Kurzfassung

Aufgrund der immensen Geschwindigkeit, in der sich die Webentwicklung vorwärtsbewegt, erlangen Anwendungen bereits in wenigen Monaten den Legacy-Status. Eines der größten Probleme stellt die Instandhaltung der benötigten Werkzeuge dar. Bei der Entwicklung unterschiedlich alter Projekte kommt es unweigerlich zu Problemen durch die unterschiedlichen Versionen der Werkzeuge.

Im Rahmen einer Analyse des Status der Frontend-Webentwicklung werden die dafür aktuell relevanten Werkzeuge gezeigt und eine Begründung für diesen Trend gesucht. Bevor auf den softaware/webdev-Container eingegangen wird, werden die Vor- und Nachteile von containerbasierten Entwicklerwerkzeugen dargestellt. Zusätzlich zu den Implementierungsdetails des softaware/webdev-Containers werden dessen Einsatzbereiche, die Motivation für die Entwicklung sowie die Besonderheiten und Probleme gezeigt.

Abstract

Due to the high speed and innovation regarding frontend web development, applications receive legacy state just a few months after the start of development. One of the biggest issues is to keep all required tools up-to-date and running. Often the tools are backwards-incompatible or have serious problems regarding versioning, putting the developer into trouble rather sooner than later.

As part of the analysis of the current state of frontend development, a comprehensive list of currently used tools is given, and a plausible reason for this legacy problem is presented. Before approaching the implementation details of the softaware/webdev-container the advantages and disadvantages of container-based development tools are listed. In addition to the implementation details of the container, possible use cases, as well as the initial motivation for its development and its particularities and problems, get explained.

Inhaltsverzeichnis

1 Einleitung

Im folgenden Kapitel wird das Unternehmen vorgestellt, in dem der Autor sein Praktikum absolviert hat, sowie eine Übersicht über die Projekte gegeben, in die er involviert war.

1.1 Das Unternehmen softaware gmbh

Die Firma softaware gmbh mit Sitz in Asten, Oberösterreich, entwickelt vorüberwiegend Individualsoftware auf Basis von Microsoft-Technologien. Seit 2012 existiert das Unternehmen unter dem Namen softaware gmbh und hat aktuell 15 Angestellte (Stand Juni 2017). Bei der Auswahl der Projekte gibt es keine Beschränkung auf bestimmte Branchen. Dadurch ergibt sich ein sehr breites Feld an Kunden, das von der Erdölindustrie über den öffentlichen Sektor bis hin zu Automobilkonzernen reicht. Die Art der entwickelten Anwendungen ist ebenfalls sehr breit gefächert, wodurch plattformübergreifende Apps genauso wie Web-Anwendungen und Enterprise-Desktop-Anwendungen entstehen.

Die Projekte werden in einer sehr flachen Unternehmensstruktur umgesetzt. Je nach Projektgröße werden Teams mit bis zu fünf Entwicklern gebildet, die in einem agilen Entwicklungsprozess an den Produkten arbeiten.

Die eingesetzten Technologien liegen überwiegend im Microsoft-Umfeld. Für große Webanwendungen ist Angular[1] das Framework der Wahl, für skalierbare Anwendungen wird Microsoft Azure[2] verwendet. Da beim Einsatz von Technologien großer Wert auf deren Aktualität gelegt wird, wurde im Rahmen des Praktikums der Einsatz von Docker[3] evaluiert. Durch die unterschiedlichen Versionen der Webprojekte und den erfolgreichen Tests von Docker ist der Container *softaware/webdev* entstanden, der in dieser Arbeit beschrieben wird.

1.2 Bearbeitete Projekte

Das Ziel des Praktikums war nicht die Abwicklung eines einzelnen Projekts, sondern Einblick in mehrere Bereiche des Unternehmens zu erhalten sowie die Erkenntnisse aus der theoretischen Bachelorarbeit des Autors im Unternehmen zu etablieren. Darüber hinaus wurde versucht, wiederkehrende Probleme in unterschiedlichsten Projekten zu analysieren und durch eine alternative Herangehensweise Lösungen zu finden. So wurden im Laufe des Praktikums folgende Projekte umgesetzt.

[1] `https://angular.io/`
[2] `https://azure.microsoft.com/`
[3] `https://www.docker.com/`

Windows Container Im Gegensatz zu Linux-Container sind Windows-Container noch sehr wenig im Einsatz. Für ein Unternehmen, das sehr stark auf Microsoft-Technologien setzt, können diese allerdings einen erheblichen Unterschied bedeuten. Der Funktionsumfang, die Stabilität und welcher Nutzen sich durch die Verwendung von Windows Containern ergibt, wurde in einer einwöchigen Analyse evaluiert. Die Möglichkeit, mithilfe von Containern ein einfacheres Testen von zeitzonenabhängigen Anwendungen zu ermöglichen, ist allerdings aufgrund eines Docker-Problems[4] zum jetzigen Zeitpunkt nicht möglich.

Microservices im Frontend Die Microservice-Architektur erlebt besonders serverseitig in den letzten Jahren einen enormen Aufschwung (vgl. [Fow14]). Die besonders einfache Verteilung der Aspekte einer Software auf mehrere Teams bietet große Vorteile, die allerdings verschwinden, sobald die Benutzeroberfläche mit aktuellen Frontend-Frameworks wie Angular oder React umgesetzt wird. Das Frontend wird zu einer einzelnen großen Anwendung, die zwar aus wiederverwendbaren Komponenten zusammengestellt wird, welche allerdings von der gewählten Frameworkversion abhängen und ein Aktualisieren dieser kostentechnisch beinahe unmöglich machen. In einem einmonatigem Forschungsprojekt wurden Möglichkeiten untersucht, wie Anwendungen zukunftssicherer entwickelt werden können und vor allem, wie neue Funktionalität mit aktuellen Technologien in ältere Anwendungen integriert werden kann.

Android-App Für einen deutschen Kunden wurde in einem einwöchigen Projekt eine Android-App auf Basis von Xamarin entwickelt, die die Wiedergabe von Audio- und Videodateien ermöglicht. Die Besonderheiten dabei waren eine möglichst einfache Benutzeroberfläche und erweiterte Funktionalität zum Vor- und Rückwärtsspringen in der Mediendatei sowie eine einstellbare Wiedergabegeschwindigkeit.

softaware/webdev-Container Wie Probleme, die in der Webentwicklung durch den Einsatz von Werkzeugen wie *npm*[5] oder *yarn*[6] entstehen, mithilfe von Docker gelöst werden können, wird in dieser Arbeit beschrieben. Im Praktikum wurden rund fünf Wochen mit der Identifizierung der Probleme, der Entwicklung des Containers und der Einführung in realen Projekten verbracht.

[4] `https://github.com/moby/moby/issues/32518`
[5] `https://www.npmjs.com/`
[6] `https://yarnpkg.com/`

2 Aktuelle Probleme der Frontend-Webentwicklung

Das aktuell größte Problem der Frontend-Webentwicklung wird in [Kir16] *Hype Driven Development* genannt. Damit ist die enorm hohe Frequenz gemeint, in der neue Technologien entstehen, hochgepriesen werden und danach genauso schnell wieder von neueren verdrängt werden.

Gerade im Web-Bereich ist durch den Trend zu Open-Source-Software eine sehr hohe Obsoleszenz zu beobachten, da durch das Abzweigen (engl. *to fork*) im Entwicklungsgraphen verschiedene Varianten von gesamten Projekten entstehen. Der daraus resultierenden Innovation stehen allerdings die Spaltung von Entwicklergemeinschaften und inkompatible Technologien gegenüber.

Ein Beispiel dafür ist die Teilung von Node.js[1] in die zwei Projekte Node.js und io.js[2] im Jahr 2014, die allerdings aus den vorhin beschriebenen Problemen 2015 wieder zu Node.js vereint wurden [Lin15]. Kurzlebige, innovative Technologien bringen die Softwareentwicklung im Generellen voran, doch um eine Technologie in seriösen Anwendungen verwenden zu können, ist ein gewisser Reifegrad notwendig. Dieser ermöglicht nicht nur stabile Versionen und besser ausgebildete Entwickler durch einen Einzug der Technologie in die Lehre, sondern vor allem durchdachte und ausgereifte Werkzeuge für Entwickler.

Nach erfolgreicher Auswahl eines Frameworks ist in der Frontend-Entwicklung die größte Hürde die Konfiguration der benötigten Werkzeuge. Die Schwierigkeit liegt in dem ständigen Erscheinen neuer Werkzeuge und der beinahe täglichen Aktualisierung dieser, wodurch teilweise inkompatible Änderungen verursacht werden.

Hilfsanwendungen wie create-react-app[3] oder Angular CLI[4] erleichtern die Erstellung von Webanwendungen, indem sie ein initiales Projekt konfigurieren, in dem bereits alle benötigten Abhängigkeiten und Werkzeuge integriert sind.

2.1 Werkzeugübersicht

Im Folgenden werden einige der Werkzeuge beschrieben, um einerseits deren Schnelllebigkeit zu veranschaulichen und andererseits einen Überblick zu liefern, welche Werkzeuge wozu aktuell eingesetzt werden.

[1] `https://nodejs.org/`
[2] `https://iojs.org/`
[3] `https://github.com/facebookincubator/create-react-app`
[4] `https://github.com/angular/angular-cli`

2.1.1 JavaScript-Modulsysteme

Die Informationen dieses Abschnitts stammen aus [Osm12] und [Pey16].

Im Gegensatz zum aktuellen Einsatzzweck von JavaScript war diese Sprache ursprünglich nicht für das Entwickeln großer Anwendungen, die aus zahlreichen Modulen bestehen, gedacht. Der ursprüngliche Zweck war es, Webseiten Interaktivität zu verleihen.

Konzepte wie Kapselung und die Verwendung von anderem Code als Modul wurden dabei nicht benötigt. Bis zur Version 6 von JavaScript ist der Sichtbarkeitsbereich einer Variablen global und Konzepte wie Namensräume sind kein Element der Sprache. Bei der Aufteilung einer JavaScript-Anwendung in mehrere Teile muss genau darauf geachtet werden, dass diese in der richtigen Reihenfolge geladen werden, um keine undefinierten Funktionen und Variablen zu erhalten.

Als JavaScript-Anwendungen immer größer wurden, entstanden einige Versuche, ein Modulsystem mit bereits existierenden Sprachmitteln nachzubauen. Im Folgenden werden die drei relevantesten dargestellt.

CommonJS

Das Ziel von CommonJS ist eine einheitliche Spezifikation für serverseitiges JavaScript. Ein Teil davon ist das Modulsystem, das in einer abgewandelten Form auch von Node.js verwendet wird. Die API besteht aus zwei Teilen:

1. `exports` ist eine Variable, der am Ende eines Modules (einer Datei) die zu exportierende Funktionalität zugewiesen wird.

2. Über die Funktion `require(<Modulname>)` kann eine andere Datei geladen und einer Variablen zugewiesen werden.

Da dieses Modulsystem lediglich eine Spezifikation ist, werden sowohl browserseitig wie serverseitig Bibliotheken benötigt, die das Laden von Modulen ermöglichen. Für das Laden im Browser werden Werkzeuge wie Webpack (siehe Abschnitt 2.1.3) oder browserify[5] verwendet, am Server ist Node.js die Standardimplementierung. Das Laden von Modulen ist synchron. Da es jedoch mithilfe eines Funktionsaufrufes geschieht, kann es überall im Quelltext geschehen und beispielsweise durch Verzweigungen optimiert werden.

AMD

Genau wie bei den Problemen der Frontend-Webentwicklung (vgl. Kapitel 2) beschrieben, ist *AMD* (Asynchronous Module Definition) eine Variante von CommonJS. Der Unterschied dazu ist lediglich die Möglichkeit des asynchronen Ladens von Quelltexten, wodurch es sich besonders für Browser eignet, da dadurch die Ladezeit einer Webseite verringert wird. Diese Funktionalität wird durch JavaScript-Callbacks ermöglicht, indem Code erst ausgeführt wird, nachdem die Abhängigkeiten geladen wurden. Die

[5] `http://browserify.org/`

De-facto-Standardbibliothek für das asynchrone Laden von Modulen im Browser ist require.js[6].

ES2015

JavaScript ist eine Implementierung des ECMAScript-Standards[7]. Daher muss eine Änderung in JavaScript zuvor im ECMAScript-Standard definiert werden. In ECMAScript 2015 wurde die Möglichkeit des Ladens von Modulen durch die zwei neuen Sprachkonstrukte `import` und `export` eingeführt. Seit 2017 gibt es auch einen Vorschlag für das asynchrone Laden von Modulen, dessen Umsetzung für eine zukünftige ECMAScript-Version geplant ist. Die Integration in die Sprache erlaubt es statischen Codeanalysewerkzeugen, einen Abhängigkeitsgraphen aufzubauen und erweiterte Funktionalität wie Autocompletion zu liefern. Dieser Standard wird Modulsysteme wie CommonJS oder AMD definitiv ablösen. Da er allerdings noch nicht in jedem Browser unterstützt wird, sind zurzeit noch Werkzeuge notwendig, die diese Abhängigkeiten auflösen und im Vorhinein ein komplettes JavaScript-Programm erstellen.

2.1.2 Paketmanager

Paketmanager sind in der Web-Welt beinahe ein Glaubenskrieg. [Far15] beschreibt bereits 2015 sieben unterschiedliche Paketmanager. Die grundsätzliche Aufgabe eines Paketmanagers ist das Zurverfügungstellen von Modulen. Dazu ist ein einheitliches Paketformat, eine zentrale Stelle zum Verteilen der Module und eine Anwendung zur Verwaltung dieser notwendig. Aktuell wird versucht, die zahlreichen Paketmanager zu vereinen. Ansonsten benötigt jeder Entwickler das Know-how für mehrere Paketmanager und muss diese auch installiert haben. Zurzeit finden die folgenden drei Paketmanager große Verbreitung:

Bower

Bower[8] wurde ursprünglich als Paketmanager für Frontend-Pakete entworfen. Der Unterschied zu den zwei nachfolgend beschriebenen ist eine flache Paketstruktur. Um die Geschwindigkeit beim Laden von Webanwendungen zu erhöhen, verfolgt Bower den Ansatz, dass Module, die von mehreren anderen Modulen benötigt werden, lediglich einmal installiert werden. Etwaige Versionskonflikte muss der Entwickler per Hand beheben. Der Trend in der Webentwicklung geht allerdings dazu, dass die gesamte Webentwicklung von einem Build-Werkzeug erstellt wird, wodurch das manuelle Einbinden von Abhängigkeiten entfällt. Dadurch empfiehlt Bower sogar auf der Homepage die Verwendung von yarn (siehe Abschnitt 2.1.2) und Webpack (siehe Abschnitt 2.1.3).

[6] `http://requirejs.org/`
[7] `https://www.ecma-international.org/publications/standards/Ecma-262.htm`
[8] `https://bower.io/`

npm

npm war ursprünglich der Paketmanager für Node.js. Die gesamte Paketstruktur wurde als Baum abgelegt und Abhängigkeiten für jedes Paket separat heruntergeladen, da dies für Serveranwendungen beinahe keinen Unterschied macht und zu keinen Versionsproblemen führt. Mit der Zeit wurden immer mehr Frontend-Pakete auf Basis von npm veröffentlicht, weshalb mit der Version 3 das Versionssystem verbessert wurde und für kompatible Pakete eine Struktur wie bei Bower verwendet wird. Am 25. Mai 2017 wurde die Version 5 veröffentlicht, um mit der Funktionalität von yarn gleichzuziehen.

yarn

yarn ist ein Paketmanager von Facebook, der die Paket-Registry von npm verwendet und einige Probleme dieses Paketmanagers behebt. Die Geschwindigkeit wird durch die Einführung eines lokalen Caches verbessert. Bei npm gab es bis zur Version 5 das Problem, dass lediglich die Version der verwendeten Abhängigkeiten spezifiziert werden konnte, nicht aber die Versionen von transitiven Abhängigkeiten. Dadurch können sehr schwierig zu findende Fehler durch unterschiedliche Paketversionen entstehen. yarn löst dieses Problem durch eine Sperrdatei, in der die Version aller installierten Pakete erfasst ist.

Zwei Ungereimtheiten bei yarn sind die irreführende Versionierung des Paketmanagers selbst, wonach derzeit die stabile Version bei 0.24.6 steht, und die Tatsache, dass zwar die Registry von npm verwendet wird, jedoch jegliche Anfragen über einen Proxy von Facebook geleitet werden [Nem17].

2.1.3 Build-Werkzeuge

Webentwicklung im Jahr 2017 besteht nicht mehr aus dem Erstellen von HTML-, CSS- und JavaScript-Dateien, wobei die Styles und Skripte in der HTML-Datei verlinkt werden. Frameworks wie Angular und React stellen Komponenten in den Vordergrund. Jede Komponente besitzt ein Template, Styles und Logik.

Die fertige Anwendung besteht aus zahlreichen Komponenten, die dynamisch geladen werden, mit einzelnen Services kommunizieren oder auch mit der Hardware des Gerätes interagieren. Anwendungen in dieser Komplexität erfordern eine wesentliche ausgereiftere Palette an Werkzeugen als einen simplen Texteditor.

In den letzten Jahren haben sich diverse Build-Werkzeuge etabliert, die aus zahlreichen Dateien und Quelltexten eine fertige, für den Browser optimierte Anwendung erstellen. Sie unterscheiden sich in der Funktionsweise und der Funktionalität, jedoch vor allem in der Anzahl der Plugins. Alle sind jedoch in JavaScript geschrieben, laufen auf Node.js und lassen sich in die *package.json*-Datei von npm integrieren. Die nachfolgend beschriebenen Build-Werkzeuge wurden aufgrund ihrer großen Verbeitung ausgewählt.

Grunt

Grunt[9] ist ein Task-Runner, der wiederkehrende Aufgaben automatisiert. Die Aufgaben werden im sogenannten *Gruntfile* deklarativ angegeben. Darin wird an Grunt eine Konfiguration als JavaScript-Objekt übergeben, in der die zahlreichen Grunt-Plugins konfiguriert werden.

Gulp

Gulp[10] ist genau wie Grunt lediglich ein Task-Runner. Der Hauptunterschied liegt in der Konfiguration. Während die Plugins von Grunt über ein JavaScript-Objekt konfiguriert werden, ist in Gulp jeder Task ein Strom von Dateien. Diese werden aus einer Quelle ausgewählt, danach werden sie von einer Reihe von Plugins transformiert und schlussendlich wieder in ein Zielverzeichnis geschrieben. Durch diese Art der Konfiguration ist Gulp wesentlich flexibler als Grunt, allerdings auch schwieriger zu konfigurieren und diese Konfiguration ist schwieriger les- und wartbar.

Webpack

Webpack[11] ist ein Module-Bundler. Im Gegensatz zu einem Task-Runner werden nicht einzelne Aufgaben abgearbeitet, sondern der Module-Bundler baut einen Graphen aller Abhängigkeiten der Applikation auf, bevor diese nach vorweg definierten Regeln erstellt wird. Diese Regeln werden genau wie bei Grunt oder Gulp von Plugins ausgeführt. Bei Webpack werden alle Teile einer Anwendung als Modul gesehen. Aufgrund des Graphen weiß Webpack genau, welche Bilder, Styles und Quelltexte wirklich referenziert werden und kann die erstellte Anwendung sehr stark optimieren. Eine weitere, sehr nützliche Funktion ist der *webpack-dev-server*, der Änderungen am Quelltext erkennt, danach automatisch eine neue Version der Anwendung erstellt und den Browser, in dem die Anwendung läuft, aktualisiert. Alle aktuellen großen Frameworks verwenden Webpack zum Erstellen der Anwendung, wodurch seit Mitte 2017 Grunt und Gulp bei neuen Projekten kaum mehr Verwendung finden.

Rollup

Rollup[12] ist ein Module-Bundler für Bibliotheken [Har17]. Während Webpack besonderen Wert auf das Teilen von Quelltext zum asynchronen Laden und auf die Verwaltung von statischen Dateien legt, ist der primäre Fokus von Rollup das Erstellen von optimierten Bibliotheken. Dazu ermöglicht Rollup auf Basis der Modulsysteme CommonJS und ECMAScript 2015 *Tree Shaking*. Dabei wird mithilfe statischer Quelltextanalyse jener Quelltext, der nicht benötigt wird, entfernt und eine Bibliothek erstellt, die so kompakt wie möglich ist.

[9] `https://gruntjs.com/`
[10] `http://gulpjs.com/`
[11] `https://webpack.github.io/`
[12] `https://rollupjs.org/`

2.2 Inkompatible Node.js/npm-Versionen

Die Basis für die meisten Werkzeuge in der Frontend-Webentwicklung bildet *Node.js*. Node.js ist eine Laufzeitumgebung, die auf der Chrome-V8-JavaScipt-Engine basiert und es ermöglicht, JavaScript ohne Browser auf dem System nativ auszuführen. Der große Vorteil, der vor allem Entwicklerwerkzeuge betrifft, ist die Plattformunabhängigkeit. Dadurch können Kommandozeilenwerkzeuge mithilfe von Webtechnologien für alle gängigen Betriebssysteme erstellt werden.

npm ist ein Paketmanager, der es ermöglicht Node.js-Anwendungen zu installieren oder Abhängigkeiten zwischen Paketen zu spezifizieren.

Wie in [Pap16] beschrieben, kann es allerdings sein, dass bestimmte Werkzeuge unter bestimmten Node.js-Versionen nicht funktionieren. Auch das Testen von Anwendungen unter verschiedenen Node.js-Versionen gestaltet sich schwierig, da dafür jedesmal eine Neuinstallation notwendig ist. Die benötigte Node.js-Version kann nicht in einer Art und Weise versioniert werden, die plattformübergreifend funktioniert und eine Warnung bei dem Vorhandensein einer falschen Version liefert. Sie kann maximal in einer Readme-Datei erfasst werden, wobei jeder Entwickler selbst sicherstellen muss, dass er die korrekte Version verwendet. Bei der Entwicklung mehrerer Projekte wird dieses Problem noch gravierender.

2.3 Parallele Entwicklung mehrerer Projekte

Wie in Abschnitt 1.1 erläutert, ist es in der Firma softaware gmbh üblich, dass ein Entwickler an mehreren Projekten gleichzeitig arbeitet. Dadurch tritt das vorhin erwähnte Problem häufiger auf, da es vorkommen kann, dass bei der Entwicklung unterschiedlicher Projekte unterschiedliche Node.js-Versionen notwendig sind.

Ein Lösungsansatz dafür ist die Verwendung von Vagrant[13]. Vagrant ist ein Werkzeug zur automatischen Verwaltung von virtuellen Maschinen und ermöglicht dadurch die deklarative Beschreibung von Entwicklerrechnern. Diese Beschreibung kann mitversioniert werden, wodurch eine konsistente Arbeitsumgebung entsteht. Eine virtuelle Maschine benötigt allerdings erhebliche Ressourcen. Wenn keine grafische Oberfläche benötigt wird, gibt es daher bessere Lösungsansätze.

Ein weiterer Ansatz ist die Verwendung von sogenannten Node.js-Versionsmanagern. Beispiele dafür sind nvm[14], n[15] oder nvm-windows[16]. Das Hauptproblem dabei ist allerdings, dass es keine Lösung gibt, die plattformübergreifend funktioniert. Außerdem wurde die Entwicklung von nvm-windows eingestellt, wodurch für Windows aktuell keine Lösung existiert. Ein weiteres Problem ist die fehlende Integration in den Quelltext eines Softwareprojekts. Daher werden diese Werkzeuge als eigenständige Anwendung auf dem Entwicklerrechner installiert und stellen lediglich eine implizite Abhängigkeit des Projektes dar, die nur schwierig automatisch installiert werden kann.

[13] `https://www.vagrantup.com/`
[14] `https://github.com/creationix/nvm`
[15] `https://github.com/tj/n`
[16] `https://github.com/coreybutler/nvm-windows`

Nicht nur die parallele Entwicklung mehrerer Projekte kann Probleme bereiten. In der Firma softaware gmbh kommt es außerdem regelmäßig vor, dass andere Software-Unternehmen als Projektpartner fungieren. Gerade in diesem Fall ist es wichtig, eine einheitliche Lösung zur Verwaltung von Node.js-Versionen zu finden, da für diese Projektpartner die Hemmschwelle zur gemeinsamen Entwicklung möglichst niedrig sein soll.

2.4 Globale Installation von npm-Paketen

Für die im letzten Abschnitt erwähnte Installation von npm-Paketen existieren zwei Möglichkeiten.

1. Für diverse Werkzeuge wird `npm install -g <Werkzeug>` als Installationskommando vorgeschlagen, wobei das Paket durch den Parameter `-g` global installiert wird. Diese Art der Installation bedeutet, dass diese Anwendungen mit ihrem jeweiligen Namen direkt von der Kommandozeile gestartet werden können. Diese Funktionalität hört sich sehr verlockend an, erfordert allerdings, dass jeder Entwickler dieses Werkzeug global installiert. Dies mag in einer Abteilung problemlos funktionieren, wird aber spätestens bei Open-Source-Projekten zum Problem, da die Mitentwicklung an mehrerer Open-Source-Projekten durch die Vielzahl an Werkzeugen erheblich komplizierter wird. Außerdem kann es zu Versionsproblemen kommen, falls zwei Projekte dasselbe Werkzeug in unterschiedlichen Versionen benötigen, da global lediglich eine Version installiert werden kann.

2. Die zweite Möglichkeit der Installation ist das Weglassen des Parameters `-g`, wodurch das Paket in der Manifest-Datei `package.json` des Projekts erfasst und in einen Ordner namens `node_modules` im aktuellen Verzeichnis installiert wird. Da dieses Paket dadurch als Abhängigkeit erfasst ist, lässt es sich mit dem Projekt gemeinsam versionieren und wird bei dessen Installation auf einem anderen Entwicklerrechner automatisch mitinstalliert. Allerdings lassen sich diese Pakete nun nicht mehr mit ihrem Namen auf der Kommandozeile ausführen, weshalb es verlockend klingen mag, `./node_modules/.bin/` zur `$PATH`-Variable hinzuzufügen. Dadurch können lokal installierte Pakete einfacher ausgeführt werden. Doch diese Änderung birgt jedoch zwei wesentliche Probleme:

   a) Erstens müsste dies jeder Entwickler machen, wodurch eine einfache Zusammenarbeit erschwert wird und wieder eine implizite Abhängigkeit des Projekts entsteht.

   b) Das wesentlich schwerwiegendere Problem betrifft allerdings die Sicherheit des Rechners. Wie in [typ12] beschrieben, ermöglicht es diese Änderung der Systemvariablen Angreifern, infizierte Versionen von Standardprogrammen wie `ls` oder `cd` unter `./node_modules/.bin/` abzulegen und dadurch kompletten Zugriff auf das Gerät zu erlangen.

Empfohlene Vorgehensweise

Um die Vorteile von global und lokal installierten Paketen zu kombinieren, gibt es bei dem Installieren von npm-Paketen den Parameter `--save-dev`. In der Manifestdatei `package.json` existiert ein eigener Bereich *devDependencies*, in dem Werkzeuge erfasst werden, die für das Projekt benötigt werden. Zusätzlich bietet das Manifest mithilfe der *npm-scripts* die Möglichkeit, eigene Namen für Kommandos zu vergeben, die dann mit `npm run <Skriptname>` gestartet werden können. Der Ausführungskontext dieser Kommandos beinhaltet automatisch den Pfad `./node_modules/.bin/`, wodurch dort installierte Werkzeuge verwendet werden können. Dies ermöglicht einheitliche Kommandos und eine automatische Dokumentation über Plattformgrenzen hinweg.

Außerdem bieten *npm-scripts* den Vorteil, dass das intern verwendete Werkzeug ausgetauscht werden kann (z. B. Webpack anstatt Gulp), für den Entwickler das Kommando aber beispielsweise weiterhin `npm run build` bleibt und er durch diesen Austausch nichts in seinem Workflow ändern muss.

3 Lösungsvorschlag auf Basis von Docker

Die Einsatzmöglichkeiten von Containern als Werkzeug zur Unterstützung des Softwareentwicklers wurden bereits in der theoretischen Bachelorarbeit des Autors beschrieben. Die Kapselung von Werkzeugen in einen Container bietet folgende Vorteile:

- Falls keine keine grafische Benutzeroberfläche benötigt wird und eine Kommandozeile ausreicht, bieten Container eine konsistente Möglichkeit der Virtualisierung ohne den Overhead von Vollvirtualisierungslösungen, wie z. B. mit Vagrant.
- Container lassen sich gemeinsam mit dem Projekt versionieren, sodass alle Softwareentwickler im Team die gleiche Version der Werkzeuge verwenden.
- Die Anzahl der benötigten Werkzeuge der Entwickler wird reduziert, da im Optimalfall lediglich eine Engine zum Starten der Container (z. B. Docker) benötigt wird.
- Falls ein Container gestartet wird, dessen Image auf dem System noch nicht vorhanden war, wird dieser automatisch heruntergeladen und gestartet. Dadurch entfällt die manuelle Installation der Werkzeuge, wodurch Probleme wie fehlende Benutzerberechtigungen, falsche Werkzeuge oder inkompatible Versionen dieser der Vergangenheit angehören.

[Dem16] ist ein exzellenter Artikel über die Verwendung von Containern in der Webentwicklung. Am Ende des Artikels wird zur Erstellung von Containern für diverse Werkzeuge aufgerufen. Im Besonderen gilt dieser Aufruf Legacy-Projekten und Open-Source-Projekten, da diese eine besonders hohe Einstiegshürde haben. Dies liegt vor allem an der Installation und Konfiguration der benötigten Werkzeuge.

Für die Entwicklung von Node.js-basierten Webprojekten existiert bereits der offizielle Node.js-Container. Dieser ist allerdings für den Betrieb und nicht die Entwicklung von Anwendungen ausgelegt.

Das Konzept des softaware/webdev-Containers ist einerseits, diesen den Node.js-Container um Funktionen zu erweitern, die eine angenehme interaktive Entwicklung von Webprojekten ermöglichen. Andererseits wird ein Großteil der Entwicklung der Dokumentation gewidmet, die eine Sammlung von Best Practices und durch die Verwendung des Containers gewonnenen Erfahrungen darstellt.

Im folgenden Kapitel werden die Implementierung dieses Containers sowie der Weg dorthin dargestellt und die dabei gewonnenen Erfahrungen beschrieben.

4 Erstellung des softaware/webdev-Docker-Containers

In diesem Abschnitt wird beschrieben, aus welchen Bestandteilen der Container besteht, welche Entscheidungen bei der Entwicklung getroffen wurden und welche Probleme und Hürden sich dabei ergaben. Weiters wird dargestellt, in welchen Aspekten der Container durch neue Erkenntnisse beim Einsatz in Kundenprojekten weiterentwickelt wurde.

4.1 Anwendungsszenarien

Wie in Kapitel 3 erläutert, ist die grundlegende Idee des Containers, Node.js und npm oder yarn als versionierbare Abhängigkeit zu einem Projekt hinzuzufügen. Anstatt der lokalen Installation dieser Werkzeuge wird lediglich Docker auf dem System benötigt. Der Container kann, wie in Quelltext 4.1 dargestellt, gestartet werden.

```
1 docker container run -it --rm -v ${pwd}:/usr/src/app softaware/
    ↪ webdev:alpine -8.1.2
```

Quelltext 4.1: Kommando zum Starten des softaware/webdev-Containers

Das Kommando `docker container run` entstammt der neuen Docker-CLI (seit Version 1.13 enthalten) und erzeugt einen Container auf Basis eines Images. Der Parameter `-it` erzeugt eine interaktive Shell, die mit dem Container verbunden ist. Durch `--rm` wird der Container nach dem Stoppen wieder entfernt. Container sollten grundsätzlich keine Daten beinhalten, was durch diese Option sichergestellt wird und das System sauber hält. Erst durch `-v ${pwd}:/usr/src/app` können die Daten des Containers persistiert werden. Dieses Kommando bindet das aktuelle Verzeichnis in das Arbeitsverzeichnis des Containers ein. Dadurch können sowohl der Host als auch der Container gleichzeitig mit diesem Verzeichnis arbeiten, wodurch es möglich wird, dass die Entwicklungsumgebung am Host läuft, npm allerdings im Container. Am Ende des Kommandos wird das Docker-Image spezifiziert. In diesem Fall wird die Alpine Linux-Version (vgl. Abschnitt 4.4) mit der Node.js-Version 8.1.2 verwendet. Das Ergebnis dieses Kommandos ist in Abb. 4.1 zu sehen.

Verwendungsmöglichkeiten des Containers

Für den Container sind grundsätzlich die folgenden drei Anwendungsszenarien vorgesehen:

```
bemayr D: ››› docker container run -it --rm -v ${pwd}:/usr/app/src softaware/webdev:alpine-8.1.2

Your Webdevelopment-Environment is ready to go - Start Coding... ;)

dev@docker:[app]>
```

Abbildung 4.1: Ausgabe beim Start des Containers

1. Wie im vorangegangenen Beispiel gezeigt, können mithilfe des Containers sehr schnell spezifische Node.js- und npm-Versionen gestartet werden. Dies kann z. B. zum Initialisieren eines Webprojektes durch `npm init` nützlich sein.

2. Weiters kann vom softaware/webdev-Image abgeleitet werden, um ein eigenes spezifischeres Image zu erstellen. In diesem können z. B. weitere Werkzeuge installiert werden oder Ports des Containers explizit freigegeben werden.

3. Falls mithilfe der ersten Anwendungsweise ein Projekt erstellt wurde, wird empfohlen, danach das Startkommando mit derselben Containerversion als Skript zum Projekt hinzuzufügen. Dadurch kann die Entwicklungsumgebung konsistent reproduziert werden und als Abhängigkeit mit dem Projekt mitversioniert werden. Ein Beispiel eines realen Projekts ist in Abschnitt 4.6 beschrieben.

Im Folgenden wird der *Dockerfile* erläutert, das die Basis des Containers darstellt und die eben beschriebenen Funktionen ermöglicht.

4.2 Dockerfile

In Quelltext 4.2 ist der Dockerfile dargestellt, der die Basis des softaware/webdev-Containers bildet. Dieser wurde vom Autor im Rahmen der Bachelorarbeit erstellt.

```
1  # specify the wanted node/npm version tag as listed in https://
       ↪ hub.docker.com/r/library/node/tags/
2  FROM node:{{ node_version }}-alpine
3  LABEL maintainer "bernhard.mayr@softaware.at"
4
5  # set path in a way that local npm-modules (e.g. ng-cli) can be
       ↪  called like executables
6  ENV PATH="./node_modules/.bin:${PATH}"
7
8  # install bash
9  RUN apk --no-cache add bash
10
11 # set the working directory
12 WORKDIR "/usr/src/app"
13
14 # prepare bash
```

```
15 COPY ./files/.bashrc /root/
16 RUN npm completion >> /root/.bashrc
17
18 # start a new bash session
19 CMD ["/bin/bash"]
```

Quelltext 4.2: Alpine-Dockerfile

Im Folgenden werden die einzelnen Kommandos des in Quelltext 4.2 abgebildeten Dockerfiles Zeile für Zeile erläutert.

Zeile 2

Die `FROM`-Anweisung ermöglicht es, dass Docker-Images voneinander erben können. In diesem Fall wird vom offiziellen Node.js-Image[1] geerbt. Dieses enthält Node.js, npm und yarn, wodurch die gesamte Basisfunktionalität bereits vorhanden ist.

`{{ node_version }}` ist keine Funktionalität von Docker, dies ist ein Platzhalter für die Versionsnummer des Node.js-Images im Build-Prozess (vgl. Abschnitt 4.3). Der Name des offiziellen Node.js-Images beginnt mit der Node.js-Versionsnummer und enthält die Art des Images als Suffix. In diesem Fall wird vom *alpine*-Image abgeleitet und der Platzhalter im Build-Prozess durch die konkrete Versionsnummer ersetzt.

Zeile 3

Die `MAINTAINER`-Anweisung in Dockerfiles wurde ab Version 1.13 durch das flexiblere `LABEL` ersetzt. Durch den *maintainer* werden Metadaten erstellt, die angeben, wer für dieses Image verantwortlich ist, und an wen sich dessen Benutzer wenden kann.

Zeile 6

Wie im Kommentar in Zeile 5 beschrieben, wird durch die `ENV`-Anweisung die Umgebungsvariable `PATH` erweitert. Diese Änderung ermöglicht es, lokal installierte Node.js-Anwendungen als Kommandos zu verwenden.

In Abschnitt 2.4 wurde bereits der Unterschied zwischen global und lokal installierten Anwendungen erläutert. Im Container ist allerdings aufgrund der Isolierung das in [typ12] beschriebene Sicherheitsrisiko wesentlich geringer. Anstatt der Erstellung eines eigenen Container-Images mit global installierten Paketen sollten diese immer in der *package.json*-Datei erfasst werden. Dadurch muss kein eigenes Image gewartet werden und auch ohne den Container sind alle Abhängigkeiten des Webprojekts definiert.

Diese Änderung an der Umgebungsvariable ist lediglich aus Kompatibilitätsgründen vorhanden, damit auch ältere Projekte, die sich historisch bedingt auf globale Abhängigkeiten verlassen, im Container verwendet werden können. Die Konfiguration eines neuen Webprojekts sollte allerdings nach Abschnitt 2.4 erfolgen.

[1] `https://hub.docker.com/_/node/`

Zeile 9

Hier wird unter der Verwendung des Alpine-Paketmanagers *apk* die Bourne Again Shell installiert. Im Gegensatz zur vorinstallierten Almquist Shell ist sie wesentlich weiter verbreitet und bietet mehr Funktionen. Durch den `--no-cache` Parameter wird die Größe des fertigen Docker-Images möglichst klein gehalten.

Zeile 12

In Zeile 12 wird das aktuelle Arbeitsverzeichnis des resultierenden Containers auf `/usr/src/app` gesetzt. Die `WORKDIR`-Anweisung eines Dockerfiles entspricht im Wesentlichen dem Wechsel in dieses Verzeichnis mit dem Kommando `cd`.

Zeile 15

Das Aussehen der Eingabeaufforderung wird in Zeile 15 geändert. Der Zweck dieser Änderung ist, dass der Entwickler auf den ersten Blick erkennen kann, ob er sich gerade innerhalb des Containers oder außerhalb befindet.

Durch das Dockerfile-Kommando `COPY` wird die Datei *.bashrc* in den Container kopiert. Diese ist in Quelltext 4.3 abgebildet. Darin wird in Zeile 2 der Eingabeaufforderung durch das Setzen der Umgebungsvariable `PS1` das Format *dev@docker:[<aktuelles Verzeichnis>]>* zugewiesen. Die kryptisch aussehende Notation ändert lediglich die Anzeigefarbe des aktuellen Verzeichnisses. In Zeile 5 wird mit `printf` eine Nachricht beim Start des Containers ausgegeben. Das Resultat ist in Abb. 4.1 zu sehen.

```
1 # --- configuring the prompt to look like "dev@docker:[<current
    ↪ -directory>]>" --- #
2 export PS1="dev@docker:\e[1;36m[\W]\e[m> "
3
4 # --- printing a welcome message --- #
5 printf '\nYour Webdevelopment-Environment is ready to go -
    ↪ Start Coding... ;)\n\n'
```

Quelltext 4.3: *.bashrc* (Bash-Konfigurationsdatei)

Zeile 16

Beim Ausführen von Skripten, oder beispielsweise dem Deinstallieren von Paketen, bietet npm auf der Kommandozeile die Möglichkeit der Autovervollständigung. Diese wird aktiviert, indem das Kommando `npm completion` ein Shell-Skript erstellt, das an die benutzerspezifische Shell-Konfiguration angehängt wird.

Das Ergebnis von `npm completion` war in einer früheren Version des Containers bereits in *.bashrc* integriert. Falls dieses allerdings bei unterschiedlichen npm-Versionen unterschiedlich ausfällt, würde die Autovervollständigung nicht korrekt funktionieren. Durch das nachträgliche Hinzufügen des Ergebnisses mithilfe der `RUN`-Anweisung ist

sichergestellt, dass das Ergebnis von `npm completion` immer zur verwendeten npm-Version passt.

Zeile 17

In Zeile 17 wird das Startkommando des Containers auf die Bourne Again Shell festgelegt. Der verwendete Node.js-Container startet standardmäßig mit dem Kommando `node` ohne weitere Parameter. Da der softaware/webdev-Container allerdings zur interaktiven Entwicklung gedacht ist, wird dieses Standardverhalten überschrieben. In Abschnitt 4.6 wird gezeigt, dass sich dieses Kommando beim Ausführen des Containers überschreiben lässt, wodurch sich der Container auch in Skripte integrieren lässt.

In einer früheren Version des Containers wurde beim Start ein Skript aufgerufen, das überprüft, ob das aktuelle Verzeichnis ein Webprojekt ist. In diesem Fall wurde `npm install` aufgerufen, um alle Abhängigkeiten zu überprüfen und im Fall von fehlenden Paketen diese nachzuinstallieren. Da dieses Verhalten für den Benutzer unerwartet ist, allerdings bei jedem Start einige Sekunden kostet, wurde dieses Feature wieder entfernt.

4.3 Build-Prozess

Um ein Docker-Image auf Basis eines Dockerfiles zu erstellen, benötigt es lediglich das Kommando `docker build .`, welches durch den Punkt nach dem Dockerfile im aktuellen Verzeichnis sucht. Das softaware/webdev-Image ist allerdings im offiziellen Docker Hub[2] veröffentlicht, um es ohne weitere Konfiguration mit `docker container run` ausführen zu können.

Anstatt einer Beschreibung zum Erstellen des Images, wurde das in Quelltext 4.4 abgebildete Powershell-Skript erstellt. Mithilfe dessen kann das Image des Containers für existierende Node.js-Versionen automatisch erstellt und veröffentlicht werden.

Docker Hub bietet zwar die Möglichkeit eines automatisierten Builds, wobei das Image bei jeder Änderung automatisch in der Cloud erzeugt wird. Dieses Service wurde auch evaluiert, Docker Hub benötigt allerdings aufgrund fehlender Funktionen in der GitHub-API Lese- und Schreibrechte auf den gesamten GitHub-Account des verknüpften Repositorys. Leserechte für das Repository wären kein Problem, da der Quelltext des softaware/webdev-Containers ohnehin öffentlich ist. Schreibrechte auf den GitHub-Account der Firma softaware gmbh sind allerdings nicht mit den Unternehmensrichtlinien zu vereinbaren.

Build-Skript

Das in Quelltext 4.4 abgebildete Build-Skript existiert nicht nur wegen der vorhin beschriebenen Einschränkungen im Docker Hub, sondern auch aufgrund der in Abschnitt 4.4 erläuterten Besonderheiten. Da der Container Alpine Linux und Debian Linux unterstützt, müssen für jede Node.js-Version zwei Images erstellt werden. Im Folgenden werden die wichtigsten Funktionen des Build-Skriptes erklärt.

[2] `https://hub.docker.com/`

```
[CmdletBinding()]
param(
  [Parameter(Mandatory=$true, HelpMessage="Enter the specified node-version (e.g. 6.10.2) from https://nodejs.org/en/download/releases/.")]
  [String]$node_version,

  [Switch]$silent,

  [Switch]$publish
)

# ===== CONSTANTS =====
$REPOSITORYNAME = "softaware/webdev"
$IMAGEPATH = "./images/"
$DOCKERFILEROOTPATH = "$($IMAGEPATH)Dockerfile"
$DISTRIBUTIONS = @("alpine","debian")

function Generate-Image {
  param(
    [String]$node_version,
    [String]$distribution,
    [Boolean]$silent,
    [Boolean]$publish
  )

  $generatedfile = $IMAGEPATH + "Dockerfile.gen"

  $dockerfile = $DOCKERFILEROOTPATH + "." + $distribution
  $tag = $distribution + "-" + $node_version
  $imagename = $REPOSITORYNAME + ":" + $tag

  $template = Get-Content $dockerfile -Raw
  $generated = $template.Replace("{{ node_version }}", $node_version)
  $generated | Out-File -Encoding UTF8 $generatedfile

  $command = {docker image build (&{If($silent) {"--quiet"}}) --force-rm -f $generatedfile -t $imagename $IMAGEPATH}
  if (!$silent) {& $command} else {& $command | Out-Null}
  Remove-Item $generatedfile

  "$($imagename) created successfully"

```

```
41      if ($publish) {
42        Publish-Image $imagename
43      }
44  }
45
46  function Publish-Image {
47    param(
48      [String]$imagename
49    )
50    $command = {docker push $imagename}
51    if (!$silent) {& $command} else {& $command | Out-Null}
52
53    "$($imagename) published successfully"
54  }
55
56  function Generate-Images {
57    param(
58      [String]$node_version,
59      [String[]]$distributions,
60      [Boolean]$silent,
61      [Boolean]$publish
62    )
63    foreach ($distribution in $distributions) {
64      Generate-Image $node_version $distribution $silent $publish
65    }
66  }
67
68  Generate-Images $node_version $DISTRIBUTIONS $silent.IsPresent
      ↪ $publish.IsPresent
```

Quelltext 4.4: *create-image.ps1* (Build-Skript des Containers)

`create-image.ps1`

Das Build-Skript selbst nimmt als einzigen Parameter die Node.js-Version des zu verwendenden Basisimages entgegen. Zusätzlich können zwei Schalter gesetzt werden:

1. `-silent` reduziert die Ausgaben auf ein Minimum.
2. `-publish` veröffentlicht die erstellten Images auf `hub.docker.com`. Dazu muss der Verwender in der Docker CLI mit einem Benutzer eingeloggt sein, der Schreibrechte auf das softaware/webdev-Repository im Docker Hub besitzt.

`Generate-Images`

Die Funktion `Generate-Images` ruft für jede registrierte Linux-Distribution die Funktion `Generate-Image` auf. So können die unterstützten Linux-Distributionen in der Konstante `$DISTRIBUTIONS` zentral erfasst werden. Dadurch wird die Lesbarkeit des Skripts verbessert. Falls der Container in Zukunft um weitere Distributionen erweitert wird, können diese einfach in das Skript integriert werden.

`Generate-Image`

`Generate-Image` enthält die Logik zum Erstellen der Images. Auf Basis der Node.js-Version und der Linux-Distribution wird das Image erstellt und, falls gewünscht, danach veröffentlicht:

- In Zeile 27 wird der zu verwendende Dockerfile ausgewählt.
- Die Zeilen 28 und 29 setzen den Namen des Images, um die Namenskonvention `softaware/webdev:<Linux-Distribution>-<Node.js-Version>` zu erreichen.
- In den Zeilen 31 bis 33 wird ein temporärer Dockerfile erstellt, bei dem der Node.js-Versionsplatzhalter (vgl. Abschnitt 4.2) durch eine echte Versionsnummer ersetzt wird.
- Auf Basis dieses Dockerfiles wird in Zeile 36 das Image mit dem in Zeile 35 generierten Kommando erstellt.
- Schlussendlich wird in Zeile 37 der generierte Dockerfile wieder entfernt.

In Abb. 4.2 ist eine Beispielausgabe zu sehen, mit der die zwei Images für die Node.js-Version 8.0.0 erstellt und veröffentlicht wurden.

```
bemayr D: ❯❯❯ .\create-image.ps1 8.0.0 -silent -publish
softaware/webdev:alpine-8.0.0 created successfully
softaware/webdev:alpine-8.0.0 published successfully
softaware/webdev:debian-8.0.0 created successfully
softaware/webdev:debian-8.0.0 published successfully
bemayr D: ❯❯❯
```

Abbildung 4.2: Ausgabe des Build-Prozesses für Node.js 8.0.0

4.4 Alpine Linux vs. Debian Linux

Um die Imagegröße eines Docker-Containers möglichst gering zu halten, wird der Einsatz der dafür optimierten Linux-Distribution Alpine[3] empfohlen. Das Alpine-Basisimage hat

[3] `https://alpinelinux.org/`

lediglich eine Größe von fünf Megabyte und ist für die Ausführung im Hauptspeicher optimiert.

Auch vom offiziellen Node.js-Image[4] existiert eine Alpine-Version, die durch die geringe Größe von etwa 17 Megabyte besticht. In der Dokumentation des Node.js-Containers wird explizit darauf hingewiesen, dass Alpine nicht auf der verbreiteten `glibc`, sondern auf `musl libc` basiert.

Nach ausführlichen Tests wurde festgestellt, dass es bei der Verwendung des Alpine-Containers Probleme mit der Übersetzung von *Sass*[5] gibt. Sass ist eine Sprache, die CSS um die Funktionalität von Variablen und Funktionen erweitert. Das Problem trat bei der Verwendung des Komponentenframeworks *Kendo UI*[6] auf und ist in [Sim17] dokumentiert. Daher wird nun auch eine auf Debian Linux basierende Version des Containers angeboten, bei der diese Probleme nicht auftreten.

Empfohlen wird aufgrund der wesentlich geringeren Größe (~20MB anstatt ~250MB) allerdings die Alpine-Version des Containers.

Debian-Dockerfile

In Quelltext 4.5 ist der Dockerfile der Debian-Version des Containers dargestellt.

Der Hauptunterschied zum in Quelltext 4.2 dargestellten Alpine-Image liegt in Zeile 2. Vom Node.js-Docker-Container existiert sowohl eine Alpine- als auch eine Debian-Version, wobei für Alpine Linux das Präfix *alpine* und für die Debian-Version kein Präfix verwendet wird. Der zweite Unterschied ist, dass die Bourne Again Shell in Debian Linux bereits inkludiert ist, weshalb diese Installation im Image entfällt.

```
1  # specify the wanted node/npm version tag as listed in https://
       ↪ hub.docker.com/r/library/node/tags/
2  FROM node:{{ node_version }}
3  LABEL maintainer "bernhard.mayr@softaware.at"
4
5  # set path in a way that local npm-modules (e.g. ng-cli) can be
       ↪  called like executables
6  ENV PATH="./node_modules/.bin:${PATH}"
7
8  # set the working directory
9  WORKDIR "/usr/src/app"
10
11 # prepare bash
12 COPY ./files/.bashrc /root/
13 RUN npm completion >> /root/.bashrc
14
15 # start a new bash session
```

[4] `https://hub.docker.com/_/node/`
[5] `http://sass-lang.com/`
[6] `http://www.telerik.com/kendo-ui`

```
16 CMD ["/bin/bash"]
```

Quelltext 4.5: Debian-Dockerfile

4.5 Dokumentation

Dokumentation war neben der möglichst einfachen Benutzung die Hauptaufgabe bei der Erstellung des softaware/webdev-Containers. Da der Container das erste Open-Source-Projekt der Firma softaware gmbh ist, soll der Fokus auf Dokumentation bei der Verbreitung des Containers helfen, wodurch auch andere Unternehmen davon profitieren können. Auf folgende Besonderheiten wurde bei der Dokumentation des Projekts Wert gelegt:

GitHub Das gesamte Projekt wurde auf GitHub im Repository softawaregmbh/docker-webdev[7] veröffentlicht. Die Dokumentation und der Aufbau der Readme-Datei ist an diejenige des Projekts Cycle.js[8] angelehnt, da diese sehr übersichtlich ist.

Docker Hub Wie in Abschnitt 4.3 beschrieben, wurde der fertige Container im Docker Hub veröffentlicht. Aufgrund der in diesem Abschnitt erläuterten Probleme ist die Verknüpfung mit GitHub nicht möglich. Diese würde zusätzlich zum automatisierten Build-Prozess den Vorteil bringen, dass die Readme-Datei aus dem GitHub-Repository auch im Docker Hub angezeigt wird. Die aktuelle, suboptimale Lösung ist das manuelle Synchronisieren der Readme-Datei von GitHub in den Docker Hub.

MicroBadger MicroBadger[9] ist ein Onlinedienst, der Informationen zu Docker-Images übersichtlich aufbereitet. Die Verwendung des Dienstes ist für Open-Source-Projekte kostenlos. Zusätzlich werden sogenannte Badges generiert, die in die Dokumentation des Projektes integriert werden können. Beim softaware/webdev-Container werden diese verwendet, um die Imagegrößen der Containerversionen übersichtlich darzustellen. Abb. 4.3 zeigt diese Übersicht.

4.6 Beispiel: Auszug aus einem Kundenprojekt

Die in Abschnitt 4.1 empfohlene Anwendung des Containers ist die Versionierung durch ein Startskript. In Quelltext 4.6 ist ein solches Powershell-Skript dargestellt, das die Verwendung des Containers vereinfacht. Der Name dieses Skripts ist in der Firma softaware gmbh normalerweise *docker-webdev.ps1*, wodurch der Entwicklerworkflow über Projektgrenzen hinweg konsistent ist. Außerdem sind diese Skripte in allen Projekten im Wurzelverzeichnis mitversioniert.

[7] `https://github.com/softawaregmbh/docker-webdev`
[8] `https://github.com/cyclejs/cyclejs`
[9] `https://microbadger.com/`

Node[2]	Latest Version	Alpine	Debian
4	4.8.3	14.3MB 17 layers	248.3MB 21 layers
6	6.11.0	18.9MB 18 layers	250.1MB 21 layers
7	7.10.0	20.4MB 17 layers	251.5MB 21 layers
8	8.1.2	21.7MB 15 layers	253.1MB 19 layers

Abbildung 4.3: MicroBadger-Übersicht über die Container-Versionen

Bei dem Projekt des hier angeführten Skripts handelt es sich um ein Node.js-Webservice, das mit dem Web-Framework *koa*[10] entwickelt wurde.

```
1  param(
2      [String]$command
3  )
4
5  # --- constants --- #
6  $PORT = "3000"
7  $PORTMAPPING = "$($PORT):3000"
8  $IMAGE = "softaware/webdev:alpine-8.0.0"
9  $CONTAINERNAME = "koa-webdev"
10
11 # --- available commands --- #
12 $START = "start"
13 $SHELL = "shell"
14 $BACKGROUND = "background"
15 $BACKGROUND_STOP = "background-stop"
16 $INTERACTIVE = "interactive"
17
18 function Print-Usage() {
19     "--- USAGE ---"
20     Print-Command-Help $START "runs 'npm start': application
       ↪ available at http://localhost:$($PORT)"
21     Print-Command-Help $SHELL "launches a bash in the root
       ↪ folder with npm, node and yarn installed"
22     Print-Command-Help $BACKGROUND "runs 'npm start' in the
       ↪ background: application available at http://localhost
       ↪ :$($PORT)"
23     Print-Command-Help $BACKGROUND_STOP "stops and removes the
```

[10] http://koajs.com/

```
        ↪ previously run container"
24      Print-Command-Help $INTERACTIVE "launches a bash with Port
        ↪ $($PORT) mapped; run your npm commands afterwards"
25  }
26
27  function Print-Command-Help($command, $help) {
28      "$($command.PadRight(25))$($help)"
29  }
30
31  switch ($command)
32  {
33      $START { docker container run -it --rm -p $PORTMAPPING -v $
        ↪ {pwd}:/usr/src/app $IMAGE npm start }
34      $SHELL { docker container run -it --rm -v ${pwd}:/usr/src/
        ↪ app $IMAGE }
35      $BACKGROUND { docker container run -d --rm -p $PORTMAPPING
        ↪ -v ${pwd}:/usr/src/app --name $CONTAINERNAME $IMAGE
        ↪ npm start }
36      $BACKGROUND_STOP { docker container stop $CONTAINERNAME }
37      $INTERACTIVE { docker container run -it --rm -p
        ↪ $PORTMAPPING -v ${pwd}:/usr/src/app $IMAGE }
38      "" {
39          "[ERROR]: No command specified"
40          Print-Usage
41      }
42      default {
43          "[ERROR]: $($command) not specified"
44          Print-Usage
45      }
46  }
```

Quelltext 4.6: *docker-webdev.ps1* (Start-Skript des Containers)

Das Startskript nimmt ein Kommando entgegen, auf Basis dessen ab Zeile 31 entschieden wird, welches Docker-Kommando ausgeführt wird. Falls kein Kommando spezifiziert wird, wird eine Hilfe, wie in Abb. 4.4 dargestellt, ausgegeben.

Der typische Ablauf eines Softwareentwicklers sieht das Starten der Anwendung mit `.\docker-webdev.ps1 start` vor. Danach können in einer zweiten Powershell-Sitzung mithilfe von `.\docker-webdev.ps1 shell` npm-Kommandos ausgeführt werden. Dies könnte zum Beispiel die Installation eines Pakets mit `npm install <Paketname>` sein.

4.7 Probleme und Besonderheiten

Bei der Verwendung des Containers gilt es, folgende Besonderheiten zu beachten:

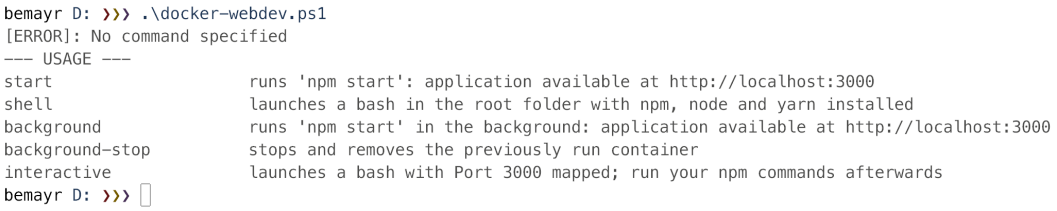

Abbildung 4.4: Hilfe zum Startskript des Containers in einer echten Applikation

- Wie bereits in Abschnitt 4.4 beschrieben, gibt es bei der Verwendung des Sass-Compilers unter Alpine Linux Probleme. Die Lösung dafür ist die Debian-Variante des Containers, bei der diese Probleme nicht auftreten.
- Vor der Verwendung des Containers muss das `node_modules`-Verzeichnis gelöscht werden. Diese Maßnahme ist notwendig, da es in Node.js native Pakete gibt, bei denen betriebssystemspezifische Varianten installiert werden. Nach dem Löschen des Verzeichnisses muss `npm install` im Container ausgeführt werden, damit die richtigen Pakete installiert werden.
- Bei der Verwendung des Containers in Kombination mit Entwicklungsumgebungen ist besondere Vorsicht geboten. Falls z. B. Visual Studio ein Webprojekt erkennt, aktualisiert es im Hintergrund automatisch die npm-Pakete. Da dies allerdings außerhalb des Containers geschieht, entstehen Inkonsistenzen im `node_modules`-Verzeichnis. Als Lösung wird die Deaktivierung dieser Funktion in der Entwicklungsumgebung vorgeschlagen.
- Viele Web-Frameworks werben mit *Hot Reloading*. Damit ist die automatische Aktualisierung des Browsers bei Änderungen am Quelltext gemeint. Bei der Verwendung des Containers muss dazu der Modus konfiguriert werden, der Dateiänderungen erkennt. Dieser muss auf Polling umgestellt werden, da es bei Docker ein Problem mit der Erkennung von Änderungen in Dateien gibt, wenn diese zwischen dem Host und dem Container geteilt sind [Doc17].
- Während der Entwicklung des Containers wurde die Möglichkeit getestet, das `node_modules`-Verzeichnis als eigenes Docker-Volume zu behandeln. Dadurch wäre die Performance beim Installieren der Pakete wesentlich besser. Allerdings benötigen die Entwicklungsumgebungen Zugriff auf dieses Verzeichnis, um die Autovervollständigung für TypeScript zu ermöglichen, da die Typinformationen direkt aus den Paketen ausgelesen werden.
- Die Verwendung des Containers ist optional. Falls sich einzelne Entwickler dagegen entscheiden, muss nur sichergestellt sein, dass sie die richtigen Versionen von Node.js und npm installiert haben. Dadurch werden heterogene Entwicklungsteams ermöglicht, bei denen nicht jeder Entwickler auf Docker angewiesen ist.

5 Resümee

Um einen kurzen Überblick über die Ergebnisse diser Arbeit zu geben, werden diese in diesem Kapitel zusammengefasst. Weiters werden die daraus gewonnenen Erkenntnisse dargestellt sowie ein Blick in die Zukunft gewagt.

5.1 Zusammenfassung

Die Anzahl und Komplexität der Werkzeuge in der Frontend-Webentwicklung im Jahr 2017 stellt ein großes Problem für Firmen da, die an vielen Projekten gleichzeitig arbeiten. Dabei sind nicht die neuen Werkzeuge problematisch, vielmehr ist es aufgrund von Versionsproblemen beinahe unmöglich, unterschiedlich alte Projekte auf einem Entwicklerrechner zu warten.

Durch die Verwendung von Docker kann ein Großteil der benötigten Werkzeuge mit dem Projekt mitversioniert werden, wodurch die Wartbarkeit von Anwendungen stark erhöht wird. Der softaware/webdev-Container bietet die Basis dazu und liefert mit der zugehörigen Dokumentation einen Überblick über die Möglichkeiten und Probleme beim Einsatz von Containern als Entwicklerwerkzeuge.

5.2 Erkenntnisse

Die in der Arbeit beschriebene Verbesserung der Webentwicklung wäre gar nicht notwendig, wenn die diversen Werkzeuge rückwärtskompatibel und korrekt versioniert wären. Da sich das allerdings im Nachhinein nicht mehr ändern lässt, löst ein Container in diesem Fall viele Probleme.

Durch die Erstellung von Startskripten kann Docker soweit abstrahiert werden, dass es auch von Entwicklern, die noch nichts mit Containertechnologien zu tun hatten, problemlos eingesetzt werden kann. Zusätzlich führt die Verwendung des Containers zu einem einheitlichen Werkzeugset und einer daraus resultierenden geringeren Hemmschwelle beim Projekteinstieg.

5.3 Ausblick

Dass sich Container sehr gut als Entwicklungswerkzeuge eignen, wird durch diesen Container sehr gut veranschaulicht. In einer idealen Welt gäbe es allerdings keine Versionskonflikte zwischen Node.js-Paketen und die Notwendigkeit für den softaware/webdev-Container wäre nicht gegeben.

Die beschriebenen Probleme könnten in den nächsten Monaten durchaus gelöst sein, doch das Aktualisieren von Legacy-Projekten ist in den meisten Fällen zu kostspielig. Genau in diesem Fall eignet sich der softaware/webdev-Container hervorragend und kann vor allem in den nächsten Jahren noch für zahlreiche Webprojekte relevant werden, die sich aktuell gerade in Entwicklung befinden.

6 Erfahrungen

Die Einführung einer neuen Technologie in ein Unternehmen ist immer eine aufregende Angelegenheit. Einerseits sind neue Technologien interessant und spannend, doch bei genauerer Betrachtung wird sofort skeptisch hinterfragt, ob das oftmals mühsame Einarbeiten in die neue Technologie auch wirklich den erwarteten Nutzen bringt. Besonders bei einer neuen Art der Technologie, wie Containertechnologien es sind, kann die anfängliche Euphorie sehr schnell der Frustration weichen. Wenn allerdings ein Problem existiert, dessen Lösung mithilfe dieser neuen Technologie realistisch erscheint, ist wesentlich weniger Überzeugungskraft notwendig.

Nach den gewonnenen Erfahrungen aus meiner theoretischen Bachelorarbeit war es für mich sehr interessant, im Praktikum wiederkehrende Probleme meiner Kollegen zu analysieren und einen völlig anderen Lösungsweg zu suchen. Solche Aufgaben gefallen mir besonders gut, da man dadurch meist lästige, unlösbar geglaubte Probleme löst, die allen Beteiligten viel Arbeit und Zeit sparen.

Nach gut drei Wochen im Praktikum war die Idee des softaware/webdev-Containers geboren, nachdem sich im Büro die Probleme mit npm und Node.js häuften.

Erst die genauere Recherche über die Werkzeuge der Webentwicklung hat mir das volle Ausmaß der Werkzeugvielfalt in der Webentwicklung vor Augen geführt. Aufgrund des Node.js-Basisimages war die erste Version des Containers sehr schnell fertig und konnte getestet werden. Der Einsatz in realen Projekten und das ständige Feedback der Kollegen führte zu einer stetigen Weiterentwicklung des Containers.

Dass der Quelltext des Containers veröffentlicht wurde, führte auch schon bei diversen Events zu interessanten Diskussionen und hat mir die Vor- und Nachteile der Open-Source-Entwicklung aufgezeigt. Auch das Veröffentlichen des Blogartikels [May17] im Blog der Firma softaware gmbh war eine sehr lehrreiche Erfahrung. Besonders als der Container beim Start eines neuen Webprojektes beinahe reibungslos in dieses integriert wurde, war es ein sehr gutes Gefühl, das Ergebnis meiner praktischen Bachelorarbeit in realen Projekten im Einsatz zu sehen.

Abbildungsverzeichnis

Quelltextverzeichnis

Literatur

[Dem16] Daniel Demmel. *Why You Should Stop Installing Your WebDev Environment Locally*. 20. Apr. 2016. URL: https://www.smashingmagazine.com/2016/04/stop-installing-your-webdev-environment-locally-with-docker/ (besucht am 25. 06. 2017).

[Doc17] Docker Inc. *Inotify on Shared Drives does not Work*. 23. Juni 2017. URL: https://docs.docker.com/docker-for-windows/troubleshoot/#inotify-on-shared-drives-does-not-work (besucht am 25. 06. 2017).

[Far15] H Andrew Farmer. *Guide to JavaScript Frontend Package Managers*. 14. Nov. 2015. URL: http://andrewhfarmer.com/javascript-frontend-package-managers/ (besucht am 23. 06. 2017).

[Fow14] Martin Fowler. *Microservices*. 25. März 2014. URL: https://www.martinfowler.com/articles/microservices.html (besucht am 20. 06. 2017).

[Har17] Rich Harris. *Webpack and Rollup: The Same but Different*. 6. Apr. 2017. URL: https://medium.com/webpack/webpack-and-rollup-the-same-but-different-a41ad427058c (besucht am 23. 06. 2017).

[Kir16] Marek Kirejczyk. *Hype Driven Development*. 23. Nov. 2016. URL: https://blog.daftcode.pl/hype-driven-development-3469fc2e9b22 (besucht am 21. 06. 2017).

[Lin15] Linux Foundation. *Node.js Foundation Combines Node.js and io.js into Single Codebase in New Release*. 14. Sep. 2015. URL: https://nodejs.org/en/blog/announcements/foundation-v4-announce/ (besucht am 21. 06. 2017).

[May17] Bernhard Mayr. *Consistent npm Development Environments using Docker*. 23. Mai 2017. URL: https://softaware.at/codeaware/2017/05/23/consistent-npm-development-environments-using-docker.html (besucht am 25. 06. 2017).

[Nem17] Gergely Nemeth. *Yarn vs npm - The State of Node.js Package Managers*. 10. Jan. 2017. URL: https://blog.risingstack.com/yarn-vs-npm-node-js-package-managers/ (besucht am 23. 06. 2017).

[Osm12] Addy Osmani. *Writing Modular JavaScript With AMD, CommonJS & ES Harmony*. 2012. URL: https://addyosmani.com/writing-modular-js/ (besucht am 23. 06. 2017).

[Pap16] John Papa. *Multiple Versions of Node with n*. 21. Mai 2016. URL: https://johnpapa.net/multiple-versions-of-node-with-n/ (besucht am 22. 06. 2017).

[Pey16] Sebastián Peyrott. *JavaScript Module Systems Showdown: CommonJS vs AMD vs ES2015*. 15. März 2016. URL: `https://auth0.com/blog/javascript-module-systems-showdown/` (besucht am 23. 06. 2017).

[Sim17] Richard Simko. *Segfault when Compilation Raises Errors*. 13. Jan. 2017. URL: `https://github.com/sass/node-sass/issues/1858` (besucht am 24. 06. 2017).

[typ12] typeoneerror (stackoverflow.com-Benutzername). *How to Use Packages Installed Locally in node_modules?* 13. März 2012. URL: `https://stackoverflow.com/questions/9679932/how-to-use-package-installed-locally-in-node-modules#comment33532258_9683472` (besucht am 22. 06. 2017).