WebAssembly – ein Jahr danach

Mirko Sertic, Thalia Bücher

WebAssembly wurde Anfang 2017 in Form eines Minimum Viable Product (MVP) in der Version 1.0 veröffentlicht. Dieser Artikel gibt einen kurzen Rückblick auf die Entstehungsgeschichte, zeigt den aktuellen Stand der Implementierung und bringt einen Ausblick auf mögliche Anwendungsfälle sowie zukünftige Erweiterungen.

Wir schreiben das Jahr 2018. Das Internet ist allgegenwärtig. Als Anwender und Entwickler steht uns eine Vielzahl unterschiedlichster Technologien für die Entwicklung moderner Web-Anwendungen zur Verfügung. Es gibt den HTML5-Standard. Wir haben CSS. Wir nutzen IndexedDB oder auch den LocalStore im Browser. Mit diesen Technologien lassen sich schnell und effizient Online- oder auch Offline-Anwendungen bauen und verteilen. Seit Anfang 2017 gibt es in diesem Universum neu WebAssembly. Um WebAssembly und dessen Bedeutung genauer zu verstehen, müssen wir jedoch eine kleine Zeitreise in die Vergangenheit unternehmen.

Wie alles begann

Im Jahr 1995 wurde ein wichtiger Meilenstein für das gesamte Internet gelegt. Als Ergebnis des Projekts „Mocha" stand zusammen mit dem Netscape Navigator Version 2.02 eine Skript-Sprache für dynamische HTML-Seiten der breiten Öffentlichkeit zur Verfügung. Diese Skript-Sprache hatte den Namen „Netscape LiveWire" und wurde später in „JavaScript" umbenannt. Damit war ein Fundament geschaffen, auf dem noch heute unsere Anwendungen aufbauen.

Die Adaption von JavaScript verlief anfangs eher zögerlich. Hintergrund war der laufende Prozess der Sprach-Normierung und -Weiterentwicklung gepaart mit den Browser-Kriegen der damaligen Zeit. Es gab noch nicht viele Standards beziehungsweise die Standards wurden sehr hart erkämpft.

Im Jahr 2005 erwähnte James Garret in seinem Aufsatz das erste Mal das Wort „AJAX". Nun war es möglich, die dynamischen Inhalte im Browser auch mit Backend-Logik zu koppeln, um noch interaktivere Anwendungen zu bauen. JavaScript wurde immer beliebter.

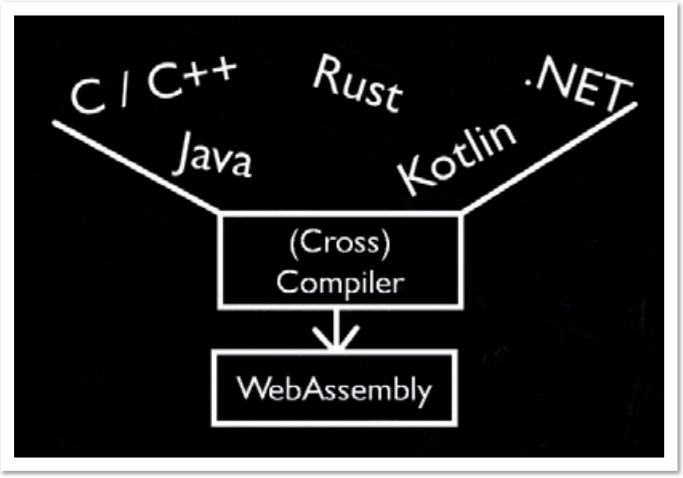

Abbildung 1: WebAssembly ist ein Compile-Target

Neue Horizonte und Möglichkeiten

Wir spulen die Zeit nun etwas vor bis in das Jahr 2010. Das Internet ist überall; auf dem Desktop; auf unseren Smartphones; auf embedded Devices; in der Gebäude-Automatisierung. JavaScript ist überall. Diese standardisierte Plattform ist eine ideale Grundlage für weitere Anwendungsfälle. In diesem Jahr betritt ein sehr interessantes Projekt die Bühne: Emscripten, im Wesentlichen ein C- beziehungsweise C++-Compiler. Dieser erzeugt jedoch keinen Binärcode in Form von X86-Instruktionen, sondern er übersetzt das Programm nach JavaScript. Somit sind völlig neue Deployment-Szenarien für bestehende C++-Anwendungen möglich.

Der Early-Adopter von Emscripten war die Entertainment-Branche, darunter große Spiele-Studios wie Epic. Durch den Einsatz von Emscripten war es möglich, die Unreal Game Engine sowie bestehende Desktop-Spiele mit minimalem Aufwand auch im Browser laufen zu lassen – aus wirtschaftlicher Sicht ein gigantischer Sprung. Die Unreal Engine komplett in JavaScript neu zu schreiben, hätte sicherlich ein großes Team über mehrere Monate bis Jahre ausgelastet. Emscripten als Compiler hat das deutlich optimiert. Ergebnis war die Unreal-Cathedral-Demo, die im Browser lief. Als Laufzeitgeschwindigkeit wurde die halbe Framerate im Vergleich zu der Desktop-Variante erreicht. Das Spiel war spielbar, jedoch gab es noch deutlich Luft nach oben.

Es geht noch schneller

Nach intensiver Zusammenarbeit zwischen Mozilla, Epic und dem Emscripten-Team wurde im Jahr 2013 „asm.js" präsentiert, eine Untermenge von JavaScript, die auf Laufzeit-Effizienz ausgelegt ist. Durch die Annotation von Ausdrücken ist es möglich, der JavaScript-Runtime Hinweise zu geben, um den Code effizienter auszuführen. Diese Typisierung von JavaScript ermöglicht es, Integer- und Fließkomma-Datentypen genauer zu unterscheiden und somit gerade die Integer-Berechnung deutlich zu verbessern. Diese Unterscheidung ist in JavaScript nicht möglich, da es hier nur Fließkomma-Zahlen gibt. Emscripten wurde mit „asm.js" als Compile-Target erweitert. Ergebnis war die Unreal Engine mit der Cathedral-Demo und annähernd nativer Geschwindigkeit im Browser.

Zu diesem Zeitpunkt gab es Browser mit sehr guten JavaScript-Just-in-time-Compilern. Durch „asm.js" wurden noch zusätzliche Hinweise für die Optimierung des Laufzeit-Verhaltens gegeben. Insgesamt war das Ergebnis im Sinne der Laufzeit sehr ansehnlich. Jedoch hatte das Ganze auch seine Schattenseite. Die Code-Basen wurden immer größer, der Quellcode immer komplexer.

Zusammen mit der Größe und der Komplexität stieg natürlich auch die Download-Zeit für Anwendungen deutlich an; auch die Parsing-Zeit für JavaScript im Browser. Als Ergebnis war die Time-to-Interactive bei sehr großen Anwendungen für spontane Internet-User nur noch bedingt akzeptabel. Die Suche nach einem Werkzeug begann, das sowohl das Laufzeit-Verhalten als auch die Download- und Parsing-Zeiten optimiert.

Die Entstehung von WebAssembly

Im Jahr 2015 wurde die WebAssembly Working Group als Reaktion auf die genannte Fragestellung gegründet. Das Ergebnis ihrer Arbeit war ein Format, das unterschiedlichste Anforderungen unter einen Hut bringt. WebAssembly ist Bytecode für das Web. Es handelt sich hier um ein binäres Format für transportable Programme, das auf Download-Größe sowie Ausführungsgeschwindigkeit optimiert ist.

Interessant an diesem Format ist, dass es in Host-Umgebungen integrierbar ist. Als primäre Host-Umgebung steht natürlich der Browser im Vordergrund. Es ist jedoch auch denkbar, WebAssembly im Backend zu nutzen. Dazu ist nur eine Host-Umgebung erforderlich, die den WebAssembly-Standard implementiert. Die Browser-Host-Umgebung setzt auf der bestehenden JavaScript-Runtime im Browser auf. Durch diese Integration wird die bestehende Just-in-time-Compiler-Mechanik im Browser wiederverwendet. Für die Browser-Hersteller ist es so wesentlich einfacher, WebAssembly zu integrieren.

WebAssembly 1.0 MVP

Anfang 2017 war es dann soweit. WebAssembly wurde in Form eines Minimum Viable Product in Version 1.0 veröffentlicht. Bemerkenswert war, dass alle Browser-Hersteller im Abstand von nur wenigen Tagen den WebAssembly-Support ausrollten. Was ist nun Teil des MVP? Schon im initialen Design wurden bewusst Stolpersteine ausgeklammert, um die erste Version des Produktes schlank zu halten. Allerdings wurde gleichzeitig dafür gesorgt, dass diese Abgrenzung keine größeren Nachteile verursacht.

Beispielhaft beinhaltet die WebAssembly-Host-Umgebung eine Sandbox, in der das Programm abgeschottet läuft. Sie ermöglicht den Zugriff auf einen linearen Speicherbereich. Dieser ist jedoch „nicht managed". Der WebAssembly-Autor muss also das Speichermanagement und gegebenenfalls auch einen Garbage-Collector selbst implementieren. Dies ist jedoch kein größerer Nachteil, da WebAssembly ein Modul-System für dynamisches Linken beinhaltet. Damit lässt sich ein bestehender Memory-Manager einfach in das WebAssembly linken, um darauf aufzubauen.

Ebenfalls wurden im MVP bewusst kein DOM-Zugriff oder die „opaque"-Data-Types implementiert. Es ist aus der WebAssembly-Sandbox nicht direkt möglich, auf das DOM oder Browser-APIs zuzugreifen. Auch dies ist kein größerer Nachteil, da über das Modul-System der Zugriff durch Wrapper-Types emuliert werden kann. Diese verursachen zwar gewisse Laufzeit-Nachteile, was jedoch im Vergleich mit der reduzierten Komplexität zu verkraften ist.

Im MVP wurde bewusst auf Unterstützung für Threading und Thread-Synchronisierung verzichtet. Dies ist einleuchtend, da die WebAssembly-Host-Umgebung auf der JavaScript-Integration im Browser aufbaut und es in JavaScript keine Threads und keine Thread-Synchronisierung gibt.

Dieser reduzierte Funktionsumfang ist jedoch kein Widerspruch. Durch die bewusste Ausklammerung dieser Funktionalitäten und gleichzeitige Unterstützung eines Modul-Systems ist eine konsistente, nutzbare und vor allem erweiterbare WebAssembly-Version 1.0 entstanden.

Kernkonzepte und Bootstrap

Genug der Theorie. Wie funktioniert nun WebAssembly? Es kommt in zwei verschiedenen Ausprägungen vor, der textuellen Repräsentation und der binären Repräsentation. Die Textform (WAT) ist für uns Menschen gedacht und soll vor allem beim Debuggen unterstützen. Es ist jedoch nicht besonders effizient für einen Computer lesbar. Deshalb gibt es das Binärformat (WASM), das nur für den Computer gedacht und auf besonders effiziente Verarbeitung optimiert ist.

Beide Formate sind sehr Hardware-nah. Als Entwickler könnten wir solche Programme schreiben, was allerdings sehr ineffizient wäre. Der eigentliche Zweck von WebAssembly ist ein anderer: Entwickler schreiben Programme in einer Hochsprache wie C++, Rust, .NET, Java oder Kotlin. Ein Compiler übersetzt diese Sprache dann nach WebAssembly. WebAssembly ist also primär ein Compile-Target für eine Hochsprache unserer Wahl. Wir haben somit das Beste aus zwei Welten: hohe Produktivität in einer bekannten Hochsprache und maximale Laufzeit-Optimierung durch Einsatz eines optimierten Binärformats *(siehe Abbildung 1)*.

WebAssembly kennt sogenannte „Module" und „Instanzen". Ein Modul ist eine Art Schablone, aus der mehrere Instanzen erzeugt werden. Jede Instanz bekommt ihre eigene Sandbox und ist darin ausführbar. Das WebAssembly-Modul hat eine Liste von Imports und Exports. Ein Export ist eine Funktion, die von außerhalb der WebAssembly-Instanz aufgerufen werden kann, also aus der WebAssembly-Host-Umgebung. Der naheliegendste Anwendungsfall ist hier der Aufruf der Main Function des Programms.

Interessant sind die Imports. Ein Import ist eine Funktion, die von innerhalb der WebAssembly-Instanz aufgerufen werden kann, jedoch nicht in der Instanz definiert ist. Importierte Funktionen sind beispielsweise für die Interaktion mit der WebAssembly-Host-Umgebung erforderlich. Jede Form von I/O wie eine Konsolen-Ausgabe oder auch ein DOM-Zugriff wird üblicherweise in Form von Imports in WebAssembly gelinkt *(siehe Abbildung 2)*.

Um eine lauffähige WebAssembly-Instanz zu erhalten, ist eine Bootstrap-Sequenz auf dem WebAssembly-Host zu durchlaufen. Im Browser gibt es dafür ein JavaScript-API. Im ersten Schritt muss die binäre WebAssembly-Repräsentation geladen werden, was zum Beispiel über einen XML-HTTP-Request erfolgen kann. Wichtig ist, dass wir kein JSON- oder XML-Dokument als Antwort erwarten, sondern einen Array-Buffer mit den geladenen Binärdaten *(siehe Listing 1)*.

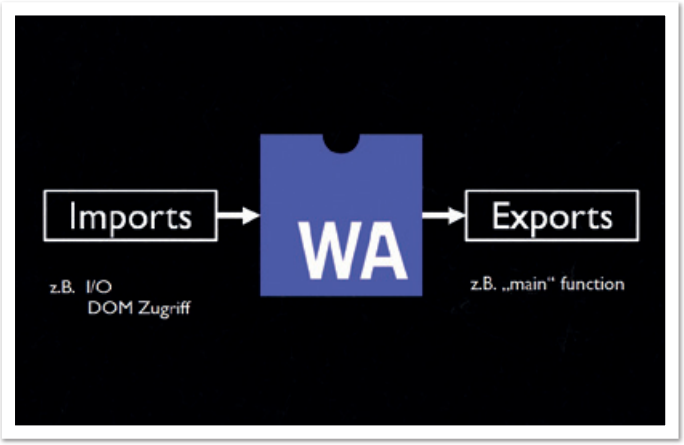

Abbildung 2: Das WebAssembly-Modulsystem

Anschließend wird via WebAssembly-JavaScript-API die Instanziierung gestartet. An diesem Punkt sind die Imports für das WebAssembly-Modulsystem anzugeben. Ein fehlender Import führt zum Abbruch der Instanziierung *(siehe Listing 2)*.

„WebAssembly.instantiate“ liefert ein Promise zurück. Bei der Instanziierung wird die WebAssembly-binäre Repräsentation validiert und kompiliert. Um den Main-Thread im Browser nicht zu blockieren, findet dies asynchron statt. Sobald dieser Vorgang abgeschlossen ist, wird das Promise mit den Referenzen auf das WebAssembly-Modul und die WebAssembly-Instanz erfüllt (fulfilled, siehe *Listing 3)*. Jetzt lässt sich der Kontrollfluss an die WebAssembly-Instanz übergeben, um etwa eine exportierte Main Function aufzurufen.

Lasst uns spielen

So weit, so gut. Welche Möglichkeiten bestehen, um erste Erfahrungen mit WebAssembly zu machen? Zum Glück gibt es da eine sehr interessante Webseite: WebAssembly Studio *(siehe „https://github.com/wasdk/WebAssemblyStudio“).* WebAssembly Studio ist „JSFiddle“ für WebAssembly. Über diese IDE kann sehr einfach ein C- oder Rust-Projekt angelegt und dieses dann nach WebAssembly kompiliert werden. Der JavaScript-Code für die Bootstrap-Sequenz wird automatisch generiert und lässt sich auch nachträglich editieren. Es ist auch möglich, die WebAssembly-Text- oder -Binärform genauer unter die Lupe zu nehmen und so tiefer in die Funktionsweise abzutauchen *(siehe Abbildung 3)*.

Wer nicht direkt Quellcode schreiben möchte, sondern eher eine lauffähige Anwendung benötigt, kann das Unity-Tanks-Demo spielen. Dabei handelt es sich um ein kleines Spiel, in dem Panzer in einer Landschaft bewegt werden können. Die Grafik ist sehr schön über WebGL umgesetzt. Hier zeigt sich sehr gut, wie WebAssembly bereits heute in Lösungen wie die Unity-Game-Engine integriert ist *(siehe Abbildung 4)*.

Andere Anwendungsfälle

Es gibt natürlich neben dem Gaming-Bereich auch weitere Anwendungsfälle. Einer davon ist „PocketSpinx.js“, ein System zur Spracherkennung. Es wurde via Emscripten nach WebAssembly übersetzt. Damit ist es möglich, Spracherkennung in Applikationen zu nutzen, ohne Remote-Services wie Alexa in eine Anwendung zu integrieren.

```
var request = new XMLHttpRequest();
request.open('GET', 'bytecoder.wasm');
request.responseType = 'arraybuffer';
request.send();
```

Listing 1

```
request.onload = function() {
  var bytes = request.response;

  WebAssembly.instantiate(bytes, {
     // Imports
     mymodule: {
add: function(a, b) {
return a + b;
}
     }
  }
    );
};
```

Listing 2

```
WebAssembly.instantiate(… ).then(function(result) {

  // Zugriff auf Modul und Instanz
  var wasmModule = result.module;
  var runningInstance = result.instance;

  // exportierte Funktion aufrufen
  runningInstance.exports.main();
});
```

Listing 3

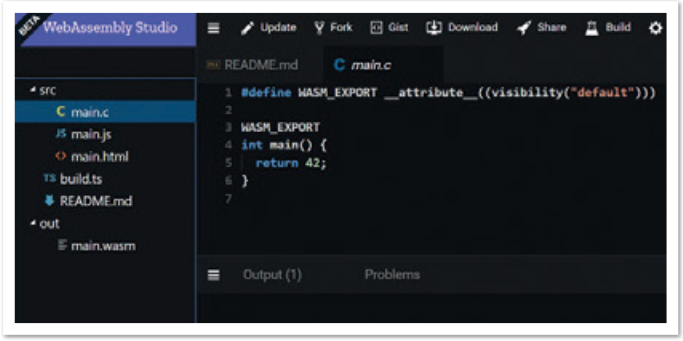

Abbildung 3: WebAssembly Studio

Abbildung 4: Unity-Tanks-Demo

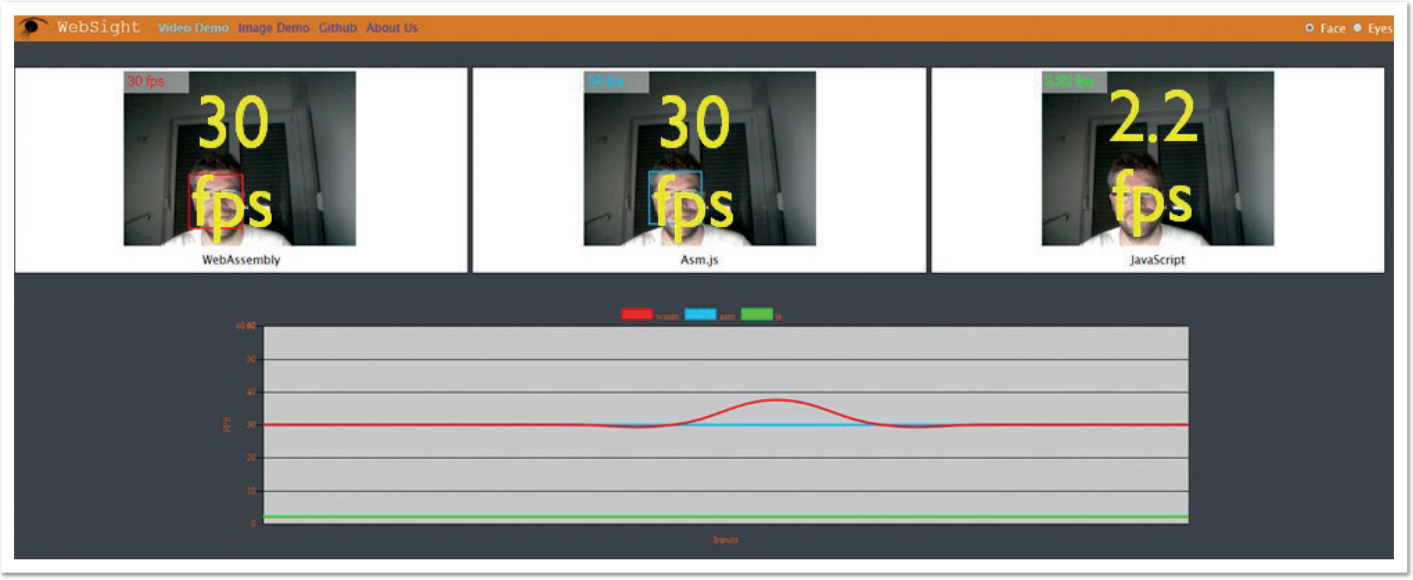

Abbildung 5: WebSight-Demo

„WebSight" ist ein anderer Anwendungsfall. Hier wurde OpenCV via Emscripten nach WebAssembly übersetzt. Diese Demo greift auf die Webcam zu und sucht in dem Live Video Stream nach Gesichtern. Interessant ist, dass gleich drei Varianten der Bildverarbeitung in Form von WebWorker-Instanzen parallel laufen. Eine WebAssembly-, eine „asm.js"- und eine JavaScript-Variante. Hier offenbart sich der mögliche Performance-Gewinn deutlich: Die WebAssembly-Variante ist um den Faktor 15 schneller als die JavaScript-Variante *(siehe Abbildung 5)*.

Missing Parts

Wo Licht ist, ist auch Schatten. Das WebAssembly-MVP funktioniert, jedoch zeigen sich in der aktuellen Version besonders die Stellen, an denen bewusst abgespeckt wurde. Aus Entwickler-Sicht fällt besonders das noch sehr einfache Tooling rund um die WebAssembly-Integration im Browser auf. Es gibt im Moment nur sehr einfache Debugging-Möglichkeiten.

Die ersten Browser beinhalten Sourcemap-Unterstützung für WebAssembly, diese könnte jedoch noch deutlich optimiert werden. Durch das Fehlen eines Memory-Managers und einer Garbage-Collection-Integration ist diese Funktionalität via Modul-System zu importieren. In der Konsequenz verursacht dieser Runtime-Code ein unnötig großes WebAssembly-Binary.

Das Fehlen eines direkten DOM-API oder des Zero-cost-Exception-Handling im MVP verursacht ebenfalls Glue Code zur Emulation von Hochsprachen mit Exception-Handling, was eigentlich nicht notwendig wäre. Unterstützung für Threading wäre wünschenswert, jedoch kann im MVP auch ohne native Thread-Unterstützung gearbeitet werden. Die WebSight-Demo zeigt es: Threads lassen sich zusammen mit dem WebWorker-API emulieren. Dies ist für die meisten Anwendungsfälle eine gute Grundlage.

Ausblick

Auch wenn das WebAssembly-MVP ein paar Ecken und Kanten hat, ist es doch schon sehr gut einsetzbar. Die Unity-Game-Engine nutzt bereits heute WebAssembly für die Spiele-Entwicklung. Unity kann jedoch mehr. Denkbar sind hier alle Formen von besonders interaktiven Produkt-Demonstrationen, die von WebAssembly profitieren können. Besonders in Verbindung mit Virtual Reality oder Augmented Reality ergeben sich völlig neue Anwendungsfälle.

Eine weitere interessante Möglichkeit ist die Migration von Legacy Code ins Web. Durch moderne Compiler können wir funktionierende und getestete Software in neuen Umgebungen laufen lassen, wie am Beispiel von WebSight-OpenCV auf einer Webseite. Die Möglichkeiten sind grenzenlos. Untermauert wird diese Perspektive durch die Übergabe der WebAssembly-Core-Spezifikation an das W3C im Februar 2018. WebAssembly ist somit ein offizieller Standard mit unglaublich viel Potenzial für die Zukunft. Das Web war schon immer gut für Überraschungen und wird es mit dieser Technologie auch weiterhin bleiben.

Mirko Sertic
mirko@mirkosertic.de

Mirko Sertic ist Software Craftsman im Web-/eCommerce-Umfeld. In Funktionen als Software-Entwickler, Architekt und Consultant in Projekten in Deutschland und der Schweiz sammelte er Erfahrungen mit einer Vielzahl von Frameworks, Technologien und Methoden. Heute arbeitet er als IT-Analyst bei der Thalia Bücher GmbH in Münster mit Schwerpunkt auf Java, eCommerce sowie Such- und Empfehlungs-Technologien. Seine Freizeit verbringt er mit seiner Freundin, seiner Familie und hin- und wieder mit Open-Source-Projekten.