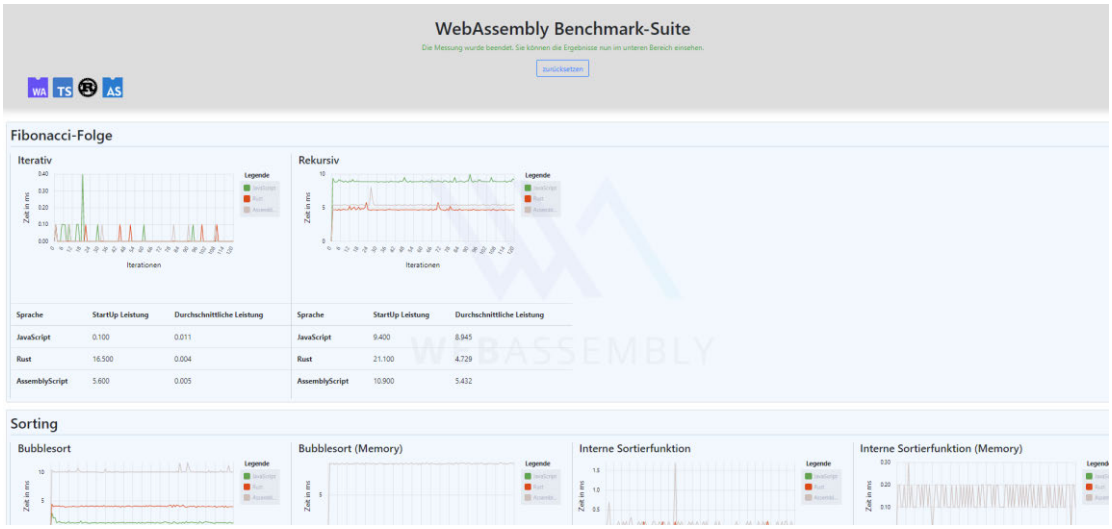

Figure 9 - Benutzungsoberfläche nach Durchlaufen des Benchmark-Prozesses

4.3 Architektur

Die Anwendung wird in einer komponentenorientierten Architektur realisiert. Diese wird vom Angular-Framework vorgegeben. Zwar können in einer Angular-Anwendung einige Parallelen zum MVC[29]- oder MVVM[30]-Entwurfsmuster gezogen werden, fest zugeordnet werden kann die Architektur aber keinem der beiden, da weder Controller noch ViewModels verwendet werden. In dieser komponentenorientierten Architektur ist die Anwendung hierarchisch in Angular-Modulen aufgebaut. Jedes dieser Module erfüllt dabei eine bestimme Funktionalität und ist wiederum aus Angular-Komponenten zusammengesetzt.

Einen detaillierteren Einblick in die Struktur der Anwendung sollen die nachfolgenden Abschnitte bieten. Dabei werden kurz angular-spezifische Eigenarten erläutert, welche maßgeblichen Einfluss auf die Architektur haben. Anschließend wird die Architektur zunächst nach Vorbild des arc42-Templates[31] in verschiedenen Detailstufen präsentiert und darauf basierend der zentrale Ablauf des Benchmark-Prozesses erläutert.

4.3.1 Angular

Angular gibt eine komponentenorientierte Architektur vor. Im Wesentlichen ist diese Architektur in voneinander unabhängigen Modulen organisiert. Jedes der Module sollte dabei eine bestimmte Funktionalität erfüllen und in sich abgeschlossen sein. Die Module wiederum können als eine Sammlung weiterer Framework-Bausteine, wie Komponenten und Services betrachtet werden. Innerhalb dieser Module besteht eine hohe Kohäsion, was bedeutet, dass Komponenten und Services eng miteinander zusammenarbeiten und nicht für andere Module verantwortlich sind. Die Funktionalität eines Moduls kann anderen Modulen allerdings zur

[29] Model View Controller

[30] Model View ViewModel

[31] https://arc42.de/

Verfügung gestellt werden, indem die entsprechenden Komponenten der Module exportiert werden.

Eine Komponente stellt in der Anwendung den grundlegendsten Baustein der Benutzungsoberfläche dar. Sie besteht wiederum aus mehreren Elementen, wobei das Template, bestehend aus einer HTML- und einer CSS-Datei die View der Komponente bildet und eine TypeScript-Klasse die Logik zur Steuerung dieser View bereitstellt.

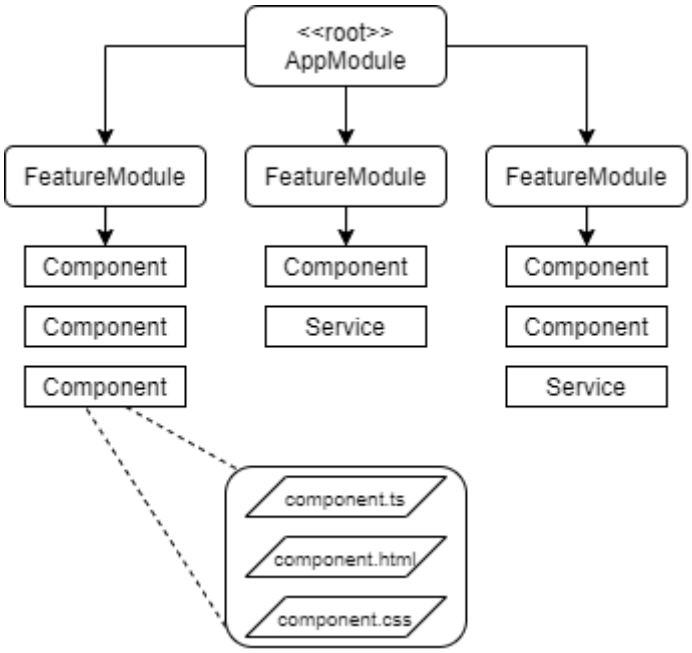

Figure 10 - Architektur einer Angular-Anwendung

Wie in der Abbildung (Figure 10) ersichtlich, steht in jeder Angular-Anwendung das AppModule an oberster Stelle, es bildet das sogenannte „root module“. In diesem Modul wird die Funktionalität zum Starten der Anwendung bereitgestellt. Außerdem sind hier alle weiteren Module des Systems enthalten. Zusätzlich befindet sich in diesem Modul eine Komponente. Diese Komponente bildet in der Regel die Hauptview, was bedeutet, dass alle weiteren Komponenten auf dieser View enthalten und somit zum Gesamtsystem zusammengesetzt sind. Ein anschauliches Beispiel für ein Modul in einer Anwendung wäre ein Dashboard. Ein solches Dashboard-Modul besteht aus verschiedenen Elementen, wie Diagramme, Informationslisten

und Bereichen die Grafiken darstellen. Jedes dieser Elemente kann dabei eine eigene Komponente sein. Zusätzlich dazu könnte das Modul einen Service enthalten, der für das Anfordern der Daten aus einem Back-End-System verantwortlich ist und diese Daten an die Komponenten weiterleitet.

4.3.2 Aufbau

Zunächst wird das System in der folgenden Abbildung (Figure 11) als Blackbox dargestellt. Auf ihr wird die technische sowie fachliche Kontextabgrenzung des Systems verdeutlicht.

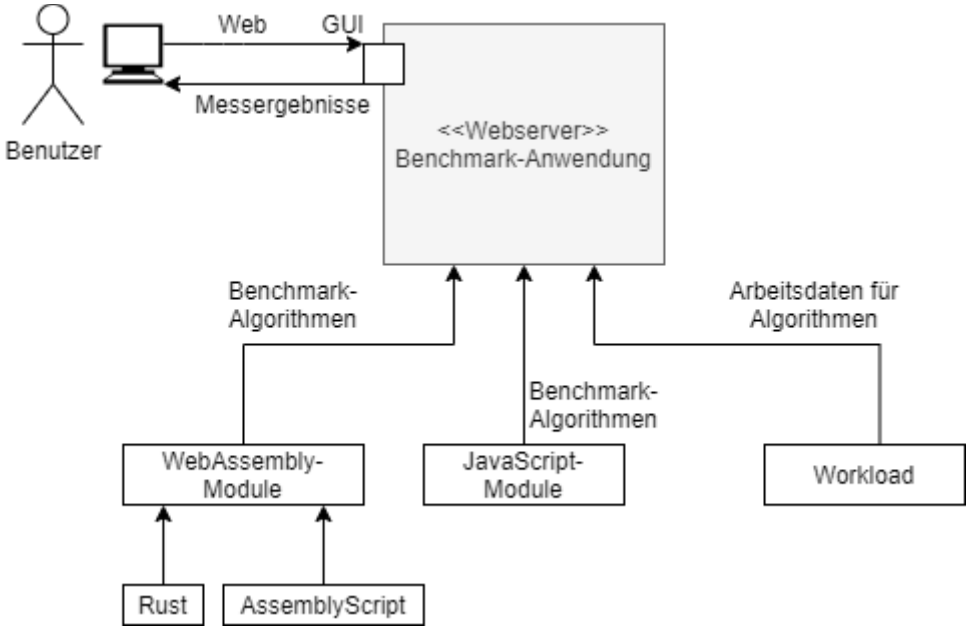

Figure 11 - Kontextabgrenzung der Benchmark-App

Da es sich um eine reine Front-End-Anwendung handelt, fällt der technische Kontext übersichtlich aus. Aus diesem Grund wurde hier auf eine rein technische Kontextabgrenzung verzichtet. Für den Betrieb der Anwendung genügt lediglich ein Webserver. Da alle relevanten Daten über dieses Front-End bereitgestellt werden, sind hier keine weiteren Systeme, wie ein Back-End-System mit Datenbank notwendig. Die Bedienung der Anwendung durch den

Benutzer erfolgt über die grafische Benutzungsoberfläche. Diese Benutzungsoberfläche kann mit jedem gängigen Webbrowser aufgerufen werden. Die Anwendung soll dem Vergleich zwischen den Technologien JavaScript und WebAssembly dienen. Diese Technologien werden als Module in die Benchmark-Anwendung importiert. Wie in der Abbildung ersichtlich werden die WebAssembly-Module jeweils aus den Sprachen Rust und AssemblyScript kompiliert. In diesen Modulen befinden sich die Algorithmen, die in der Benchmark-Anwendung jeweils mit einem bestimmten Workload gespeist und anschließend ausgeführt werden. Innerhalb der Anwendung wird jeweils die Zeit gemessen, welche die jeweilige Technologie zur Ausführung der Algorithmen benötigt hat. Die daraus errechneten Messergebnisse werden anschließend auf der Benutzungsoberfläche angezeigt und somit an den Benutzer zurückgegeben.

Das folgende fachliche Datenmodel (Figure 12) gibt einen detaillierteren Einblick auf die in der Anwendung enthaltenen Elemente und ihre Funktionen.

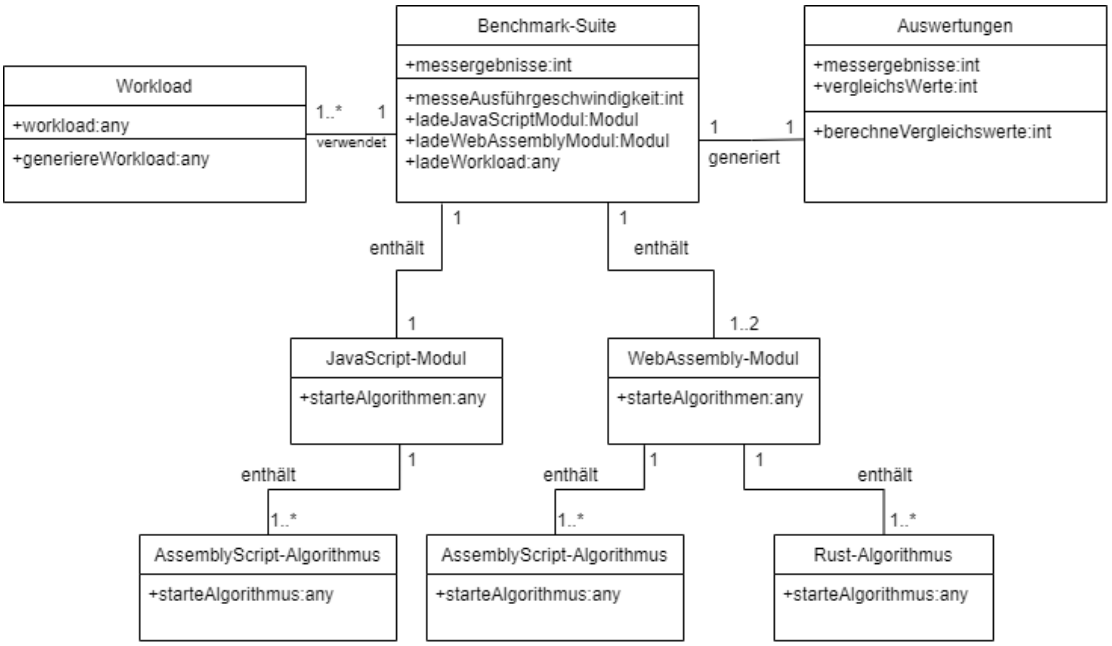

Figure 12 - Fachliches Datenmodel der Benchmark-App

Diese zu vergleichenden Technologien liegen als Wasm-Module und JavaScript-Module vor. Jedes dieser Module kann, wie im Datenmodell zu erkennen ist, mehrere AssemblyScript- und Rust-Algorithmen enthalten, genauso, wie das JavaScript-Modul mehrere JavaScript-

Algorithmen enthalten kann. Die Benchmark-Suite muss die entsprechenden Module laden, wobei jede Suite immer nur ein JavaScript-Modul und höchstens zwei WebAssembly-Module besitzt. Die geladenen Module werden dann mit dem generierten Workload als Parameter ausgeführt. Nach Ausführung des Benchmark-Prozesses, werden die Auswertungen generiert, in denen die Vergleichswerte enthalten sind. Mit diesen Vergleichswerten können die Technologien untereinander verglichen werden können.

In dem BPMN (Figure 13) wird der Benchmark-Prozess nochmal genauer beleuchtet.

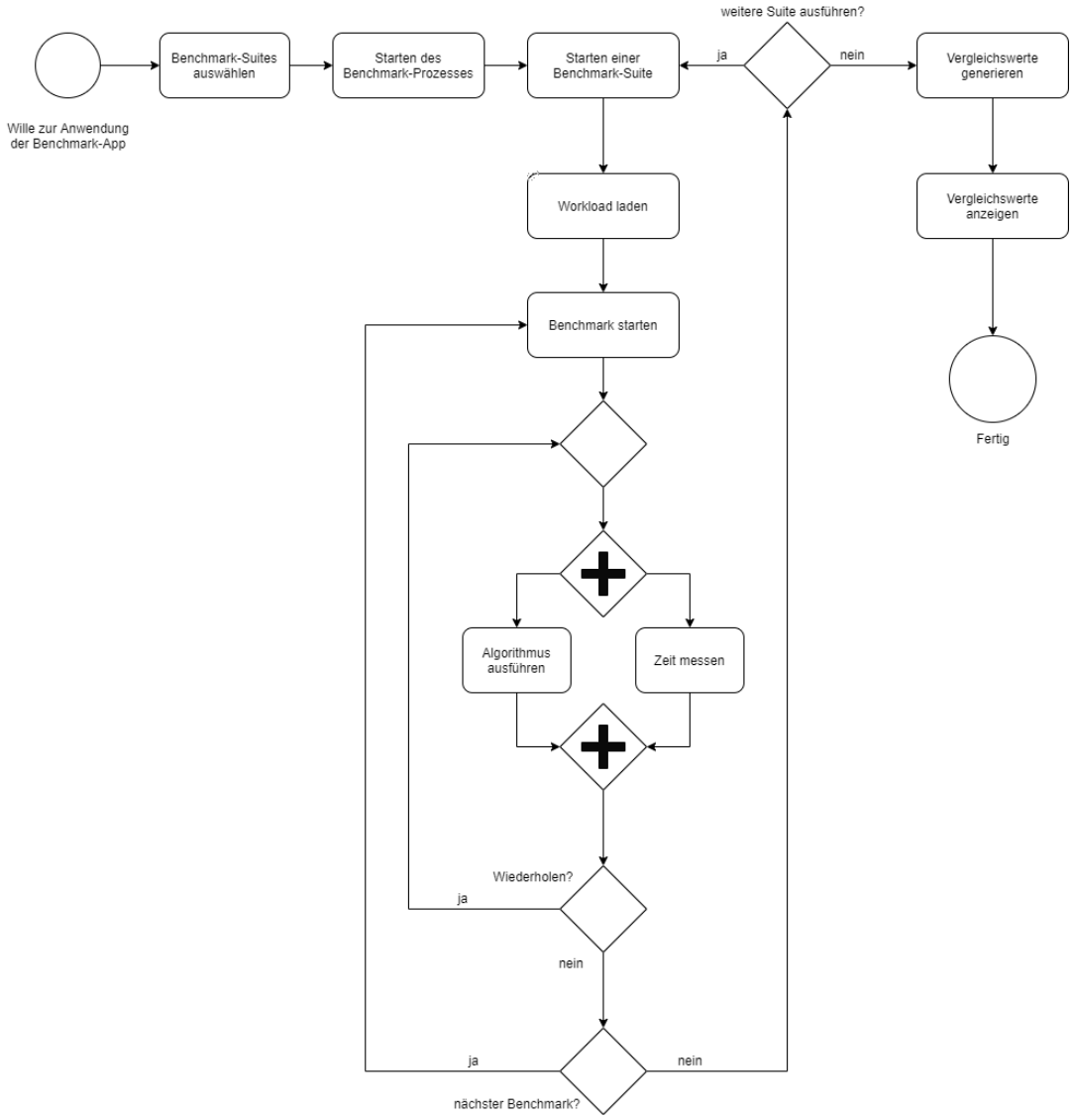

Figure 13 - BPMN des Benchmark-Prozesses

Es soll dem Benutzer möglich sein, die gewünschten Benchmark-Suites auszuwählen. Nachdem er das getan hat, startet er den Benchmark-Prozess. Während dieses Prozesses, werden alle gewählten Benchmark-Suites nacheinander durchlaufen. Innerhalb einer jeden Suite wird dabei der Workload geladen und anschließend jeder Benchmark dieser Suite durchlaufen. Dabei kann jeder Benchmark beliebig oft wiederholt werden, während bei jedem dieser Durchläufe die Zeit gemessen wird. Nachdem der Benchmark-Prozess abgeschlossen ist, werden die Vergleichswerte generiert und anschließend angezeigt.

Mit der nun folgenden Ansicht (Figure 14) wird begonnen die Anwendung aus technischer Sicht zu betrachten. Dazu werden hier die Module der Anwendung abgebildet und diese somit auf oberster Ebene betrachtet.

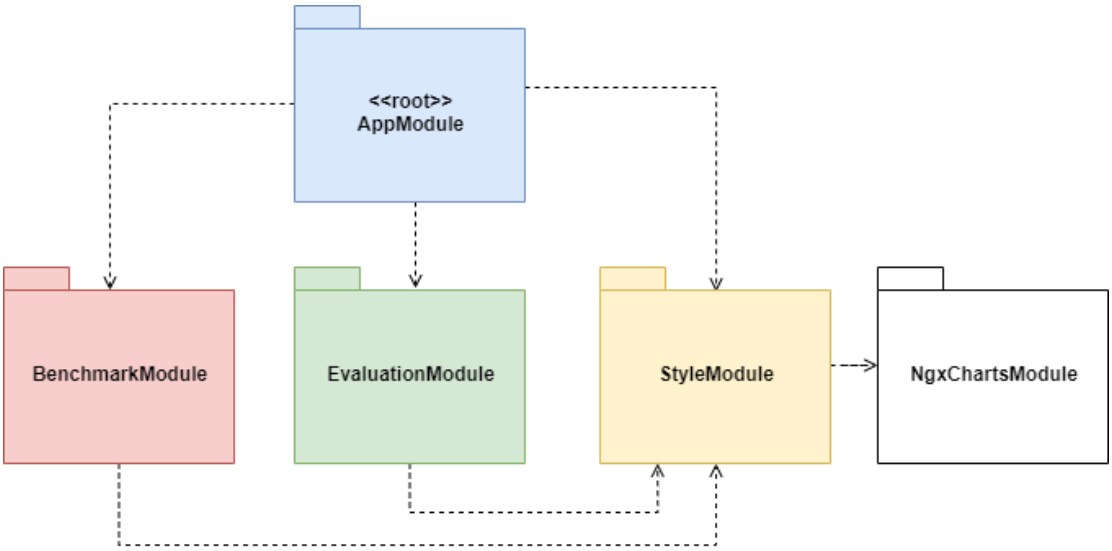

Figure 14 - Angular-Module der Benchmark-App

Die Abbildung verdeutlicht, wie sich die Anwendung in fünf Module unterteilt. Zugunsten der Übersichtlichkeit wurden hier nur die Module abgebildet, welche die für die Erfüllung des Zwecks dieser Anwendung geschriebenen Logik enthalten. Angular-spezifische Module, welche interne Funktionalitäten bereitstellen, werden hier folglich nicht mit aufgeführt. Die Module wurden farblich hervorgehoben. Dies soll die Zuordnung der Komponenten zu den Modulen in den nachfolgenden Abbildungen erleichtern.

Das AppModule (Blau) befindet sich hier an oberste Stelle. Es enthält alle weiteren Module sowie eine Komponente auf dessen View die Benutzungsoberfläche und somit alle weiteren Komponenten der Anwendung gerendert werden. Diese Module sind funktional geschlossene Einheiten und tragen entsprechend dieser Funktion sinnvolle Bezeichnungen. Nachfolgend werden die Module kurz beschrieben.

BenchmarkModul

Das BenchmarkModul enthält alle Benchmarks der Anwendung. Jede Benchmark-Suite wird dabei wiederum in einer eigenen Komponente realisiert, die jeweils aus einer TypeScript-Klasse und dem dazugehörigen Template besteht. Die TypeScript-Klasse enthält dabei die Logik zum Laden und Ausführen der in dieser Komponente enthaltenen Benchmarks. Über das Template wird jeweils ein Eingabeformular gerendert, über das der Benutzer die Konfigurationen für diesen Benchmark vornehmen kann. Neben den Suites ist in diesem Modul außerdem noch eine Komponente mit dem Namen „BenchmarkCardComponent“ zu finden. Diese Komponente hat eine rein grafische Funktion. Jede der Benchmark-Suites, wird über diese BenchmarkCardComponent in die Anwendung eingebunden und kann somit durch das in dieser Komponente enthaltene Template mit den dazugehörigen CSS-Regeln als Benchmark-Karte auf der grafischen Oberfläche gerendert werden. Durch die Implementierung dieser Komponente, muss der Code für die grafische Oberfläche sowie die dazugehörige Logik für jede dieser Karten nur einmal geschrieben werden und kann für beliebig viele Benchmark-Suites wiederverwendet werden. Dies wurde zu Gunsten der Wartbarkeit entschieden, da falls nötig, der Code nur an einer Stelle angepasst werden muss. So kann durch die Wiederverwendung viel Code eingespart wird, was erheblich zur Übersichtlichkeit beiträgt.

EvaluationModule

Das EvaluationModule ist für das Entgegennehmen und Darstellen der in den Benchmark-Suites gemessenen Ergebnisse verantwortlich. Dieses Modul enthält zwei Komponenten, die EvaluationPageComponent und die SuiteEvaluationChartComponent. Die EvaluationPageComponent nimmt dabei alle Ergebnisse entgegen und rendert anschließend für jede Benchmark-Suite

jeweils eine Instanz der SuiteEvaluationChartComponent. Die SuiteEvaluationChartComponent hat die Aufgabe, die entgegengenommenen Ergebnisse in präsentationsfähige Daten umzuwandeln, damit diese korrekt und sinnvoll in den Liniendiagrammen sowie Auswertungstabellen dargestellt werden können. Anschließend verwendet diese Komponente die LineChartComponent aus dem StyleModule und rendert diese wiederum für jedes Ergebnis einer Suite. Die LineChartComponent enthält das Liniendiagramm aus dem NgxChartsModule und alle dazugehörigen Konfigurationen sowie CSS-Regeln für eine übersichtliche Darstellung.

StyleModule

Im StyleModule befinden sich Komponenten, die ausschließlich eine grafische Funktion haben. Diese Komponenten können in jedem Modul der Anwendung wiederverwendet werden, ohne dabei an die konkrete Funktion des jeweiligen Moduls gebunden zu sein. In diesem Modul befinden sich zwei Komponenten, die ButtonComponent und die LineChartComponent. Die ButtonComponent führt die Funktion eines normalen HTML-Buttons aus. Da in der Anwendung Buttons mit speziellen Designs und angepasstem Verhalten, wie Animationen benötigt werden, wurde diese Funktionalität in einer eigenen Komponente gekapselt. Dadurch kann diese ohne großen Overhead, in der gesamten Anwendung beliebig oft wiederverwendet werden. Die Art des Buttons und damit das Design, kann durch einen Input, den „ButtonType“, bestimmt werden. Die zweite Komponente in diesem Modul, die LineChartComponent wurde bereits im EvaluationModule erwähnt. Ihre Aufgabe ist es, Daten als Input entgegenzunehmen und diese dann grafisch in einem Liniendiagramm zu visualisieren.

NgxChartsModule

Dieses Modul wird nur vom StyleModule verwendet. Es wurde nicht speziell für diese Anwendung implementiert, sondern stammt aus einer externen Bibliothek und wurde als Abhängigkeit

aus dem NPM-Repository zu dieser Anwendung hinzugefügt. Es wurde von Swimlane[32] entwickelt und stellt neben dem Liniendiagramm noch eine Reihe weiterer Diagrammarten bereit.

Neben den gerade beschriebenen Modulen und Komponenten, enthält die Anwendung noch eine Reihe weiterer Elemente, wie Typen, Interfaces und Funktionen, auf die im weiteren Verlauf noch näher eingegangen wird. Einen Überblick über die in der Anwendung maßgeblich zur Funktion beitragenden Elemente, gibt das Klassendiagramm (Figure 15) auf der nächsten Seite.

[32] https://swimlane.gitbook.io/ngx-charts/

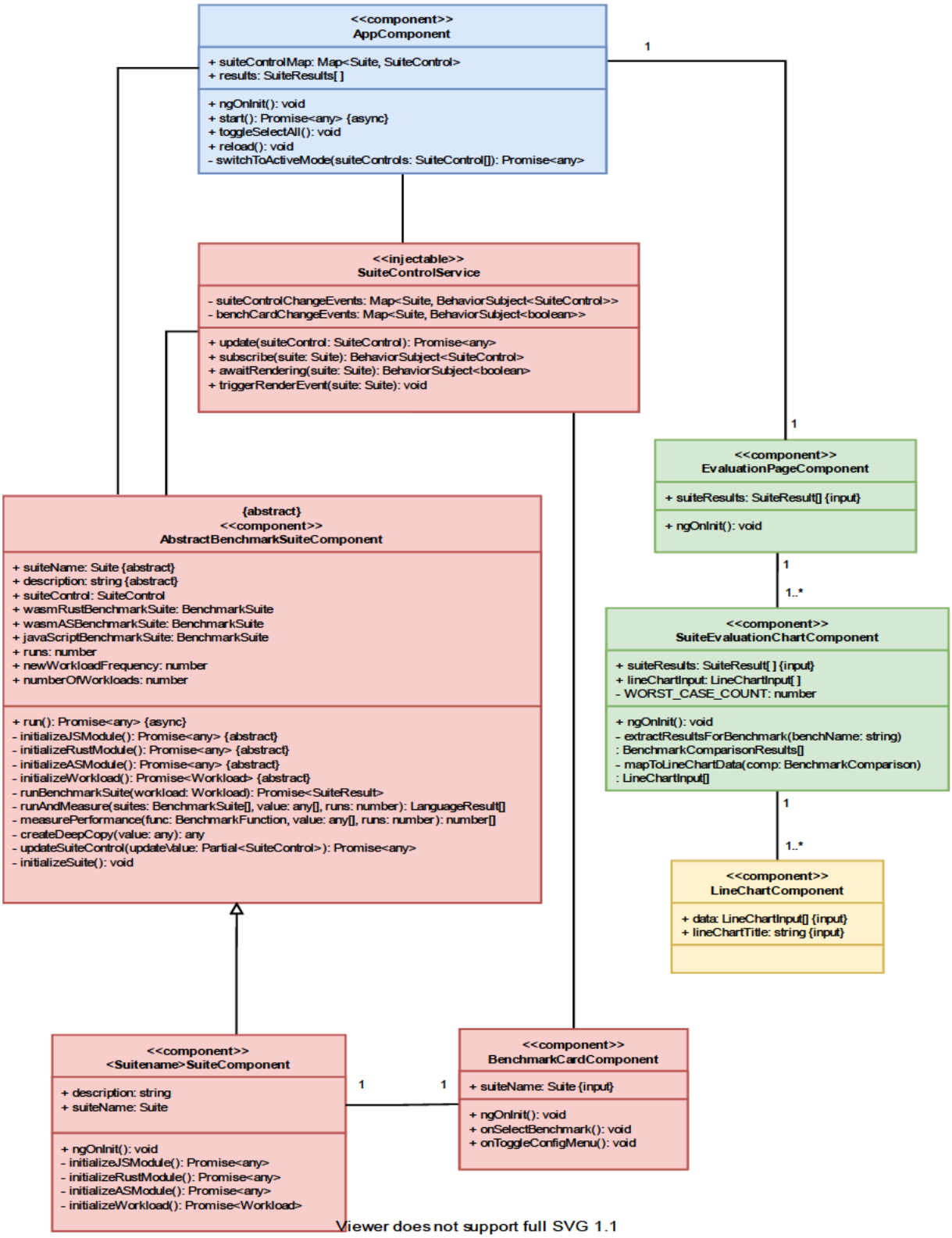

Figure 15 - Klassendiagramm der Benchmark-App

Diese Abbildung verfeinert die Ansicht auf die Anwendung um eine weitere Stufe. Abgebildet sind hier die wichtigsten Klassen, deren Struktur und wie sie miteinander in Verbindung stehen. Durch die farbliche Hervorhebung kann hier nachvollzogen werden, welchem Modul die Klassen angehören. Die Struktur der Klassen bildet jeweils deren relevanten Funktionen und Felder ab. Zur besseren Übersicht wurde in diesem Diagramm wieder überwiegend auf Angular-spezifische Elemente verzichtet, die beispielsweise zum Rendern und Bedienen von grafischen Elementen wie Formular- und Inputfelder zwingend erforderlich sind.

Zu finden ist in diesem Diagramm die abstrakte Klasse **AbstractBenchmarkSuite**. Von ihr erben alle Benchmark-Suite-Klassen. Verdeutlicht werden soll das durch die darunterliegende Klasse mit der Bezeichnung **<Suitename>SuiteComponent**. Diese Klasse steht stellvertretend für alle von der abstrakten Klasse erbenden konkreten Klassen. Jede dieser erbenden Klassen ist in einer eigenen Komponente realisiert. Der Platzhalter <Suitename> im Namen der Komponente steht dabei für die Bezeichnung der jeweiligen Benchmark-Suite. Eine in dieser Anwendung implementierte Komponente heißt so z.B. „FibonacciSuiteComponent“. Die abstrakte Klasse gibt die Felder und Funktionen vor, die von den konkreten Klassen initialisiert bzw. implementiert werden müssen, sowie die Reihenfolge, in der diese Funktionen aufgerufen werden. Unter anderem wird in den Feldern Name und Beschreibung der jeweiligen Suite gehalten, sowie die Anzahl der Iterationen, die Frequenz, wie oft der Workload geändert werden soll und die Anzahl der Workloads, die sich daraus ergeben. Die Funktion, mit der diese Klasse angesprochen wird, ist die Funktion *#run*. In dieser Funktion findet die Initialisierung statt. Dazu zählt das Laden der auszuführenden Module, das Generieren des Workloads, sowie die Initialisierung des Zustandes der Suite. Anschließend werden die Algorithmen der Module nacheinander durchlaufen und währenddessen die Ausführungsgeschwindigkeit gemessen und gespeichert. Diese Funktionen sind ebenfalls in der abstrakten Klasse implementiert. Der Großteil, der für die Benchmark-Suites wichtigen Funktionen ist somit bereits in der abstrakten Klasse implementiert. Zurück bleiben vier abstrakte Funktionen, die von den konkreten Benchmark-Suite-Klassen implementiert werden müssen. In diesen Funktionen werden die JavaScript- und WebAssembly-Module geladen, sowie der Workload, mit denen die zu messenden Algorithmen der Module ausgeführt werden. Darüber hinaus wird in der *#ngOnInit*-Funktion das Eingabefeld einer jeden Suite initialisiert, in denen die Konfiguration zu den Workloads vorgenommen werden kann. Neben den bereits erwähnten Feldern in der abstrakten

Klasse, ist außerdem das Feld, welches das SuiteControl-Objekt hält, von besonderer Bedeutung. Dieses Objekt enthält eine Reihe von Properties zur Steuerung der jeweiligen Benchmark-Suite sowie zur Speicherung ihres Zustands. Nach Instanziierung einer Benchmark-Suite-Klasse, wird dieses SuiteControl-Objekt initialisiert. Nach dieser Initialisierung wird die #*update*-Funktion im **SuiteControlService** aufgerufen und dieses Objekt dabei an den Service übergeben. Der SuiteControlService bildet das zentrale Kommunikationselement zwischen den Komponenten. Durch Aufruf dieser Funktion, wird über ein sogenanntes „BehaviorSubject“, welches in diesem Service für jede verfügbare Suite der Anwendung gehalten wird, ein Event ausgelöst. Jede Komponente, die zuvor die #subscribe-Funktion in diesem Service aufgerufen hat und somit für die entsprechende Suite eine Subscription abgeschlossen hat, wird durch dieses Event über die Zustandsänderung der entsprechenden Suite benachrichtigt. In jeder Komponente der Anwendung kann somit der aktuelle Zustand der Suites abgefragt und gehalten werden. Eine Komponente, die immer den aktuellen Zustand der Suites benötigt ist z.B. die **BenchmarkCardComponent**. Wie bereits erwähnt, verwendet jede Suite eine Instanz dieser Komponente zum Rendern der Benchmark-Karten auf der Benutzungsoberfläche. Ein SuiteControl-Objekt enthält u.a. die Information, ob eine Suite momentan ausgeführt wird, bereits durchlaufen wurde oder selektiert, aber noch nicht an der Reihe war. Die grafischen Elemente der BenchmarkCardComponent, können so auf den Zustand der Suites reagieren und somit z.B. die Ladeanimation einblenden. Zusätzlich zu der #*subscribe*-Funktion kann aber auch die #*update*-Funktion im SuiteControlService von anderen Klassen zwecks Steuerung der Suites aufgerufen werden. Unter anderem greift so z.B. die **AppComponent** auf die SuiteControl-Objekte zu und kann somit nach Klick auf den Start-Button alle zuvor ausgewählten Benchmark-Suites starten. Die AppComponent ist dem AppModule angesiedelt und befindet sich in der Hierarchie somit an oberster Stelle. Auf der dazugehörenden View werden die Views aller weiteren Komponenten gerendert. Die AppComponent bildet so die zentrale Steuerung der Anwendung. Sie hält mit dem Feld „suiteControlMap“ die SuiteControl-Objekte und somit den Zustand aller Benchmark-Suites der Anwendung. Über die Funktion #*start* werden die selektierten Suites der Reihe nach gestartet und deren Messergebnisse anschließend im Feld „results“ gespeichert.

Neben den bisher erwähnten Klassen sind in diesem Diagramm außerdem die Klassen des EvaluationModules zu finden. Wie schon zuvor erläutert, werden diese benötigt, um

Messergebnisse der Suites grafisch in Diagrammen darzustellen. Die **EvaluationPageComponent** nimmt alle Messergebnisse aus dem Feld „results“ der AppComponent entgegen und instanziiert für jede Suite eine **SuiteEvaluationChartComponent**. Diese Komponente errechnet aus diesen Ergebnissen wiederum die Daten, die zur Auswertung benötigt werden und instanziiert anschließend für jede Suite eine **LineChartComponent**. Auf den Templates dieser Komponenten wird dann jeweils ein Liniendiagramm mit einer Auswertungstabelle gerendert.

Abschließend soll zur Beschreibung des Aufbaus der Anwendung noch eine weitere Abbildung (Figure 16) dienen. Durch diese Abbildung wird ersichtlich, wie die Templates der einzelnen Komponenten aus den Modulen miteinander verknüpft sind und somit die gesamte grafische Oberfläche bilden.

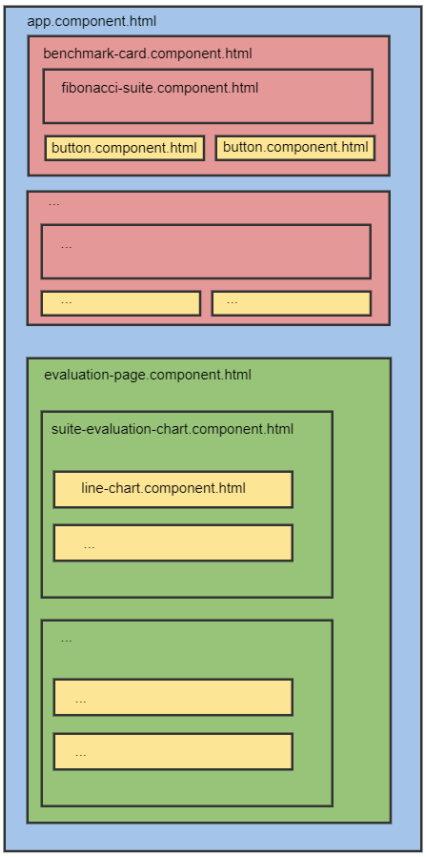

Figure 16 - Anordnung der Views der Benchmark-App

Auch hier blau hervorgehoben, ist die „app.component.html“. Sie befindet sich auf der obersten Ebene und enthält alle weiteren Templates. Diese Templates werden durch sogenannte Selektoren in Form von HTML-Tags auf der app.component.html angeordnet und dadurch in den DOM-Tree eingefügt. Zusätzlich können die Templates durch diverse CSS-Regeln, in die für die Anwendung benötigt grafische Form gebracht werden. Die Benchmark-Karten werden hier durch die „benchmark-card.component.html“ dargestellt. Auf ihr werden die Input-Felder der einzelnen Benchmark-Suite-Komponenten gerendert, in dieser Abbildung beispielhaft mit der „fibonacci-suite.component.html“ dargestellt. Außerdem befinden sich auf der „benchmark-card.component.html“ die Buttons der ButtonComponent, welche aus dem StyleModule wiederverwendet werden. Die nach der benchmark-card.component.html folgende Fläche mit den „…“-Bezeichnungen soll abbilden, dass diese Komponente, abhängig von der Anzahl der Benchmark-Suites, beliebig oft auf der app.component.html gerendert werden kann. Im darunterliegenden Teil der Abbildung ist die „evaluation-page.component.html“ zu finden. Hier wird deutlich, wie die „suite-evaluation.component.html“ für das Ergebnis einer jeden Suite gerendert wird. Für jedes Messergebnis der darin befindlichen Benchmarks wird dann nochmals auf dessen Template eine „line-char.component.html“ gerendert. Auch dort sollen die Flächen mit den „…“-Bezeichnungen darstellen, dass die entsprechenden Komponenten beliebig oft vorkommen bzw. gerendert werden können.

4.3.3 Ablauf

Der im vorherigen Abschnitt erläuterte Aufbau hat die wichtigsten Elemente der Anwendung aufgezeigt und wie sie miteinander in Verbindung stehen. Die zentrale Funktion dieser Elemente ist das Ausführen, Messen und Aufzeigen der Ergebnisse der Benchmarks. Im folgenden Abschnitt sollen anhand eines Sequenzdiagrammes (Figure 17) die wichtigsten Schritte dieses Prozesses aufgezeigt werden:

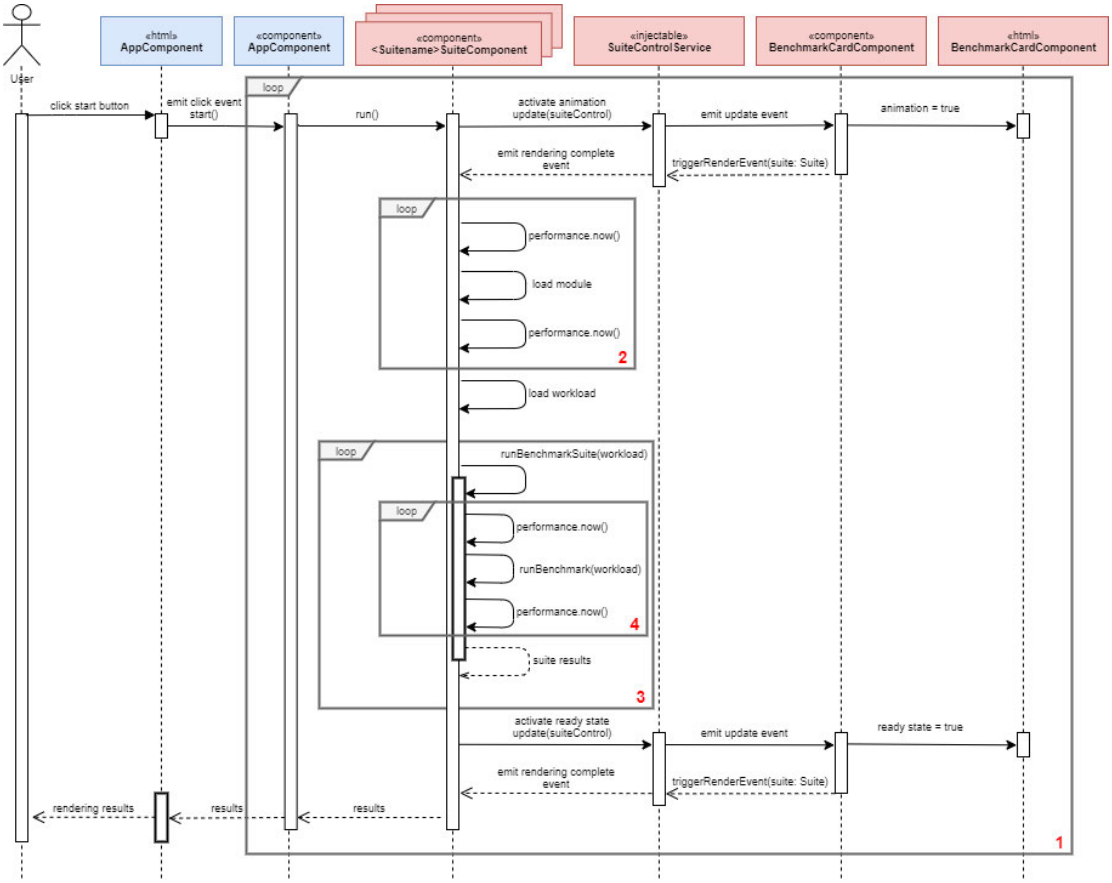

Figure 17 - Sequenzdiagramm der Benchmark-App

1. Nachdem vom Benutzer die von ihm gewünschten Benchmark-Suites selektiert und alle Konfigurationen vorgenommen wurden, betätigt er den Startbutton, der in der AppComponent gerendert wird.
2. Durch betätigten des Startbutton wird die #*run*-Funktion in der AppComponent ausgeführt. In dieser Funktion werden alle selektierten Benchmark-Suites der Reihe nach ausgeführt (Schleife 1), in dem die in dem SuiteControl-Objekt enthaltene #*run* -Funktion der jeweiligen Suites aufgerufen wird.
3. Innerhalb der #run-Funktion der Suites werden nun wieder der Reihe nach, die folgenden Prozesse durchlaufen:
    a. Aktivieren der Ladeanimation der jeweiligen Suite durch Aufruf der #*update*-Funktion im SuiteControlService.
        i. Im SuiteControlService wird ein Updateevent emittiert, sodass die BenchCardComponent, in der die Suite gerendert wird, das aktuelle SuiteControl-Objekt erhält.
        ii. Durch Erhalten des SuiteControl-Objekts erfährt die BenchCardComponent den aktuellen Zustand der Suite und rendert die Animation, die aufzeigen soll, dass der Benchmark-Prozess in dieser Suite gerade durchlaufen wird.
        iii. Nachdem die Animation gerendert wurde, ruft die BenchCardComponent die #triggerRenderEvent-Funktion im SuiteControlService auf, um ein Signal zurückzugeben, dass das Rendern abgeschlossen ist und die Suite mit Benchmark-Prozess fortfahren kann.
    b. Im nächsten Schritt folgt eine Schleife (Schleife 2), in der die JavaScript und Wasm-Module geladen werden, die die zu messenden Algorithmen der Suite enthalten.
    c. Vor und nach jedem Ladevorgang eines jeden Moduls, wird die Zeit mit der Funktion #*performance.now* gemessen. Dies hat den Hintergrund, dass die Zeit, die benötigt wird, um die Module zu laden, in die spätere Auswertung mit einfließt.
    d. Nachdem die Module geladen wurden, wird der Workload initialisiert. Dieser Workload unterscheidet sich je nach Benchmark-Suite. So kann dieser z.B. im

Falle der Fibonacci-Suite aus nur einer Zahl bestehen und im Falle der RegEx-Suite aus einem Text mit 1 Mio. Zeichen.

e. Anschließend folgt der eigentliche Benchmark-Prozess in zwei geschachtelten Schleifen. Jede Benchmark-Suite verfügt zwischen einem und mehrere Benchmarks, dies bedeutet, dass jeder Benchmark-Algorithmus, einer jeden Technologie, also JavaScript, Wasm-Rust und Wasm-AssemblyScript durchlaufen werden muss. Dies soll durch die äußere Schleife (Schleife 3) verdeutlicht werden.

f. Innerhalb dieser Schleife wird jeder Benchmark-Algorithmus, abhängig von der eingestellten Iterationsanzahl mehrmals durchlaufen. Dies soll durch die innere Schleife (Schleife 4) verdeutlich werden.

g. Vor und nach jeder Iteration wird wieder die Zeit mit *#performance.now* gemessen und gespeichert.

h. Nachdem die Algorithmen jeder Technologie durchlaufen wurden, wird der Zustand, dass der Benchmark-Prozess der Suite abgeschlossen hat wieder über die *#update*-Funktion des SuiteControlServices übermittelt. Somit wird die Ladeanimation in der BenchCardComponent in eine „Fertig“-Meldung geändert.

i. Zuletzt werden die gespeicherten Messergebnisse zurück an die AppComponent übermittelt.

4. In der AppComponent wird daraufhin die EvaluationPageComponent mit Liniendiagrammen und Auswertungstabellen gerendert und dem Benutzer somit die Ergebnisse präsentiert. Dieser Punkt wird im Sequenzdiagramm nicht im Detail abgebildet.

Synchrone Kommunikation

Worauf im Entwurf und in der darauffolgenden Implementierung der Anwendung vor allem geachtet werden muss, ist, dass alle Aufrufe während des Benchmark-Prozesses synchron ausgeführt werden. Dies bedeutet in Bezug auf diese Anwendung, dass der nächste Schritt im Prozess erst dann ausgeführt werden darf, wenn der davor beendet wurde. Vor allem die Kommunikation zwischen den Komponenten erfolgt in Angular oft asynchron. Events, wie das

Updateevent im SuiteControlService, das u.a. im Schritt 3 des zuvor beschriebenen Ablaufes emittiert wird, werden meist nicht synchron von allen Empfängern empfangen. Dieses Verhalten kann dazu führen, dass Ladeanimationen nicht zum gewünschten Zeitpunkt gerendert werden, sondern erst nachdem alle darauffolgenden Funktionen ausgeführt wurden. Des Weiteren kann es vorkommen, dass diese asynchronen Aufrufe während des Benchmark-Prozesses ausgeführt werden, was die Messergebnisse verfälschen würde. Aus diesem Grund muss die auf das Updateevent reagierende Komponente immer ein Renderevent emittieren. Auf dieses Renderevent kann die Komponente, die ursprünglich die *#update*-Funktion aufgerufen hat, warten, wodurch sichergestellt werden kann, dass alle anderen Prozesse fertig ausgeführt wurden.

4.3.4 Wasm-Module

Die Wasm-Module werden in eigenen Projekten realisiert, d.h. zu jedem Modul existiert ein eigenes Projekt dessen Architektur losgelöst von der Architektur der Benchmark-Anwendung ist. Die Projekte bestehen jeweils nur aus den, in den Benchmark-Suites auszuführenden Funktionen. Die Signaturen der Funktionen stellen somit die Schnittstelle des jeweiligen Moduls dar, was bedeutet, dass diese Funktionen direkt von der Benchmark-Anwendung aufgerufen werden und der Workload dabei als Paramater übergeben wird.

4.4 Messverfahren

Wie bereits im Ablauf beschrieben, wird jeder Benchmark der Anwendung mehrmals durchlaufen. Die Anzahl der Durchläufe ist auf 120 voreingestellt und kann vom Benutzer nach Belieben angepasst werden. Wie in den Grundlagen beschriebenen, ist ein mehrmaliges Ausführen eines jeden Benchmarks sinnvoll, da die Technologien bei mehrmaligen Durchlaufen derselben Algorithmen mit gleichem Workload verschiedene Verhalten aufweisen. So findet bei JavaScript beispielsweise eine Optimierung durch den JIT-Compiler statt, wodurch die Ausführung nach einer gewissen Zeit schneller werden kann. Gleichzeitig kann sich die Ausführungsgeschwindigkeit verlangsamen, da sowohl JavaScript als auch AssemblyScript einen

Garbage Collector benötigen, der in regelmäßigen Abständen ausgeführt wird. Neben der Anzahl der Durchläufe kann außerdem die Frequenz eingestellt werden, durch die festgelegt wird, nach wieviel Durchläufen ein neuer Workload verwendet wird. Auf diese Weise lassen sich realistischere Szenarien nachbilden, da derartige Algorithmen in richtigen Anwendungen nur selten mit denselben Daten arbeiten. Dadurch lässt sich ermitteln, wie die Technologien und ihre Optimierungen bei gleichen Algorithmen mit veränderten Daten reagieren.

Während die Algorithmen in den Benchmarks also mehrmals durchlaufen werden, wird vor und nach jedem dieser Durchläufe, die Zeit gemessen und die Differenz dieser Zeitpunkte in Millisekunden gespeichert. In der Auswertung am Ende des Benchmark-Prozesses, können so aus den gesammelten Messergebnissen der Benchmarks jeweils zwei aussagekräftige Werte errechnet werden, die Startzeit und die durchschnittliche Ausführungszeit. Die Startzeit wird dabei aus der Zeit errechnet, die benötigt wird, um die Module mit den Algorithmen der jeweiligen Technologien herunterzuladen und zu initialisieren, addiert mit der Ausführungsdauer des ersten Durchlaufs der Algorithmen. Dieser Wert ist wichtig, da eine schnelle Reaktionszeit gerade im Browserumfeld von größerer Bedeutung ist, damit die entsprechenden Internetseiten schneller aufgebaut werden können. Die durchschnittliche Ausführungszeit wird errechnet aus dem Durchschnitt der Messergebnisse aller Durchläufe, abgesehen vom ersten, da der erste meist sehr viel höher ist als der Rest. Aus diesem Wert lässt sich ableiten, ob die Anwendung mit dieser Technologie überhaupt lauffähig ist.

4.5 Benchmarks

Entsprechend der Anforderungen wurden für die Anwendung eine Reihe von Benchmarks gewählt, die es zu implementieren gilt. Diese Benchmarks sind entsprechend ihrer Themen zu Benchmark-Suites zusammengefasst. In der nachfolgenden Tabelle (Tabelle 1) werden alle Suites mit den darin enthaltenen Benchmarks aufgelistet. Zusätzlich beschreibt die Spalte „Art“, was mit den entsprechenden Benchmarks evaluiert werden soll.

Tabelle 1 - Alle Suites mit Benchmarks und deren Zweck

Benchmark-Suite	***Benchmarks***	***Art***
Fibonacci	rekursiv, iterativ	rechenintensiv, ohne Kommunikation
Hanoi	iterativ	rechenintensiv, ohne Kommunikation
Sorting	Bubblesort, interne Sortierfunktion	rechenintensiv, Übertragen von Zahlen
Regex	Strings ersetzen, Vorkommen zählen	Übertragen von Strings
Base64	Text kodieren, Zahlen kodieren	Übertragen von Strings und Zahlen
Crypto (AES)	AES-128, AES-256	rechenintensiv, Übertragen von Strings
Hash (SHA)	SHA-256, SHA-512	rechenintensiv, Übertragen von Strings
Bildbearbeitung	Blur-Effekt	rechenintensiv, Übertragen von komplexen Objekten
DOM-Manipulation	DOM-Elemente erzeugen und löschen	Kommunikation mit Web-API testen
Transfer	Nummern übertragen, Zahlen übertragen	reiner Geschwindigkeitstest beim Übertragen

5 Implementierung

Basierend auf den im vorherigen Kapitel beschriebenen Entwurfsentscheidungen wird nun die Implementierung der Anwendung behandelt. Dabei wird die Anwendung zuerst im Allgemeinen beschrieben und danach vor allem auf die Umsetzung der zuvor beschriebenen Prozesse, anhand konkreter Codebeispiele eingegangen. Anschließend werden die implementierten Benchmarks sowie deren Validierung näher beleuchtet. Am Ende des Kapitels wird aufgezeigt, ob die definierten Anforderungen mit der hier implementierten Anwendung erfüllt wurden.

5.1 Verzeichnisstruktur

Wie die Architektur, ist auch die Verzeichnisstruktur (Figure 18) durch Angular als Grundgerüst vorgegeben. Sie wird bei Erstellung des Projekts generiert und bildet den Angular-Workspace. Auf der Stammebene sind u.a. mehrere Dateien zu finden. Von erwähnenswerter Bedeutung sind hier die Dateien *angular.json* und die *package.json*.

Die angular.json bietet Arbeitsbereichsweite und projektspezifische Konfigurationsvorgaben für Build- und Entwicklungswerkzeuge, die von der Angular-CLI bereitgestellt werden.

Die package.json enthält neben der Beschreibung des Projektes wie Name und Version die Abhängigkeiten des Projekts, d.h. alle externen Pakete bzw. Bibliotheken, die in diesem Projekt verwendet werden, dazu gehören auch die implementierten Wasm-Module. Außerdem ist es möglich BASH-Skripte in dieser Datei anzulegen, die z.B. der Verwaltung des Projektes dienen können.

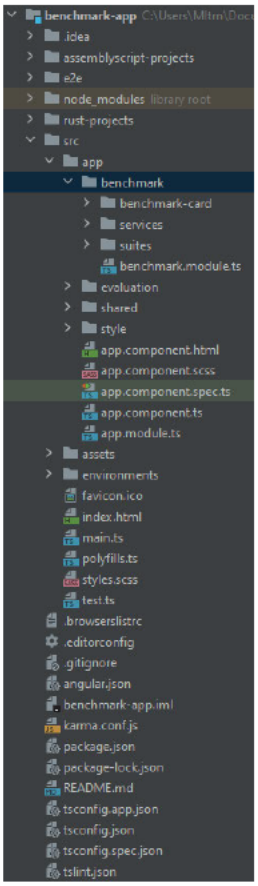

Figure 18 - Verzeichnisstruktur

Neben diesen beiden Dateien sind hier mehrere Unterverzeichnisse zu finden. Erwähnenswert sind darunter vor allem „assemblyscript-projects“, „rust-projects“, „node_modules“ und das „src“-Verzeichnis.

Wie bereits erwähnt wurden die Rust- und AssemblyScript-Projekte jeweils in eigenen Projekten realisiert. Zugunsten einer einfacheren Handhabbarkeit bzgl. Erweiterung und Wartung, liegen diese Projekte im selben Stammverzeichnis wie das Hauptprojekt. Diese Projekte sind in den Verzeichnissen rust-projects bzw. assemblyscript-projects zu finden. Zusätzlich dazu, wurden in der package.json Skripte realisiert, die für den Import dieser Projekte als Wasm-Module in das Hauptprojekt dienen sollen. So kann z.B. mit dem Konsolenbefehl `npm run build:as:fibonacci` bewirkt werden, dass das in AssemblyScript implementierte Fibonacci-Projekt automatisch zu einem Wasm-Modul kompiliert, und anschließen in das korrekte Verzeichnis des Hauptprojekt kopiert wird. Weitere Benchmarks können so durch Einfügen in die Verzeichnisstruktur und erweitern der Skripte einfacher dem Projekt hinzugefügt werden.

In dem Unterverzeichnis *node_modules* befinden sich alle installierten Abhängigkeiten des Projektes. Dazu zählen alle Angular-spezifischen Bibliotheken sowie alle zusätzlich installierten durch NPM-Pakete hinzugekommenen Bibliotheken.

Im *src*-Verzeichnis befindet sich der Quellcode der Benchmark-Anwendung. Dieser enthält wieder eine Reihe automatisch generierter Dateien. Dazu zählt die *styles.css*, welche die globalen CSS-Regeln für das Projekt enthält wie auch die *main.ts* und *index.html* die hauptsächlich zum Booten der Anwendung benötigt werden. Außerdem unterteilt sich das Verzeichnis nochmals in die drei weitere Unterverzeichnisse. Im *environments*-Verzeichnis befinden sich Dateien mit globalen Konfigurationen der Anwendung. Im *assets*-Verzeichnis werden alle Dateien gespeichert, die nach dem Bauen der Anwendung unverändert mit in die Anwendung integriert werden sollen. Unter anderem werden in diesem Verzeichnis auch die Wasm-Module gespeichert. Im *app*-Verzeichnis liegt der eigentlich Quellcode der Anwendung. Hier zu finden sind die im Architektur-Kapitel beschriebenen Module *AppModule* sowie alle weiteren Module mit logisch zur Funktion gewählten Bezeichnern *benchmark*, *evaluation* und *style*. Im *shared*-Verzeichnis befinden sich Typdefinition und Hilfsfunktionen, die in mehreren Modulen der Anwendung verwendet werden.

5.2 Prozesse

Nachfolgenden werden nun die bereits in den Entwurfsentscheidungen beschriebenen Prozesse anhand ausgewählter Codebeispiele näher erläutert. Dazu zählt die Initialisierung der Anwendung, welche durchlaufen wird, sobald die Anwendung vom Benutzer aufgerufen wird. Außerdem wird der Benchmark-Prozess beschrieben, der nach betätigen des Start-Buttons ausgeführt wird und die Umsetzung der Synchronität aller Prozesse, die in diesem Projekt eine wichtige Rolle spielt. Neben den Codebeispielen werden Ausschnitte aus den Templates und damit aus der View der jeweiligen Komponenten dargestellt. Dadurch kann besser nachvollzogen werden, wie die Komponenten miteinander in Verbindung stehen.

5.2.1 Synchronität

Wie bereits beschrieben, finden in dieser Anwendung gezwungenermaßen asynchrone Prozesse statt, die zwecks korrekter Ausführung der Benchmarks synchronisiert werden müssen. Der Grund dafür ist, dass Angular dem asynchronen Programmierparadigma „reactive programming“ folgt. Bei diesem Paradigma erfolgt die Kommunikation zwischen den Komponenten der Anwendung nach dem Observer-Pattern. Dabei verwaltet ein Objekt, welches Subject genannt wird eine Liste von Beobachtern, die durch Auslösen eines Events automatisch über die daraus folgenden Zustandsänderungen benachrichtigt werden.

In diesem Projekt wird diese Kommunikation über den SuiteControlService realisiert. Dieser Service erzeugt bei Initialisierung für jede vorhandene Benchmark-Suite ein sogenanntes BehaviorSubject. Dieses BehaviorSubject ist Teil der von Angular verwendeten RxJS-Bibliothek und implementiert das Observer-Pattern. Über den entsprechenden Funktionsaufruf kann anschließend jede Komponente der Benchmark-Anwendung ein Beobachter für die gewünschte Suite werden und wird dadurch über Zustandsänderungen informiert. Da das Erzeugen und Empfangen der Events durch das BehaviorSubject allerdings nicht synchron verläuft, kann es vorkommen, dass Empfänger nicht direkt auf die Events reagieren und somit zuvor ungewollt andere Prozesse ausgeführt werden. Damit dieses Problem nicht auftreten kann, müssen diese

Prozesse synchronisiert werden, was bedeutet, dass alle Funktionsaufrufe der Reihe nach und nicht parallel ausgeführt werden.

Die Komponente die zwingend synchron auf Zustandsänderungen reagieren muss, ist die BenchCardComponent. Sie wird von den Suites jeweils vor und nach Durchlaufen des Benchmark-Prozesses über den SuiteControlService benachrichtigt und muss daraufhin die View korrekt rendern, sodass die dortigen Elemente zu den korrekten Zeitpunkten des Prozesses ein- bzw. ausgeblendet werden. Aus diesem Grund werden bei Initialisierung des SuiteControlServices im Konstruktor zwei BehaviorSubject-Objekte für jede Suite erzeugt (Listing 4).

```
1  private readonly suiteControlChangeEvents: Map<Suite,<SuiteControl>>
2      = new Map<Suite, BehaviorSubject<SuiteControl>>();
3
4  private readonly benchCardChangeEvents: Map<Suite, BehaviorSubject<boolean>>
5      = new Map<Suite, BehaviorSubject<boolean>>();
6
7  constructor() {
8      Object.values(Suites).forEach(value => {
9        this.suiteControlChangeEvents.set(value, new  BehaviorSubject<SuiteControl>(null));
10       this.benchCardChangeEvents.set(value, new BehaviorSubject<boolean>(null));
11     });
12   }
```

Listing 4 - Die zwei Maps des SuiteControlServices

Diese BehaviorSubject -Objekte werden in zwei Maps gehalten, wie in Zeile 1 und 4 zu sehen ist. Die *suiteControlChangeEvents*-Map aus Zeile 1 enthält die zuvor beschriebenen BehaviorSubject-Objekte. Durch sie können Events zur Zustandsänderungen der Suites ausgelöst und beobachtet werden. Die *benchCardChangeEvents*-Map aus Zeile 4 enthält nochmals ein BehaviorSubject-Objekt für jede Suite. Die Events dieser Objekte werden allerdings ausschließlich durch die BenchCardComponent ausgelöst. Auf das Auslösen dieses Events wird in der *#update*-Funktion gewartet, um ein synchrones Verhalten zu erzwingen (Listing 5).

```
1  update(suiteControl: SuiteControl): Promise<any> {
2
3      return new Promise<any>(resolve => {
4
5        this.awaitRendering(suiteControl.suite)
6          .pipe(take(1))
7          .subscribe(_ => {
8              resolve();
9          });
10
11       this.suiteControlChangeEvents.get(suiteControl.suite).next(suiteControl)
12     });
13 }
```

Listing 5 - Update-Funktion des SuiteControlServices

Wenn diese Funktion aufgerufen wird, wird ein Promise-Objekt erzeugt. Promises sind ebenfalls ein Konstrukt asynchroner Programmierung. Die Funktion, die innerhalb eines Promise-Objekts ausgeführt wird, kann entweder asynchron oder blockierend ausgeführt werden. Im Falle einer blockierenden Ausführung, wird der Code, der diese Funktion aufgerufen hat, erst dann fortgeführt, wenn die zum Promise gehörende *#resolve*-Funktion (Zeile 8) aufgerufen wird. Diese *#resolve*-Funktion wird hier erst aufgerufen, wenn ein benchCardChange-Event ausgelöst wurde. Das Warten dieses Events wird durch Aufruf der Funktion *#awaitRendering* (Zeile 5) ausgelöst. Erzeugt wird dieses Event durch die BenchCardComponent, und zwar erst dann, wenn die BenchCardComponent auf die Zustandsänderung der entsprechenden Suite reagieren konnte.

Folgendes Sequenzdiagram (Figure 19) soll den eben geschilderten Ablauf nochmal verdeutlichen.

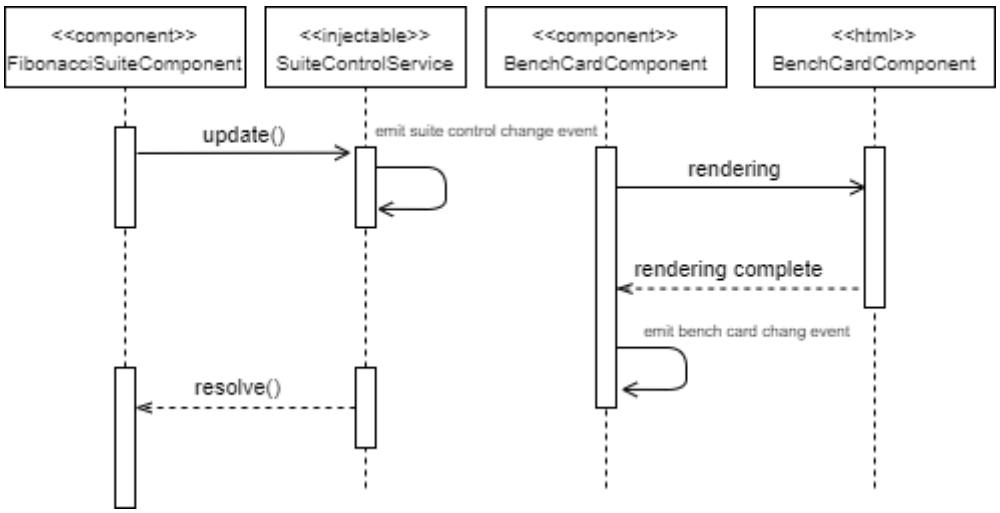

Figure 19 - Erzwungene Synchronisation durch Warten auf die Events

Zu beachten ist hierbei, dass das „rendering“ in der BenchCardComponent erst dann ausgeführt wird, wenn das „suite control change event“ ausgelöst wurde. Nachdem das Rendern abgeschlossen wurde, wird durch die BenchCardComponent ein „bench card change event“ ausgelöst. Erst dann wird die *#resolve*-Funktion des Promis-Objekts im SuiteControlService aufgerufen und erst dann wird der Prozess in der FibonacciSuiteComponent fortgesetzt.

Neben der Kommunikation, finden noch zwei weitere asynchrone Prozesse in der Anwendung statt, die zwingend synchronisiert werden müssen. Diese Prozesse werden im weiteren Verlauf des Kapitels beschrieben.

5.2.2 Initialisierung

Wird die Benchmark-App über einen Browser aufgerufen, so wird auf oberster Ebene die AppComponent gerendert, auf ihr befinden sich alle weiteren Views der Anwendung. Folgende Abbildung zeigt einen Ausschnitt aus der Implementierung der AppComponent-View:

```
1  <div class="col-3" *ngIf="showColumn[0]">
2      <app-benchmark-card [suiteName]="suiteNames.fibonacci">
3        <app-fibonacci></app-fibonacci>
4      </app-benchmark-card>
5
6      <app-benchmark-card [suiteName]="suiteNames.base64">
7        <app-base64></app-base64>
8      </app-benchmark-card>
9
10     <app-benchmark-card [suiteName]="suiteNames.image">
11       <app-image-editing></app-image-editing>
12     </app-benchmark-card>
13 </div>
```

Listing 6 - Ausschnitt aus der View der AppComponent

Zu erkennen sind in diesem Ausschnitt drei Benchmark-Suites, die durch Platzieren der entsprechenden Selektoren innerhalb der BenchCardComponent-Selektoren, auf der View gerendert werden. Zu erkennen ist außerdem, dass jede Suite über das *suiteName*-Attribute den jeweiligen Namen der Suite als Input erhält. Diese Namen sind eindeutige Bezeichner der Suites und werden anschließend von den BenchCardComponent-Instanzen benötigt, um sich als Beobachter auf die Zustandsänderungen der jeweiligen Suites im SuiteControlService anzumelden.

Durch das Rendern der BenchCardComponent mit der jeweiligen Suite darin, werden diese automatisch instanziiert und initialisiert. Neben diesen Komponenten wird außerdem automatisch eine Instanz des SuiteControlService durch das Angular-Framework erzeugt und durch den Konstruktor in jede Komponente injiziert. Der SuiteControlService ermöglicht, wie bereits beschrieben, die Kommunikation zwischen den Komponenten und spielt zudem eine zentrale Rolle.

Während der Initialisierung einer jeder Benchmark-Suite wird die *#update*-Funktion des SuiteControlServices zum ersten Mal aufgerufen. Dabei versendet jede Suite ein SuiteControl-Objekt mit dem initialen Zustand als Parameter. Folgendes Listing (Listing 7) zeigt einen Ausschnitt aus der *#initializeSuite*-Funktion der Benchmark-Suites. In dieser Funktion findet die Initialisierung der Suite statt.

```
1  protected initializeSuite(): void {
2      this.updateSuiteControl(
3        {
4          run: () => this.run(),
5          suite: this.suiteName,
6          title: this.suiteName,
7          description: this.description,
8          showCard: true,
9          validConfig: true,
10       }
11     ).then();
12
13 ...
```

Listing 7 - Ausschnitt aus der #initializeSuite-Funktion der Benchmark-Suites

Zu erkennen sind in diesem Ausschnitt ein Teil der Properties des SuiteControl-Objekts (Zeile 4-9), mit denen der initiale Zustand der Suite übermittelt wird. Ganz oben ist die *#run*-Funktion der Suite zu erkennen, deren Referenz im SuiteControl-Objekt gespeichert wird und somit von der AppComponent zum Starten der jeweiligen Suite verwendet werden kann. Des Weiteren wird mit diesem Aufruf der Name der Suite übertragen. Der Name ist ein eindeutiger Bezeichner in dieser Anwendung und wird vom SuiteControlService benötigt, um ein SuiteControl-Change-Event für die passende Suite auszulösen. Die Beschreibung wird von der BenchCard-Component benötigt, um diese auf der BenchCard zu rendern. Je nach Zustand des *showCard*-Properties wird die entsprechende BenchCard gerendert oder nicht und das *validConfig*-Propertie zeigt an, ob die getätigten Konfigurationen valide sind. Falls nicht, kann die Suite nicht gestartet werden und es wird eine entsprechende Nachricht gerendert. Neben den Properties ist hier außerdem der *#then*-Aufruf in Zeile 11 von Bedeutung. Wie im Kapitel Synchronität geschildert, kann die *#update*-Funktion im SuiteControlService jeweils asynchron oder blockierend aufgerufen werden. Durch Aufruf dieser Funktion *#then* wird diese Funktion asynchron aufgerufen. Dies ist an dieser Stelle der Fall, da hier kein Prozess zwingend synchronisiert

werden muss. Die Initialisierung kann somit asynchron stattfinden, was den Ablauf dieses Prozesses beschleunigen kann. Durch den Aufruf der *#update*-Funktion, der durch jede Benchmark-Suite bei der Initialisierung erfolgt, haben die AppComponent sowie alle BenchCard-Component-Instanzen der aktuellen Zustand der Suites erhalten und die Initialisierung der Anwendung ist nun abgeschlossen.

5.2.3 Benchmark-Prozess

Durch Betätigen des Starten-Buttons in der Benutzungsoberfläche, wird die *#start*-Funktion der AppComponent aufgerufen. Innerhalb dieser Funktion wird nun der Zustand einer jeden Suite durch Aufruf der Funktion *#switchToActiveMode* angepasst (Listing 8).

```
1  private switchToActiveMode(suiteControls: SuiteControl[]): Promise<any> {
2      const updateEvents: Promise<any>[] = [];
3
4      suiteControls.forEach(suite => {
5        if (suite.isSelected) {
6          updateEvents
7            .push(this.suiteControlService.update({
8              ...suite,
9              ...{isActive: true, showConfigMenu: false}
10           }));
11       } else {
12         updateEvents
13           .push(this.suiteControlService.update({
14             ...suite,
15             ...{showCard: false}
16           }));
17       }
18     });
19
20     ...
21
22     return Promise.all(updateEvents);
```

Listing 8 - Ausschnitt aus der #switchToActiveMode-Funktion der Benchmark-Suites

Dabei werden die SuiteControl-Objekte aller Suites durchlaufen und so durch das *isSelected*-Property geprüft, ob diese vom Benutzer selektiert wurden oder nicht. Wenn eine Suite selektiert wurde, wird diese als aktiv markiert und das Konfigurationsmenu wird geschlossen (Zeile 9), andernfalls wird das *showCard*-Property auf *false* gesetzt und damit bewirkt, dass die nicht

selektierten Karten während des Benchmark-Prozesses ausgeblendet werden. Diese Zustandsänderung wird durch die *#update*-Funktion des SuiteControlServices ausgeführt, der in diesem Fall als synchroner Aufruf stattfindet. Erwirkt wird diese Synchronität durch den Aufruf *Promise.all* in Zeile 22. Die *Promise.all*-Funktion gibt ein Promise-Objekt zurück, welches erfüllt wird, sobald alle Promise-Objekte der übergebenen Liste erfüllt wurden. In diesem Fall sind

```
1 async start(): Promise<any> {
2
3     ...
4
5     await this.switchToActiveMode(suiteControls);
6
7     ...
8
9 }
```

Listing 9 - Aufruf der *#switchToActiveMode*-Funktion

dies alle Promise-Objekte, die jeweils durch die *#update*-Funktionen zurückgegeben werden. In Kombination mit dem Aufruf der oben gezeigten Funktion mit dem *await*-Operator, wie in Listing 9 gezeigt, kann ein synchrones Verhalten erzwungen werden. Der *await*-Operator erwirkt, dass der damit verbundene Funktionsaufruf erst komplett ausgeführt werden muss, bevor mit dem Rest Funktion fortgefahren wird.

Auf dieselbe Weise wird auch die Synchronität innerhalb der *#run*-Funktionen einer jeden Suite ermöglicht, die jetzt der Reihe nach ausgeführt werden.

Bevor in der entsprechenden Suite der Benchmark-Prozess gestartet wird, wird wieder eine Zustandsänderung auf synchrone Weise ausgeführt (Listing 10).

```
1  async run(): Promise<any> {
2
3      ...
4
5      await this.updateSuiteControl(
6        {
7          isRunning: true,
8        }
9      );
10
11     ...
12
13 }
```

Listing 10 - Zustandsänderung der Suite bei Start des Benchmark-Prozesses

Diese Zustandsänderung bewirkt, dass die „Wird Ausgeführt"-Animation auf der jeweiligen BenchmarkCard gerendert wird. Anschließend folgt der Hauptteil der Funktion (Listing 11).

```
async run(): Promise<any> {

    ...

    const jsStartTime = performance.now();
    this.javaScriptBenchmarkSuite = await this.initializeJSModule();
    startUpTimes.set(Languages.javaScript, performance.now() - jsStartTime);

    const rustStartTime = performance.now();
    this.wasmRustBenchmarkSuite = await this.initializeRustModule();
    startUpTimes.set(Languages.rust, performance.now() - rustStartTime);

    const asStartTime = performance.now();
    this.wasmASBenchmarkSuite = await this.initializeASModule();
    startUpTimes.set(Languages.assemblyScript, performance.now() - asStartTime);

    const workload = await this.initializeWorkload();

    let suiteResult: SuiteResult = await this.runBenchmarkSuite(workload)
       .catch(error => {
              suiteError = error;
              return null;
    });

    ...

}
```

Listing 11 - Laden der Wasm-Module, Generieren des Workloads, Starten der Suite

Dabei werden zuerst die Module mit den Benchmark-Algorithmen geladen. Da diese Module erst zum Zeitpunkt des Benchmark-Prozesses benötigt werden, werden diese auch erst in diesem Moment vom Browser des Benutzers heruntergeladen und initialisiert. Da dies ebenfalls ein asynchroner Prozess ist, wird hier wieder der *await*-Operator verwendet, um ein synchrones Verhalten zu erzwingen. Das Laden der Module muss vorallem aus dem Grund synchron sein, da an dieser Stelle die erste Messung des Benchmark-Prozesses stattfindet. Gemessen wird dabei die Zeit die benötigt wird, um die entsprechenden Module zu laden. Während dieses Messvorgangs dürfen keine anderen Prozesse stattfinden, die den Ladeprozess beeinflussen könnten. Gemessen werden die Zeiten jeweils vor und nach jedem Ladevorgang eines Moduls und dabei die Differenz gespeichert. Zum Messen der Zeit wird hier sowie im

Rest der Anwendung, die von JavaScript bereitgestellte Funktion *#performance.now()* verwendet.

Die *#performance.now()* Methode gibt einen DOMHighResTimeStamp zurück, der in Millisekunden gemessen wird und auf fünf Tausendstel einer Millisekunde (5 Mikrosekunden) genau ist. [17]

Nachdem das Laden und Initialisieren der Module abgeschlossen ist, wird in Zeile 17 der Workload geladen und dieser anschließend in Zeile 19 der *#runBenchmarkSuite*-Funktion als Parameter übergeben. In dieser Funktion werden die geladenen Module durchlaufen und dabei jeweils jeder Algrithmus dieser Module in der Funktion des folgendenen Ausschnitts ausgeführt (Listing 12).

```
1  protected measurePerformance(func: BenchmarkFunction, workloads: any[], runs: number):
2  number[] {
3
4      const results = [];
5      let wlIndex = 0;
6
7      for (let i = 1; i <= runs; i++) {
8        const workload = this.makeDeepCopy(workloads[wlIndex]);
9
10       const startTime = performance.now();
11
12       func(workload);
13
14       const endTime = performance.now();
15
16       results.push(endTime - startTime);
17
18       if (workloads[wlIndex + 1] && (i % this.newWorkloadFrequency) === 0) {
19         wlIndex++;
20       }
21     }
22
23     return results;
24 }
```

Listing 12 - Wiederholtes Ausführen der Benchmarks und gleichzeitiger Messung

Dieser Funktion wird der auszuführende Algorithmus als Parameter übergeben, sowie eine Liste aller Workloads und die Anzahl, wie oft der Algorithmus ausgeführt werden soll. Anschließend wird der übergebene Algorithmus in einer For-Schleife wiederholt ausgeführt. Dabei wird am Anfang einer jeder Iteration eine deep copy vom übergebenen Workload erstellt. Dies ist nötig, da TypeScript nur Referenzen in den Variablen speichert und auch nur diese bei einer Wertzuweisung auf eine andere Variable übergeben werden. Würde also beispielsweise bei einer ungeordneten Liste mit Zahlenwerten eine Sortierung durchgeführt werden, so wäre

diese Liste nach der ersten Iteration sortiert und würde in den darauffolgenden Iterationen in bereits sortierter Form an den Algorithmus übergeben werden. Durch das Erstellen einer deep copy, wird so bei jeder Iteration eine ungeordnete Liste übergeben. Nachdem diese deep copy also erstellt wurde, wird der Algorithmus anschließend durchlaufen, dabei wird die Zeit gemessen und die Differenz anschließend gespeichert. Am Ende einer jeden Iteration wird geprüft, ob entsprechend der eingestellten *newWorkloadFrequency* ein neuer Workload geladen werden muss.

Nachdem alle Algorithmen durchlaufen wurden, ist der Benchmark-Prozess der Suite abgeschlossen und die gesammelten Messwerte werden zurück an die AppComponent übermittelt. Diese gibt die Werte, wie im Kapitel „Ablauf“ beschrieben, weiter an die EvaluationComponent, wo diese zusammen mit den Ergebnissen der anderen Suite ausgewertet und in der View gerendert werden.

5.3 Benchmarks

Nachdem auf die wichtigsten Prozesse der Anwendungen eingegangen wurde, werden jetzt die in den verschiedenen Sprachen implementierten Benchmarks näher erleutert. Zuvor muss angemerkt werden, dass sich im Zuge der Implementierung gezeigt hat, dass nicht jeder Benchmark mit jeder Technologie realisiert werden konnte. Vor allem trifft dies auf AssemblyScript zu. Wie bereits beschrieben, handelt es sich bei AssemblyScirpt um eine noch sehr junge Sprache. Aus diesem Grund gibt es bisher nur eine dürftige Unterstützung durch interne oder externe Bibliotheken. Komplexe Algorithmen, wie AES oder die Manipulation des DOM-Trees konnten in dieser Sprache daher nicht implementiert werden.

Es wurde versucht die Benchmarks in möglichst gleicher Struktur zu implementiert. Da für einige Algorithmen allerdings bereits externe sowie interne Bibliotheken existieren, wäre eine Neuimplementierung nicht sinnvoll gewesen, da diese Bibliotheken bereits hinreichend getestet und dementsprechend als performante Lösungen gelten. In den nachfolgenden Erläuterungen wird zu jeder Benchmarks-Suite angegeben, in welchen Sprachen sie realisiert werden

konnte, ob Bibliotheken verwendet wurden und mit welcher Art von Workload die Algorithmen arbeiten.

5.3.1 Kommunikation mit WebAssembly

Um eine Kommunikation zwischen den Wasm-Modulen und der in JavaScript geschriebenen Webanwendung zu ermöglichen, müssen diverse Vorkehrungen getroffen werden. Dazu gehört u.a. das Exportieren der in Rust und AssemblyScript geschriebenen Funktionen, sodass diese nach dem Laden des Moduls von JavaScript angesprochen werden können. Wie bereits im Grundlagen-Kapitel beschrieben, handelt es sich bei WebAssembly um eine maschinennahe Sprache. Aus diesem Grund verfügt diese Technologie in der momentanen Version über nur vier primitive Datentypen. Komplexe Datentypen wie Strings oder gar in JavaScript definierte Datentypen werden nicht direkt unterstützt. Um mit diesem Problem umgehen zu können, wurden von den Communities weitere Tools entwickelt. Dabei wurden von den Entwicklern der Technologien jeweils unterschiedliche Ansätze verwendet. Wie diese Ansätze genauer aussehen und inwiefern sich diese unterscheiden, wird im Kapitel Evaluierung genauer erläutert.

5.3.2 Fibonacci

Für die Fibonacci-Suite wurde jeweils eine iterative sowie rekursive Version des Algorithmus in jeder der drei Sprachen implementiert. Vor allem die rekursive Version weist durch ihre exponentielle Komplexität schnell eine lange Laufzeit auf, wodurch sie sich gut für das Testen einer komplexen Berechnung eignet. Außerdem muss hier keine große Kommunikation zwischen Wasm-Modul und Anwendung stattfinden.

Als Parameter wird den Funktionen jeweils nur eine Zahl übergeben, die Angibt, wie weit die Fibonacci-Folge berechnet werden soll. Aus diesem Grund ist hier keine Generierung eines speziellen Workloads erforderlich.

5.3.3 Hanoi

Die Hanoi-Suite enthält den klassischen Türme von Hanoi-Algorithmus in rekursiver Form. Implementiert wurde der Algorithmus wieder in allen Sprachen. So wie bei der Fibonacci-Suite wird auch hier ein hoher Berechnungsaufwand kombiniert mit geringer Kommunikation geboten.

Als Workload wird hier lediglich die Anzahl der Scheiben mitgegeben.

5.3.4 Base64

Die Base64-Suite wurde ebenfalls in jeder Sprache realisiert. Für die Realisierung wurden externe Bibliotheken eingesetzt. Der Einsatz dieser Bibliotheken ist sinnvoll, da diese die Algorithmen für jede der Sprachen auf performante Weise implementieren. Umgesetzt wurde jeweils die Kodierung von Strings und Zahlen.

Die Generierung des Workloads geschieht in ausgelagerten Funktionen. Dabei wurde die externe Bibliothek „random-words[33]“ für die zu kodierenden Strings verwendet und die von TypeScript mitgelieferte Bibliothek „Math“ zum Generieren der Zahlen.

[33] https://www.npmjs.com/package/random-words

Die Funktionen (Listing 13) generieren dabei einen String mit einer beliebigen Anzahl von zufälligen Wörtern und eine ungeordnete Liste beliebiger Länge mit Zufallszahlen zwischen 0 und 255.

```
export function createRandomNumberList(numberOfNumbers: number): number[] {
  const numberList = [];

  for (let i = 0; i < numberOfNumbers; i++) {
    numberList.push(createRandomNumber());
  }

  return numberList;
}

export function createRandomNumber(max = 255): number {
  return Math.floor(Math.random() * Math.floor(max));
}

export function createRandomText(numberOfWords: string): string {
  return randomWords({exactly: numberOfWords, join: ' '});
}
```

Listing 13 - Funktionen zum Generieren des Workloads

5.3.5 Sortierung

Für die Sortierung-Suite wurde in jeder Sprache der Bubblesort implementiert. Außerdem wurde in jeder Sprache jeweils eine interne von der jeweiligen Sprache mitgelieferte Sortierungsfunktion verwendet.

Als Workload wird hier jeweils eine Liste mit Zufallszahlen beliebiger Länge mitgegeben. Diese wurde ebenfalls mit der in Listing (Listing 13) beschrieben Funktion generiert.

5.3.6 Crypto

Die Crypto-Suite implementiert den AES-Algorithmus in der 128-Bit- und der 256-Bit-Version. Realisiert werden konnte der Algorithmus nur in JavaScript und Rust. In AssemblyScript existieren noch keine passenden Bibliotheken. Für Rust wurde die Bibliothek „aes[34]“

[34] https://docs.rs/aes/0.6.0/aes/

verwendet, für JavaScript die Bibliothek „aes-js[35]“. Um das Verhalten WebAssemblys bei der Übertragung genauer beobachten zu können, wurden die Benchmarks jeweils in drei Versionen implementiert. Bei der ersten Version wird der zu kodierende String unverändert und im Ganzen an das Modul gesendet. In der zweiten Version wird der String zu Bytes konvertiert und jeweils einzeln in 16-Bit-Blöcken an die Module gesendet. In der dritten Version wird der String in Bytes konvertiert und als Ganzes an die Module gesendet. Die Form von 16-Bit-Blöcken wurde hier gewählt, da die AES-Funktionen der externen Bibliotheken aller Benchmarks die Daten jeweils immer in diesem Format entgegennehmen. Dies ist auf die Funktionsweise des AES-Verschlüsselungs-Algorithmus zurückzuführen.

Für die Generierung des Workloads wurde in diesem Benchmark wieder die „random-words“-Bibliothek verwendet. Der für die AES-Verschlüsselung verwendete Schlüssel ist ebenfalls eine zufällige Folge von Zeichen.

5.3.7 Hash

Für die Hash-Suite wurden die Algorithmen SHA-256 und SHA-512 in allen Sprachen implementiert. In JavaScript wurde dafür die externe Bibliothek „crypto-js[36]“ verwendet für Rust die Crate „sha2[37]“ und für AssemblyScript die Bibliothek „as-hmac-sha2[38]“. Die Algorithmen nehmen jeweils einen String entgegen und geben diesen in kodierter Form ebenfalls als String zurück.

Als Workload wurde auch hier wieder die Bibliothek „random-words“ verwendet.

[35] https://www.npmjs.com/package/aes-js

[36] https://www.npmjs.com/package/crypto-js

[37] https://docs.rs/sha2/0.9.5/sha2/

[38] https://github.com/jedisct1/as-hmac-sha2

5.3.8 RegEx

Für die RegEx-Suite wurden in jeder Sprache jeweils zwei Funktionen implementiert. Eine Funktion nimmt drei Parameter entgegen, einen Text, ein Zeichen oder Zeichenkette, nach der in diesem Text gesucht werden soll und eine weitere Zeichenkette, mit der die im Text gefundenen Zeichenketten ausgetauscht werden sollen. Die zweite Funktion nimmt nur einen Text und eine Zeichenkette entgegen und zählt anschließend die Vorkommen dieser Zeichenkette in dem übergebenen Text.

Auch hier wurde für die Generierung des Workloads „random-words“ verwendet.

5.3.9 Bildbearbeitung

Die Bildbearbeitung-Suite konnte in allen Sprachen implementiert werden. Da für die Sprache Rust bereits eine umfangreichere Unterstützung bzgl. der Kommunikation mit JavaScript angeboten wird, wurden die Algorithmen hier auf unterschiedliche Weise implementiert. Für Rust wurde die Bibliothek „photon[39]“ verwendet. Photon nutzt unter anderem Bibliotheken wie „web-sys“ wodurch hier direkt mit den in JavaScript implementierten Canvas-Objekten kommuniziert werden kann. Photon verfügt über einen sehr performanten Algorithmus zum Manipulieren der Bilddaten. AssemblyScript hingegen bietet keine dieser Bibliotheken an. Um hier eine performante Lösung realisieren zu können, mussten die Bilddaten des Canvas-Objektes zuvor zu Bytes konvertiert werden, damit diese anschließend direkt im linearen Speicher des Wasm-Moduls platziert werden konnten. Anschließend konnten diese Daten mit einem selbst implementierten „Convolution-Filter“ manipuliert werden. Der in JavaScript implementierte Algorithmus wurde ebenfalls selbst implementiert und ähnelt der Struktur des AssemblyScript-Algorithmus.

Also Workload wurde hier ein 24-Bit-JPG verwendet.

[39] https://github.com/silvia-odwyer/photon

5.3.10 DOM-Manipulation

Die DOM-Manipulation-Suite wurde in Rust und in JavaScript implementiert. Zur Realisierung wurde die für JavaScript übliche Web API verwendet, die zu Erstellung von Webinhalten und Anwendungen einschließlich DOM-Manipulation verwendet werden kann.

Für Rust bietet die Crate „web_sys[40]“ ein großen Teil der Web API an. So kann ähnlich wie in JavaScript der DOM-Tree manipuliert werden. Im folgenden Code-Ausschnitt ist z.B. in Zeile 8 erkennbar, wie aus dem aus JavaScript bekannten *document*-Objekt über die im DOM vergebene ID ein HTML-Element angesprochen werden kann.

```
1  pub fn run(input: i32) -> Result<(), JsValue> {
2      let window = web_sys::window().expect("no global `window` exists");
3      let document = window.document().expect("should have a document on window");
4      let container = document
5          .get_element_by_id("app")
6          .expect("should have #loading on the page");
7
8      document
9          .get_element_by_id("app")
10         .expect("should have #loading on the page")
11         .dyn_ref::<HtmlElement>()
12         .expect("#loading should be an `HtmlElement`")
13         .style()
14         .set_property("display", "block")?;
15
16     for i in 0..input {
17         let p = document.create_element("p")?;
18         p.set_id(&format!("p{}", i));
19         let h1 = document.create_element("h1")?;
20         h1.set_id(&format!("h1{}", i));
21
22     ...
```

Listing 14 - Ausschnitt aus dem DOM-Manipulations-Benchmarks in Rust

In den Benchmarks wird jeweils ein HTML-Element in der Anwendung angesprochen und diverse Kindknoten mit Paragrafen und Headlines hinzugefügt. Anschließend werden diese mehrmals über die vergebene ID angesprochen und deren Inhalt verändert und diese am Ende eines jeden Benchmarks wieder aus dem DOM-Tree entfernt.

AssemblyScript bietet leider noch keine passende API zur Manipulation des DOM-Trees.

[40] https://rustwasm.github.io/wasm-bindgen/api/web_sys/

5.3.11 Transfer

Um noch besser evaluieren zu können, wie sich WebAssembly bei der Übertragung verschiedenen Datentypen verhält, wurden in dieser Benchmark-Suite mehrere Benchmarks implementiert, in denen lediglich Daten an die Module gesendet werden. Die implementierten Algorithmen tun nichts weiter, als die Daten zu empfangen und in derselben Form wieder zurückzusenden. Gesendet werden ein 5000 Wörter-String, ein 5000 Nummern-Array, eine einzelne Nummer, ein einzelnes Wort, ein einzelnes Zeichen (Char) und ein einelementiges Array.

5.4 Validierung

Um ausschließen zu können, dass der Benchmark-Prozess nicht durch fehlerhafte Algorithmen verfälscht wird, wurde eine Validierung dieser Algorithmen durch Unit-Tests realisiert. Diese Tests wurden direkt in der Benchmark-Anwendung mithilfe des „Jasmine-Test-Frameworks[41]“ implementiert. Sie durchlaufen jede Benchmark-Suite und laden dabei die entsprechenden Wasm-Module. Jeder Algorithmus innerhalb einer jeden Suite wird anschließend durch Ausführung validiert, indem die Rückgabewerte mit zuvor definierten Werten abgeglichen werden.

[41] https://jasmine.github.io/

5.5 Erfüllung der Anforderungen

Zuletzt wird in diesem Kapitel erläutert, ob die im Kapitel „Anforderungsanalyse“ definierten Anforderungen mit der in dieser Arbeit entwickelten Anwendung erfüllt wurden.

Bis auf einige kleine Schwierigkeiten, konnten fast alle Anforderungen vollständig erfüllt werden. Die Anwendung ist auf allen gängigen Browsern ausführbar bis auf die Probleme, die AssemblyScript bei der Speicherverwaltung im mobilen Chrome-Browser hat. In der Anwendung können alle gewünschten Benchmarks einzeln oder gesammelt ausgeführt werden. Außerdem können Konfigurationen, wie Art des Workloads oder Wiederholungen, an den Benchmark-Suites vorgenommen werden. Die Anforderungen FA-1 bis FA-3 sind somit erfüllt. Für die Anwendung wurden insgesamt 16 Benchmarks (ohne Transfer) implementiert. Dabei wurde darauf geachtet, dass mit den Benchmarks möglichst alle Aspekte der Technologien beleuchtet werden konnten. Einzige Ausnahme dabei war, dass AssemblyScript noch nicht für jeden Anwendungsfall eine Unterstützung durch Bibliotheken anbietet und daher nicht jeder Benchmark mit dieser Sprache realisiert werden konnte. Die realisierbaren Benchmarks haben allerdings einen ausreichenden Einblick in die Technologie geboten. Deshalb kann die Anforderung FA-4 ebenfalls als erfüllt angesehen werden kann. FA-5 und FA-6 wurden ebenfalls erfüllt. Nach Durchlaufen des Benchmark-Prozesses werden alle Ergebnisse in Form von Liniendiagrammen angezeigt. Zusätzlich werden zu jeder Technologie die mittlere Ausführungsdauer sowie die StartUp-Leistung angegeben.

NFA-1 wurde erfüllt. Alle Algorithmen der Benchmarks wurden, sofern es möglich war, in einer ähnlichen bis gleichen Struktur realisiert. Bei den Benchmarks, bei denen eine gleiche Struktur nicht möglich war oder es sinnvoll war, eine bereits vorhandene performante Lösung in Form von Bibliotheken zu verwenden, wurden die entsprechenden Bibliotheken verwendet.

NFA-2 wurde erfüllt. Eine übersichtliche Benutzungsoberfläche wurde realisiert und die Bedienbarkeit dabei auf ein nötiges und angemessenes Minimum reduziert.

NFA-3 wurde erfüllt. Falls Fehler in Benchmarks auftauchen, so wird nur der entsprechende Benchmark nicht durchgeführt. Nach Durchlaufen des Benchmark-Prozesses, wird der Fehler an der Stelle ausgegeben an der eigentlich hätten die Messwerte stehen sollen.

NFA-4 wurde erfüllt. Durch die Strukturierung der Anwendung mit wiederverwendbaren Komponenten und abstrakten Klassen sowie der Nutzung diverser Skripte zum Kompilieren neuer Wasm-Module, kann die Anwendung leicht erweitert werden.

6 Evaluierung

In diesem Kapitel wird die hier entwickelte Anwendung evaluiert werden. Dazu wird zuerst die Testumgebung beschrieben, in denen die Messungen durchgeführt wurden. Außerdem wird nochmals kurz erläutert, wie der Ablauf eines Benchmark-Prozesses aussieht und mit welchen Workloads die Algorithmen arbeiten. Anschließend folgt die Auswertung. Dabei werden die aussagekräftigsten Ergebnisse näher beleuchtet und Schlussfolgerungen über den Hintergrund dieser Ergebnisse getroffen. Nach der Auswertung der Ergebnisse folgt noch ein Einblick in die Handhabung der Technologien. Die Art der Implementierung sowie die Kompatibilität unterscheidet sich leicht und fließt daher ebenfalls in die Bewertung mit ein. Zum Ende des Kapitel wird ein abschließendes Urteil gegeben, ob sich der Einsatz von WebAssembly lohnt.

6.1 Testumgebung

Wie in den Anforderungen definiert, werden die Testläufe auf den verbreitetsten Browsern durchgeführt. Für die Testläufe auf Chrome, Firefox und Edge wurde ein Laptop mit dem Betriebssystem Windows 10 verwendet, für Safari wurde ein MacBook Pro und für die Mobilversion von Chrome ein Galaxy S9 Android Phone verwendet. Die genauen technischen Daten werden in der folgenden Tabelle (Tabelle 2) abgebildet.

Tabelle 2 - Arbeitsumgebungen

Gerät	***Bezeichnung***	***Betriebssystem***	***Technische Daten***
Laptop	Dell Latitude 5590	Windows 10 Enterprise	i7-8650U-32GBRAM
Laptop	MacBook Pro	Big Sur	i7- -16GB RAM
Smartphone	SM-G960F	Android 10	Samsung Exynos 9810 - 4GB RAM

Des Weiteren wurden die folgenden Browserversionen verwendet:

Tabelle 3 - Eingesetzte Browser und Versionierung

Browser	***Version***
Chrome	91.0.4472.114 (64-Bit)
Firefox	89.0.1 (64-Bit)
Edge	91.0.864.54 (64-Bit)
Chrome (mobile)	91.0.4472.101
Safari	14.0.3

Um eine störungsfreie Umgebung zu gewährleisten, wurden alle Prozesse, die die Messungen beeinflussen könnten, geschlossen. Lediglich die Browser mit der Benchmark-App wurden zum Zeitpunkt der Testläufe ausgeführt.

Der Ablauf eines jeden Benchmarks geschieht wie folgt:

1. Laden der Module
2. Generieren des Workloads
3. Wiederholtes Ausführen der Benchmark-Algorithmen

Bei diesem Ablauf ist es wichtig anzumerken, dass der Workload bei jeder Ausführung der zumessenden Funktion erneut als Parameter übergeben wird. Die Daten werden also jedes Mal erneut in den linearen Speicher der Wasm-Module geladen und die Ergebnisse auch wieder von dort gelesen. Somit fließt das Kopieren dieser Daten in jede Messung mit ein.

Bis auf den Bildbearbeiten-Benchmark wurden alle Benchmarks bei jeder Ausführung 120-mal hintereinander ausgeführt. Der Bildbearbeiten-Benchmark ist rechenintensiver als die anderen und wurde daher nur 10-mal wiederholt.

Wie bereits beschrieben, wurden die Workloads der Benchmarks zufällig generiert. Die folgende Tabelle (Tabelle 4) zeigt nochmals alle ausgeführten Benchmark-Suites auf sowie die Art der Workloads.

Tabelle 4 - Alle Suites, eingesetzte Workloads und Anzahl der Wiederholungen

Benchmark-Suite	***Workload***	***Wiederholungen***
Fibonacci	1 Nummer	120
Hanoi	1 Nummer	120
Factorial	1 Nummer	120
Sorting	Array mit 1000 Nummern	120
Regex	Text mit 3000 Wörtern (19,8 kB String)	120
Base64 - Text kodieren	Text mit 5000 Wörtern (33 kB String)	120
Base64 - Zahlen kodieren	Array mit 5000 Nummern	120
Crypto (AES)	Text mit 2500 Wörtern (16,6 kB String)	120
Hash (SHA)	Text mit 18000 Wörtern (119 kB String)	120
DOM-Manipulation	Kein Workload – lediglich manipulierende Funktionen	120
Bildbearbeitung	JPG-Datei (860 kB)	10

6.2 Auswertung

Zugunsten der Übersichtlichkeit, werden nicht alle Messergebnisse im Einzelnen aufgezeigt und bewertet. Lediglich die aussagekräftigsten Ergebnisse werden thematisiert. Eine Auflistung aller Messergebnisse ist im Anhang „Messergbnisse“ zu finden. Es ist anzumerken, dass die in Firefox gemessenen Ergebnisse des Öffteren bei 0 liegen, da die Funktion *#performance.now()* dort nur auf 1ms genau misst und nicht wie die anderen Browser mit einer Genaugikeit von 5µs.

Die gemessenen Ergebnisse lassen WebAssembly in dieser Auswertung als klaren Sieger hervorgehen. Die durch die Wasm-Module bereitgestellten Algorithmen konnten in 12 von 16 Benchmarks bessere Ergebnisse erzielen, teilweise sogar mit berträchtlichen Vorsprung gegenüber JavaScript.

In den Auswertungen ist erkennbar, dass WebAssembly vorallem in den Aufgaben am besten abschneidet, in denen aufwendige Berechnungen durchgeführt werden, ohne das dabei eine große Kommunikation zwischen dem Wasm-Modul und JavaScript stattfinden muss. Das wird durch die Benchmark-Suites „Hanoi“, „Faktoriell“ und „Fibonacci“ ersichtlich. Die Benchmarks in diesen Suites bestehen teilweise aus rekursiven Algorithmen mit hoher Komplexität. Die Kommunkiation besteht dabei jeweils aus nur zwei Zahlenwerten. Die Ergebnisse dieser Suite entsprechen hier den Erwartungen. Denn schon in der offiziellen Dokumentation [37] der Technologien wird eine JavaScript-WebAssembly-API als optimal beschrieben, wenn die Kommunikation dabei auf ein Minimum reduziert wird. So schneidet WebAssembly bei den Benchmarks der Sorting-Suite weitaus schlechter ab als in den zuvor geanannten Suites.

Zum Vergleich stellt die folgende Abbildung (**Fehler! Verweisquelle konnte nicht gefunden werden.**) oben die Ergebnisse des rekursiven Fibonacci-Algorithmus dar und unten die Ergebnisse des Bubblesort-Algorithmus.

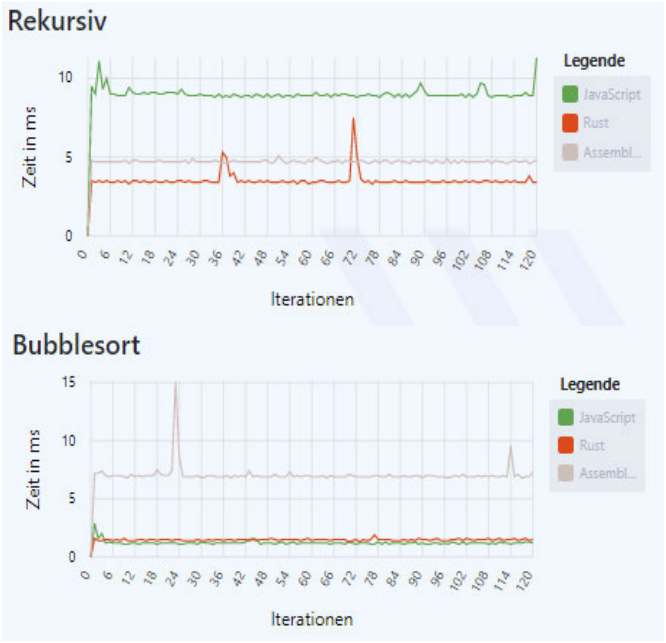

Figure 20 - Vergleich der Messerergebnisse. Fibonacci oben, Bubblesort unten

Die von JavaScript bereitgestellte Sortierfunktion *#sort()*[42] ist überaus performant, sodass Webassembly hier im Schnitt fast 6-mal mehr Zeit benötigt als JavaScript. Es zeigt sich hier also bereits, dass der Aufruf einer Funktion mit geringer Komplexität kombiniert mit dem Senden vieler Daten, ein eher negatives Ergebnis für WebAssembly erzielt. So ist es also sinnvoller, dem Wasm-Modul einen größeren Satz Daten bereitzustellen, auf dem dann anschließend komplexere Operationen ausgeführt werden. Optimal ist dabei, wenn die Ergebnisse dieser komplexen Operationen in minimalistischer Form an die Anwendung zurückgesendet werden.

Ebenfalls zu erwarten war, dass WebAssembly bei den Benchmarks schlechter abschneidet bei denen Operationen auf Strings ausgeführt werden. Wie schon beschrieben, ist WebAssembly eine maschinennahe Sprache. Aus diesem Grund sind hier nur vier Datentypen verfügbar. Soll

[42] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/sort

also mit Strings kommuniziert werden, müssen diese Strings zuvor in kompatible Datentypen umgewandelt werden und wieder zurück, diese Prozesse können viel Zeit in Anspruch nehmen. Die Base64-Benchmarks machen dies gut deutlich. So ist auf dem Bild (Figure 21) gut zu erkennen, wie viel besser WebAssembly bei dem Benchmark abschneidet, bei denen ausschließlich nummerische Werte versendet werden. Während das Kodieren des Textes hier fast 3-mal länger dauert als bei JavaScript, geht das Kodieren der nummerischen Werte mehr als 10-mal schneller.

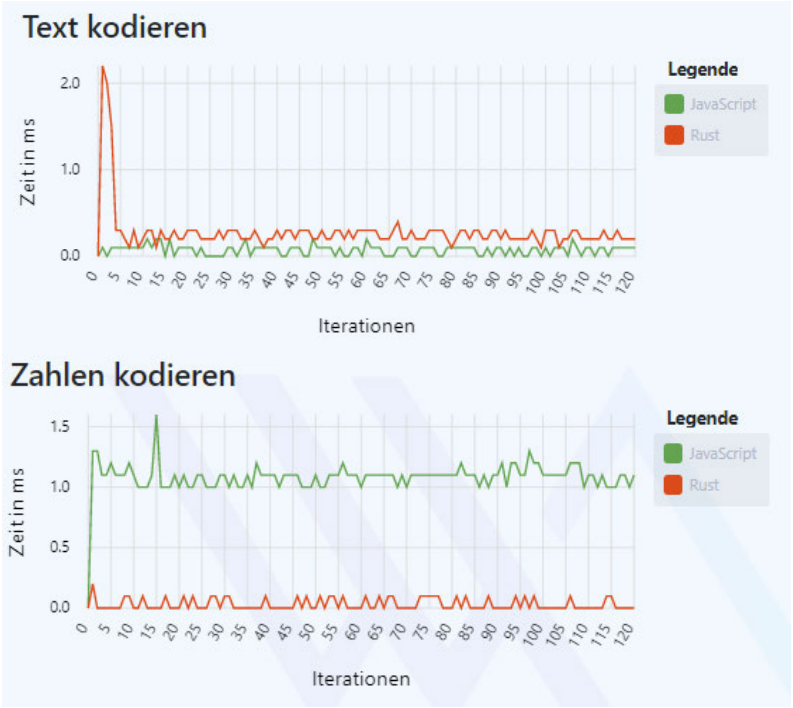

Figure 21 - Ergebnisse der Base64-Suite

Die in der Crypto-Suite erzielten Ergebnisse sind ein gutes Beispiel dafür, dass WebAssembly bei komplexeren Operation auf großen Daten bessere Ergebnisse erzielen kann. Die Datenmenge besteht hier zwar, wie bei der Base64-Suite auch, aus einem großen String, allerdings ist die Ausführung des AES-Algorithmus in Rust so performant, dass sicher der Einsatz eines Wasm-Moduls dennoch lohnt. Die folgende Abbildung (Figure 22) zeigt die Ergebnisse des AES-128-Benchmarks, das in Rust geschriebene Modul ist hier im Schnitt 3-mal schneller.

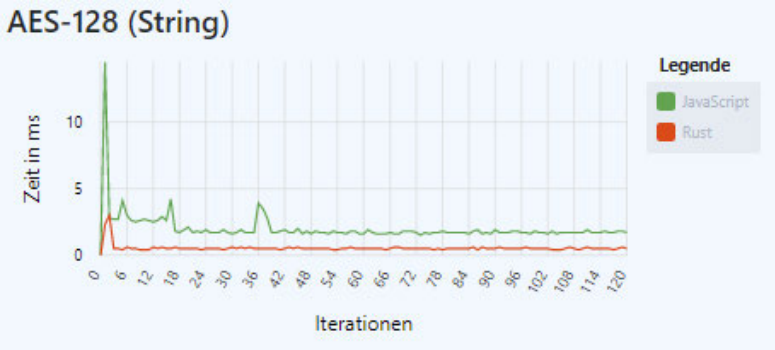

Figure 22 - Ergebnisse des AES-128-Benchmarks mit String-Parameter

Aufgrund der verschiedenen Kommunikationsarten, die in dieser Suite implementiert wurden, lässt sich hier eine interessante Beobachtung machen. Der zu kodierende String wird auf drei Arten an das Modul übergeben, einmal unverändert als ganzer String, einmal konvertiert und aufgeteilt in 16-Bit-Blöcke und einmal konvertiert in Bytes als ein großer Block. Die Auswertung (Figure 23) zeigt, dass es wesentlich performanter ist, den kompletten Datensatz aufeinmal an das Modul zu übergeben, als mehrere kleine Datensätze in mehreren Funktionsaufrufen zu verarbeiten. Die Verarbeitung der kleinen 16-Bit-Datensätze dauert im

Schnitt beinahe 3-mal länger, als das Versenden eines großen Datensatzes. Beide dieser Varianten sind sogar langsamer als der in JavaScript implementierte Algorithmus.

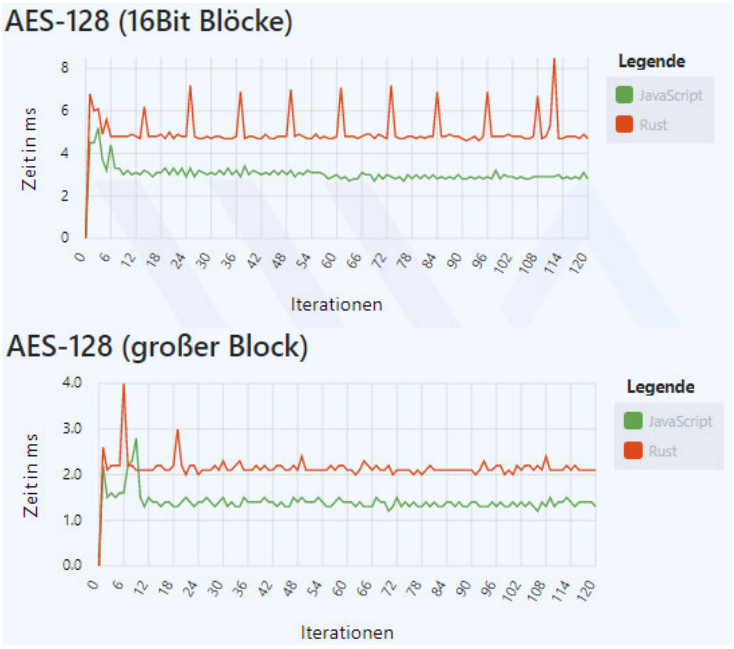

Figure 23 - Ergebnisse des AES-128-Benchmarks mit 16-Bit-Blöcke (oben) und mit einem großen Byte-Block (unten)

Bemerkbar macht sich der Performanzunterschied zwischen WebAssembly und JavaScript vor allem auch in der Hash-Suite. Während beim SHA-256-Benchmark noch relätiv ähnliche Werte zwischen den Technologien zu beobachten waren, hebt sich WebAssembly beim SHA-512-Benchmark erheblich von JavaScript ab. Die Ausführungszeit der Benchmarks durch die Wasm-Module ist für beide Benchmarks nahezu identisch. JavaScript hingegen benötigt für den SHA-512-Benchmark mehr als 3-mal mehr Zeit wie für den SHA-256- Benchmark und benötigt damit mehr als 15-mal länger als das Wasm-Modul.

Die DOM-Manipulation-Suite hat ergeben, dass sich WebAssembly nicht besonders gut für diese Aufgabe eignet. Wie im Implementationskapitel erwähnt, existieren für Rust bereits Bibliotheken[43] mit der die Web-API angesprochen werden kann. Diese Oprationen dauern

[43] https://rustwasm.github.io/wasm-bindgen/web-sys/index.html

allerdings im Schnitt drei Mal länger, als die in JavaScript ausgeführten Operationen. Aus diesem Benchmark ergibt sich auch der Grund für die Tatsache, warum WebAssembly nicht als JavaScript-Ersatz dienen wird. Zugriffe auf das Host-System durch das Wasm-Modul sind momentan nicht direkt möglich und werden aufgrund der Sicherheitsrichtlinien von WebAssembly niemals möglich sein. Daher sind diese Manipulationen am DOM-Tree sehr teuer, was die Module für diesen Einsatz nahezu unbrauchbar macht.

Mit der Bildbearbeiten-Suite wurde ein Anwendungsfall evaluiert, mit denen die WebAssembly-Entwickler direkt auf ihrer Website werben. Tatsächlich ist WebAssembly hier bis zu 4-mal schneller als JavaScript. Das Rust-Modul ist 3,96-mal schneller und das AssemblyScript-Modul immerhin noch 1,55-mal schneller. Hier ist anzumerken, dass die beiden Algorithmen einige Unterschiede aufweisen. So wurde für Rust die Photon-Bibliothek[44] verwendet. Dabei handelt es sich um eine Bildbearbeitungs-Software mit sehr performanten Algorithmen. Wie schon zuvor bei der DOM-Manipulation bietet Rust hier außerdem wieder Bibliotheken zum direkten Kommunizieren mit den JavaScript-Objekten an. AssemblyScript bietet hier noch keinerlei Unterstützung. Hier wurde der Algorithmus selber implementiert. Die Kommunikation wurde dabei vorallem dadaruch erschwert, dass die JavaScript-Objekte nicht direkt angesprochen werden können. Die Daten müssen zuvor in Bytes umgerechnet und anschließend direkt in den linearen Speicher importiert werden, damit diese im Wasm-Modul verarbeitet werden können.

Im folgenden Diagramm (Figure 24) kann man außerdem erkennen, dass die Performnance von JavaScript durch die JIT-Komplierung begünstigt wurde. Der JIT-Kompiler konnte hier deswegen Optimierungen vornehmen, da für diesen Benchmark nur ein Bild als Workload

[44] https://github.com/silvia-odwyer/photon

verwendet wurde. Es ist zu erwarten, dass WebAssembly bei wechelndem Workload, also mehreren Bildern, nochmals einen deutlichen Performanzvorsprung erreichen würde.

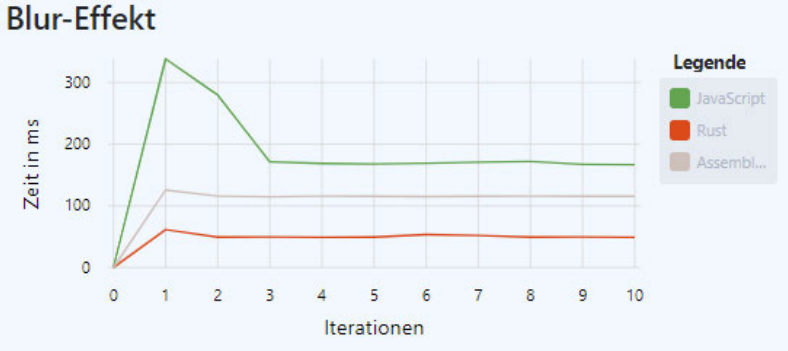

Figure 24 - Ergebnisse des Bildbearbeitungs-Benchmarks

6.2.1 Rust vs. AssemblyScript

Vergleicht man die beiden Sprachen, die in dieser Arbeit zum Generieren der Wasm-Module verwendet wurden, dann ist Rust die bessere Wahl. AssemblyScript hat einige Nachteile mit sich gebracht. Einer davon ergab sich aus den Schwierigkeiten, die bei der Umsetzung der Benchmarks entstanden sind. AssemblyScript ist noch eine junge Sprache, aus diesem Grund existieren bisher nur wenige Bibliotheken. Die Algorithmen, die für den AES- oder Base64-Benchmark benötigt worden wären, sind nicht trivial und konnten daher im Rahmen dieser Arbeit nicht implementiert werden. Ein weiteres Problem hat sich bei der Kompabilität mit anderen Plattformen gezeigt. So mussten die in AssemblyScript geschriebenen Wasm-Module z.B. für den mobilen Chrome-Browser auf Android teilweise deaktiviert werden, da diese Fehler in der Speicherzuweisung erzeugt haben. Die Auswertung hat des Weiteren ergeben, dass die AssemblyScript-Module mit Ausnahme des Regex-Ersetzen-Benchmarks mindestens doppelt soviel Zeit für die Ausührung benötigt haben, wie die Rust-Module. Die AssemblyScript-Module waren in den meisten Fällen zwar trotzdem noch schneller als die

JavaScript-Algorithmen, wenn es allerdings darum geht, eine maximale Performanz zu erzielen, sollte für die Entwicklung Rust gewählt werden.

Der Grund dafür, dass die AssemblyScript-Module langsamer sind, ist die automatische Speicherverwaltung, die dort zum Einsatz kommt. Während sich der Entwickler in Rust selbst um die Bereinigung des Speichers kümmern muss, ist in AssemblyScript der Garbage Collector dafür verwantwortlich. Der Garbage Collector muss paralell zum Programm ausgeführt werden, was zu Leistungseinbußen führt. WebAssembly selbst unterstützt das Ausführen eines Garbage Collectors nicht, weswegen die AssemblyScript-Entwickler eine eigene Lösung implementieren mussten. Es ist allerdings in Planung das WebAssembly in Zukunft eine eigene Garbage Collection implementieren wird. Dies wird vermutlich bewirken, dass AssemblyScript deutlich performanter werden wird.

6.2.2 Performance-Unterschiede der Plattformen

Die Auswertung hat ergeben, dass es bei der Ausführung der Benchmarks auf den verschiedenen Browsern keine nennenswerten Unterscheide in der Performanz gibt. Alle Benchmarks auf allen Browsern konnten ungefähr denselben Leistungsunterschied zu JavaScript aufweisen. Auch das Initialisieren der Module hat auf allen Plattformen ungefähr dieselbe Zeit in Anspruch genommen. Allein der mobile Chrome-Browser auf Android ist durch die Probleme bei der Speicherzuweisung mit AssemblyScript aufgefallen. Wie von den Entwicklern angemerkt [20], ist allerdings auch hier zu erwarten, dass dieses Problem in naher Zukunft gelöst sein wird.

6.3 Handhabung

Im Allgemeinen kann der Umgang mit WebAssembly als sehr bequem bewertet werden. Da JavaScript bzw. TypeScript schon von Haus aus das Einbinden der Wasm-Module unterstützt, wird einem sowohl beim Import der Module als auch beim Ansprechen der darin befindlichen Funktionen, ein großer Teil der Arbeit abgenommen. Die von den Entwicklern bereitgestellten

Bibliotheken für Rust und AssemblyScript erleichtern die Arbeit zusätzlich. Wie schon beschrieben, wurden dabei bei den Sprachen unterschiedliche Ansätze zur Kommunikation zwischen den Modulen und JavaScript gewählt.

Während der Generierung der Module aus dem in Rust geschriebenen Code, generiert die „wasm-pack"-Toolchain automatisch den sogenannten Glue-Code mit. Dieser Code ist JavaScript geschrieben und übernimmt für den Entwickler u.a. das Konvertieren und Übertragen von Daten an das Modul. Außerdem enthält der Code TypeScript-Typdeklarationen. So können die Wasm-Funktionen auf ihre Typen geprüft werden, sofern TypeScript beim Entwickeln verwendet wird. Außerdem kann die verwendete IDE[45] somit eine Autovervollständigung anbieten. Die folgende Abbildung (Listing 15) zeigt, wie ein in Rust geschriebenes Modul in eine TypeScript geschriebene Anwendung importiert werden.

```
1 import('src/assets/wasm/rust/crypto').then(module => {
2
3     // Ausführen der ersten Funktion
4     module.hash_sha256(workload.input);
5
6     // Ausführen der zweiten Funktion
7     module.hash_sha512(workload.input);
8 });
```

Listing 15 - Import eines in Rust implementierten Wasm-Moduls

Beim Importieren der Module mit der von JavaScript bereitgestellten *#import*-Funktion muss hier lediglich das Verzeichnis angegeben werden, in dem sich das Wasm-Modul und der entsprechende Glue-Code befindet. Die in dem Modul befindlichen Funktionen können anschließend aufgerufen werden, wie normale JavaScript-Funktionen.

In AssemblyScript wird kein Glue-Code generiert, stattdessen kann hier ein von den Entwicklern bereitgestellter Loader[46] verwendet werden, der die Arbeit mit den Modulen bequemer machen soll. Er spiegelt die relevanten Teile der WebAssembly-API wider und bietet darüber

[45] Integrierte Entwicklungsumgebung (z.B. IntelliJ IDEA)

[46] https://www.assemblyscript.org/loader.html

hinaus Funktionen zum Zuweisen und Lesen von Strings, Arrays und Klassen. Der Loader instanziiert das Modul unter Verwendung der WebAssembly-API und fügt dabei zusätzlichen Nutzen hinzu, indem er die Exportnamen auswertet und eine schönere Objektstruktur daraus macht. Wird z.B. eine ganze Klasse mit dem Namen *Foo* aus dem Modul exportiert, steht bei Verwendung des Loaders ein *exports.Foo*-Konstruktor zur Verfügung. Im Gegensatz zur Rust-Lösung können hier allerdings nicht einfach JavaScript-Strings als Parameter an die Funktionen übergeben werden. In diesem Fall müssen spezielle Funktionen des Loaders verwendet werden, die es ermöglichen, den Speicher im Wasm-Modul zu reservieren. Anschließend kann der String, der automatisch in Bytes übersetzt wird auf diesem Speicher platziert werden. Die Parameter, die dann an die im Modul befindliche Funktion übergeben werden, sind Pointer, die auf die entsprechende Adresse im Speicher verweisen. Die folgende Abbildung (Listing 16) stellt diesen Prozess dar.

```
Loader.instantiate(
    fetch('assets/wasm/assemblyscript/hash/optimized.wasm'),
).then(({exports}) => {

    const {
      sha256AsString,
      sha512AsString,
      __newString,
      __getString
    } = exports;

    const runSha256AsString = (value: string) => {
      const valuePointer = __newString(value);
      // @ts-ignore
      const resultPointer = sha256AsString(valuePointer);
      return __getString(resultPointer);
    };

    const runSha512AsString = (value: string) => {
      const valuePointer = __newString(value);
      // @ts-ignore
      const resultPointer = sha512AsString(valuePointer);
      return __getString(resultPointer);
    };

    // Ausführen der ersten Funktion
    runSha256AsString(workload.input);

    // Ausführen der zweiten Funktion
    runSha512AsString(workload.input);
});
```

Listing 16 - Import eines in AssemblyScript implementierten Wasm-Moduls

Wie hier deutlich zu erkennen ist, ist der Implementierungsaufwand um einiges höher als bei den Rust-Modulen. Das liegt daran, dass die Funktionen zum Konvertieren und Übertragen der Daten hier nicht automatisch im Glue-Code ausgeführt werden, sondern aus der Instanz exportiert und manuell ausgeführt werden müssen.

Der Grund für das Weglassen des Glue-Codes in AssemblyScript ist, dass die Entwickler damit einen geringeren Overhead bei den Modulen erwirken wollten. Dafür muss mehr Arbeit in die Einbindung der Module gesteckt werden als bei Rust. Es ist also etwas vom Geschmack des Entwicklers sowie vom Einsatzgebiet abhängig, welche der beiden Ansätze hier als besser zu bewerten ist.

6.4 Abschließendes Urteil

WebAssembly konnte in so gut wie jedem simulierten Anwendungsfall der Benchmark-App überzeugen. Vor allem rechenintensive Operationen konnten durch die Funktionen der Wasm-Module meist um ein Vielfaches schneller erledigt werden als durch die JavaScript-Funktionen. Das war auch der Fall, wenn diese Operationen auf Strings ausgeführt wurden. Zusätzlich kann die Erstellung der Wasm-Module aus der Quell-Sprache als sehr einfach bewertet werden. Das Einbinden der Module war, wie schon beschrieben, ebenfalls überaus leicht zu bewerkstelligen. Durch den Gewinn an Performanz, die einfache Handhabung und der großen Anzahl an Sprachen, aus denen die Module generiert werden können, ist der Einsatz von WebAssembly in beinahe jedem Projekt denkbar. Da die Unterstützung dieser Sprachen außerdem stetig zunimmt, wird es in naher Zukunft kaum noch einen Entwickler geben, der nicht im Stande ist, auf einfache Weise performante Wasm-Module zu entwickeln.

7 Zusammenfassung und Ausblick

7.1 Zusammenfassung

Ziel dieser Arbeit war die Evaluierung der Technologie WebAssembly. Zu diesem Zweck wurde eine Webanwendung mit mehreren Benchmark-Funktionen entwickelt. Diese Benchmark-Funktionen wurden jeweils in JavaScript, Rust und AssemblyScript implementiert, sodass die Ausführungsgeschwindigkeit der verschiedenen Implementationen miteinander verglichen werden konnten.

Die Entwicklung dieser Anwendung mit anschließender Evaluierung konnte erfolgreich durchgeführt werden, damit wurde das Ziel dieser Arbeit erreicht. Diese Arbeit gibt außerdem einen Einblick in die Funktionsweise von WebAssembly und wie die eingesetzten Wasm-Module in bestehende Anwendung integriert werden können. Zusätzlich konnte aufgezeigt werden, dass der Einsatz von WebAssembly schon bei kleinen Anwendungsfällen überaus lohnend ist. Des Weiteren konnten Stärken und Schwächen der Technologie ermittelt werden sowie die Unterschiede zwischen den in Rust und in AssemblyScript entwickelten Modulen.

Vor allem interessant war der Einblick in den Entwicklungsstand von AssemblyScript. Es hat sich gezeigt, wie einfach es ist Wasm-Module zu generieren und dabei eine Sprache zu verwendet die syntaktisch nahezu identisch mit TypeScript ist. Vor allem für Entwickler aus dem Front-End-Bereich ist das von besonderem Interesse.

7.2 Ausblick

Zusammengefasst lässt sich sagen, dass noch viel von WebAssembly zu erwarten ist. Schon jetzt ist es möglich, auf einfache Weise aus vielen Sprachen Wasm-Module zu generieren [18], die häufig um ein vielfaches performanter sind als JavaScript. Mit zukünftigen Entwicklungen, wie SIMD[47] und Threading ist nochmals eine deutliche Verbessung der Performanz zu erwarten. Weitere interessante Technologien sind Wasmtime[48] oder Node.js[49] mit denen es möglich ist, WebAssembly-Module außerhalb des Browsers, also im Grunde als stand-alone auszuführen.

Mit diesen Themen tun sich interessante Ansätze für weiterführende Arbeiten auf. So wäre das Ausführen der Benchmarks mit mehreren Threads vorstellbar. Ein weiterer Ansatz wäre die Implementierung komplexerer Benchmarks in Form von Real-Application-Benchmarks. Wie in den Grundlagen erwähnt, sind nur diese Benchmarks im Stande, die wahre Performanz eines Systems zu ermitteln. Im Rahmen eines solchen Benchmarks, könnte z.B. eine native Anwendung, die in Rust oder einer anderen Sprache geschrieben wurde zu WebAssembly kompiliert werden. Anschließend könnte evaluiert werden, was dabei die Vor- und Nachteile sind. Bei einer solchen Evaluierung könnte vor allem der Performanzunterschied zwischen der nativen Anwendung und der WebAssembly-Anwendung im Fokus stehen.

[47] single instruction multiple data

[48] https://wasmtime.dev/

[49] https://nodejs.org/en/

8 Literaturverzeichnis

[1] mozilla, „developer.mozilla.org,“ 12 06 21. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/DOMHighResTimeStamp. [Zugriff am 30 06 21].

[2] ITWissen, „ITWissen.info,“ 21. [Online]. Available: https://www.itwissen.info/Rechenleistung-computing-power.html. [Zugriff am 04 05 21].

[3] W3C, „World Wide Web Consortium,“ 21. [Online]. Available: https://www.w3.org/2019/12/pressrelease-wasm-rec.html.en. [Zugriff am 09 06 21].

[4] JAX, „JAX,“ 2021. [Online]. Available: https://jax.de/blog/webassembly-fuer-java-eine-revolution/. [Zugriff am 29 06 21].

[5] wiki.selfhtml.org, „wiki.selfhtml.org,“ 21. [Online]. Available: https://wiki.selfhtml.org/wiki/HTML/Tutorials/HTML5/Entstehung_und_Entwicklung#HTML_als_Auszeichnungssprache. [Zugriff am 30 06 21].

[6] J. N. Robbins, Learning Web Design, Canada: O'Reilly, 18.

[7] WHATWG, „HTML Spec,“ 21. [Online]. Available: https://html.spec.whatwg.org/. [Zugriff am 29 06 21].

[8] D. Flanagan, JavaScript : das umfassende Referenzwerk, 6 Hrsg., Köln: O'Reilly, 12.

[9] S. Koch, JavaScript. Einführung, Programmierung und Referenz – inklusive Ajax, 5 Hrsg., Heidelberg: dpunkt, 09, p. 457.

[10] D. Cameron, HTML5, JavaScript und jQuery: Der Crashkurs für Softwareentwickler, 1 Hrsg., Heidelberg: dpunkt, 15.

[11] K. Hoffman, Programming WebAssembly with Rust, 4 Hrsg., Raleigh, North Carolina: The Pragmatic Bookshelf, 19.

[12] G. Gallant, WebAssembly in Action, Shelter Island: Manning, 19.

[13] Clark, „hacks.mozilla.org,“ 17. [Online]. Available: https://hacks.mozilla.org/2017/02/creating-and-working-with-webassembly-modules/. [Zugriff am 03 04 21].

[14] L. Clark, „hacks.mozilla,“ 17. [Online]. Available: https://hacks.mozilla.org/2017/02/a-crash-course-in-just-in-time-jit-compilers/. [Zugriff am 03 04 21].

[15] WebAssembly, „WebAssembly.org,“ 21. [Online]. Available: https://webassembly.org/. [Zugriff am 04 03 21].

[16] Wasm, „webassembly.org/,“ 21. [Online]. Available: https://webassembly.org/. [Zugriff am 30 06 21].

[17] mozilla, „developer.mozilla.org,“ 21. [Online]. Available: https://developer.mozilla.org/en-US/docs/WebAssembly. [Zugriff am 05 03 21].

[18] Wasm, „Wasm Guide,“ 21. [Online]. Available: https://webassembly.org/getting-started/developers-guide/. [Zugriff am 28 06 21].

[19] Vladimir, „blog.sqreen,“ 18. [Online]. Available: https://blog.sqreen.com/webassembly-performance/. [Zugriff am 30 06 21].

[20] AS, „AssemblyScript Docu,“ 21. [Online]. Available: https://www.assemblyscript.org/types.html. [Zugriff am 30 06 21].

[21] Wasm, „Wasm Specs,“ 17. [Online]. Available: https://webassembly.github.io/spec/core/appendix/custom.html. [Zugriff am 04 20 21].

[22] Wasm, „Wasm Tools,“ 19. [Online]. Available: https://github.com/WebAssembly/tool-conventions/blob/master/ProducersSection.md. [Zugriff am 06 05 21].

[23] Mozilla, „research.mozilla.or,“ 21. [Online]. Available: https://research.mozilla.org/rust/. [Zugriff am 28 06 21].

[24] S. K. a. C. Nichols, THE RUST PROGRAMMING LANGUAGE, 1 Hrsg., San Francisco: nostarch, 19.

[25] AssemblyScript, „The AssemblyScript Book,“ 21. [Online]. Available: https://www.assemblyscript.org/basics.html#strictness. [Zugriff am 25 06 21].

[26] „Wikipedia,“ 21. [Online]. Available: https://de.wikipedia.org/wiki/Rechenleistung#cite_note-2. [Zugriff am 05 03 21].

[27] S. Souders, „High performance web sites,“ 08. [Online]. Available: https://dl.acm.org/doi/pdf/10.1145/1466443.1466450. [Zugriff am 05 04 21].

[28] S. Shi, Q. Wang und P. Xu, „Benchmarking State-of-the-Art Deep Learning Software Tools,“ IEEE, 16.

[29] T. Garsiel, Web-Browser, Funktionsweise von Browsern: Hinter den Kulissen moderner Web-Browser, -: html5rocks, 11.

[30] D. B. W. Dai, „Benchmarking Deep Learning Hardware and Frameworks: Qualitative Metrics," Southeast Missouri State University, Missouri, 19.

[31] ISO/IEC, „ISO/IEC 14756:1999(E)," ISO/IEC, 99.

[32] AppleInc, „browserbench," 18. [Online]. Available: https://browserbench.org/JetStream/in-depth.html. [Zugriff am 05 04 21].

[33] A. Vogel, „How fast is WebAssembly?," University of Technology Braunschweig, Braunschweig, 20.

[34] Wasm, „Wasm Docs," 21. [Online]. Available: https://webassembly.org/docs/use-cases/. [Zugriff am 06 28 21].

[35] Angular, „angular.io," 21. [Online]. Available: https://angular.io/guide/what-is-angular. [Zugriff am 30 06 21].

[36] TS, „typescriptlang.org," 21. [Online]. Available: https://www.typescriptlang.org/. [Zugriff am 30 06 21].

[37] Zeno Rocha, „Rust and WebAssembly," 20. [Online]. Available: https://rustwasm.github.io/docs/book/game-of-life/implementing.html. [Zugriff am 28 06 21].

[38] mozilla, „developer.mozilla.org,“ 21. [Online]. Available: https://developer.mozilla.org/de/docs/Web/API/Performance/now. [Zugriff am 30 06 21].

[39] D. G. M. H. J. B. Andreas Haas, „Bringing the Web up to Speed with WebAssembly,“ ACM, Barcelona, 17.

[40] itwissen, „itwissen,“ 14 04 18. [Online]. Available: https://www.itwissen.info/SIMD-single-instruction-multiple-data-SIMD-Rechnerarchitektur.html. [Zugriff am 30 06 21].

I Anhang 1

I.1 Messergebnisse

Fibonacci-Folge (I)	Chrome	**Δ**	Firefox	**Δ**	Edge	**Δ**	Safari	**Δ**	Chrome (m)	**Δ**
JavaScript	0,032		0,028		0,019		0,022		0,039	
Rust	0,003	**10,67**	0,004	**7,00**	0,003	**6,33**	0,009	**2,44**	0,008	**4,88**
AssemblyScript	0,005	**6,40**	0,008	**3,50**	0,007	**2,71**	0,012	**1,83**	0,009	**4,33**

Fibonacci-Folge (R)	Chrome	**Δ**	Firefox	**Δ**	Edge	**Δ**	Safari	**Δ**	Chrome (m)	**Δ**
JavaScript	27,861		35,261		22,585		5,891		42,638	
Rust	10,766	**2,59**	9,084	**3,88**	8,779	**2,57**	3,513	**1,68**	13,019	**3,28**
AssemblyScript	13,87	**2,01**	10,479	**3,36**	12,724	**1,77**	6	**0,98**	20,314	**2,10**

Bubblesort	Chrome	**Δ**	Firefox	**Δ**	Edge	**Δ**	Safari	**Δ**	Chrome (m)	**Δ**
JavaScript	3,021		4,538		3,032		1,076		5,33	
Rust	3,588	**0,84**	3,294	**1,38**	3,739	**0,81**	1,958	**0,55**	6,2	**0,86**
AssemblyScript	17,328	**0,17**	16,748	**0,27**	17,48	**0,17**	10,588	**0,10**	6,99	**0,76**

Interne Sortierf.	Chrome	**Δ**	Firefox	**Δ**	Edge	**Δ**	Safari	**Δ**	Chrome (m)	**Δ**
JavaScript	0,041		0,202		0,025		0,008		0,083	
Rust	0,134	**0,31**	0,076	**2,66**	0,128	**0,20**	0,143	**0,06**	0,199	**0,42**
AssemblyScript	0,226	**0,18**	0,261	**0,77**	0,226	**0,11**	0,151	**0,05**	0,222	**0,37**

Hanoi	Chrome	Δ	Firefox	Δ	Edge	Δ	Safari	Δ	Chrome (m)	Δ
JavaScript	21,916		23,807		21,203		10,664		35,393	
Rust	7,002	**3,13**	2,05	**11,61**	2,003	**10,59**	3,1	**3,44**	8,8	**4,02**
AssemblyScript	10,41	**2,11**	8,034	**2,96**	10,32	**2,05**	4,874	**2,19**	15,703	**2,25**

Regex (Ersetzen)	Chrome	Δ	Firefox	Δ	Edge	Δ	Safari	Δ	Chrome (m)	Δ
JavaScript	0,718		0,336		0,741		0,16		1,2	
Rust	0,76	**0,94**	0,773	**0,43**	0,671	**1,10**	0,471	**0,34**	0,998	**1,20**
AssemblyScript	0,317	**2,26**	0,261	**1,29**	0,315	**2,35**	0,193	**0,83**		

Regex (Zählen)	Chrome	Δ	Firefox	Δ	Edge	Δ	Safari	Δ	Chrome (m)	Δ
JavaScript	0,512		0,37		0,499		0,109		0,8	
Rust	0,327	**1,57**	0,378	**0,98**	0,342	**1,46**	0,235	**0,46**	0,221	
AssemblyScript										

Base64 (String)	Chrome	Δ	Firefox	Δ	Edge	Δ	Safari	Δ	Chrome (m)	Δ
JavaScript	0,191		0,193		0,198		0,025		0,309	
Rust	0,503	**0,38**	0,471	**0,41**	0,469	**0,42**	0,437	**0,06**	0,776	**0,40**
AssemblyScript										

Base64 (NUM)	Chrome	Δ	Firefox	Δ	Edge	Δ	Safari	Δ	Chrome (m)	Δ
JavaScript	3,006		1,311		2,703		0,697		5,712	
Rust	0,052	**57,81**	0,092	**14,25**	0,051	**53,00**	0,059	**11,81**	0,124	**46,06**
AssemblyScript										

AES-128 (String)	Chrome	Δ	Firefox	Δ	Edge	Δ	Safari	Δ	Chrome (m)	Δ
JavaScript	4,617		9,059		4,639		3,832		14,769	
Rust	1,396	**3,31**	1,42	**6,38**	1,487	**3,12**	0,857	**4,47**	2,751	**5,37**
AssemblyScript										

AES-256 (String)	Chrome	**Δ**	Firefox	**Δ**	Edge	**Δ**	Safari	**Δ**	Chrome (m)	**Δ**
JavaScript	5,098		10,765		4,933		4,403		16,603	
Rust	1,628	**3,13**	1,975	**5,45**	1,665	**2,96**	0,992	**4,44**	3,091	**5,37**
AssemblyScript										

SHA-256	Chrome	**Δ**	Firefox	**Δ**	Edge	**Δ**	Safari	**Δ**	Chrome (m)	**Δ**
JavaScript	11,944		15,622		10,84		4,706		13,88	
Rust	1,642	**7,27**	1,765	**8,85**	1,641	**6,61**	1,571	**3,00**	4,3	**3,23**
AssemblyScript	8,148	**1,47**	7,84	**1,99**	8,18	**1,33**	5,134	**0,92**	10,22	**1,36**

SHA-512	Chrome	**Δ**	Firefox	**Δ**	Edge	**Δ**	Safari	**Δ**	Chrome (m)	**Δ**
JavaScript	38,475		29,543		36,76		13,714		44,2	
Rust	1,312	**29,33**	1,479	**19,97**	1,313	**28,00**	0,958	**14,32**	4,2	
AssemblyScript	5,578	**6,90**	5,437	**5,43**	5,516	**6,66**	3,353	**4,09**	8,332	

DOM - Man.	Chrome	**Δ**	Firefox	**Δ**	Edge	**Δ**	Safari	**Δ**	Chrome (m)	**Δ**
JavaScript	0,013		0,008		0,011		0,008		0,024	
Rust	0,06	**0,22**	0,042	**0,19**	0,03	**0,37**	0,017	**0,47**	0,087	**0,28**
AssemblyScript										

Bildbearbeitung	Chrome	**Δ**	Firefox	**Δ**	Edge	**Δ**	Safari	**Δ**	Chrome (m)	**Δ**
JavaScript	458,033		519,778		455,611		188,667		790,189	
Rust	115,711	**3,96**	192	**2,71**	117,833	**3,87**	134,778	**1,40**	312,867	**2,53**
AssemblyScript	296,233	**1,55**	352,111	**1,48**	288,844	**1,58**	208,778	**0,90**	345,86	**2,28**

Factorial (R)	Chrome	**Δ**	Firefox	**Δ**	Edge	**Δ**	Safari	**Δ**	Chrome (m)	**Δ**
JavaScript	0,043		0,025		0,05		0,008		0,069	
Rust	0,003	**14,33**	0,006	**4,17**	0,002	**25,00**	0,008	**1,00**	0,008	**8,63**
AssemblyScript	0,008	**5,38**	0,008	**3,13**	0,004	**12,50**	0,008	**1,00**	0,013	**5,31**

Erklärung zur selbstständigen Bearbeitung einer Abschlussarbeit

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich gemacht.

Ort	Datum	Unterschrift im Original