Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Bachelorarbeit

Jonny Niebuhr

Konzeption und Umsetzung einer IDE in einem Docker-Container

Fakultät Technik und Informatik
Studiendepartment Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Jonny Niebuhr

Konzeption und Umsetzung einer IDE in einem Docker-Container

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. Stefan Sarstedt
Zweitgutachter: Prof. Dr. Ulrike Steffens

Eingereicht am: 09. Juni 2017

Jonny Niebuhr

Thema der Arbeit
Konzeption und Umsetzung einer IDE in einem Docker-Container

Stichworte
Docker, VM, Container, Benutzeroberfläche, Persistenz, Hostcomputer

Kurzzusammenfassung
Das Ziel dieser Arbeit war es ein Konzept zu entwickeln und umzusetzen, das die Bedienung der grafischen Benutzeroberfläche einer Software, die in einen Docker Container betrieben wird, gewährleistet. Dazu wurden zunächst mögliche Lösungsansätze betrachtet und näher beschrieben. Anschließend wurde ein Prototyp erstellt, der den Anforderungen entspricht. In diesem Zusammenhang wurde die Vorgehensweise dargelegt und erläutert. Am Schluss dieser Arbeit wurden alle Ansätze noch einmal kurzgefasst und bewertet.
Jonny Niebuhr

Title of the paper
Conception and implementation of an IDE in a docker container

Keywords
Docker, VM, container, user-interface, persistence, host-computer

Abstract
The goal of this work was to develop and implement a concept, which allows to run software that is operated in a docker container via a graphical user interface. For this purpose, possible approaches were considered and described in detail. As a result, the approach was shown and explained. Subsequently, a prototype was created which meets the requirements. At the end of this thesis, the approaches are briefly reviewed and evaluated.

Inhaltsverzeichnis

1 Einleitung

1.1 Vorwort

Integrierte Entwicklungsumgebungen (Abkürzung IDE, vom Englischen integrated development environment) sind komplexe Werkzeuge. Sie bieten vielseitige Konfigurationsmöglichkteiten und können für Projekte der Softwareentwicklung verschiedenster Art eingesetzt werden. Oft benötigen IDEs bestimmte Software, wie zum Beispiel Sprachbibliotheken, um Aufgaben ausführen zu können. Dies führt dazu, dass auf dem PC Abhängigkeiten zwischen der Software entstehen.
Innerhalb einer Arbeitsgruppe, in der mehrere Personen mit der gleichen IDE Konfiguration und Software-Abhängikeiten tätig sind, werden einzelne Prozessschritte der Konfiguration und Installation von Software mehrmals von verschiedenen Personen ausgeführt. Dies widerspricht dem Grundsatz *don't repeat yourself* (kurz: DRY) und ist mit einem größeren zeitlichen Aufwand verbunden. Darüber hinaus könnte die mehrfache Bearbeitung durch unterschiedliche Personen aufgrund des Mangels an Automatisierung eine Fehlerquelle darstellen. Folglich ist es das Ziel dieser Arbeit, ein Konzept vorzustellen und umzusetzen, bei dem einzelne Arbeitsschritte soweit wie möglich automatisiert werden, sodass die genannten Negativeffekte entfallen. Docker weist alle Eigenschaften auf um dieses Ziel zu ermöglichen. Es verspricht, wie bei virtuellen Maschinen, Plattformunabhängigkeit, allerdings ohne Kompromisse bei Performance und Ressourcenverbrauch einzugehen. Des Weiteren wird eine Software inklusive aller Konfigurationen und Abhängigkeiten in einem Docker Image gebündelt, die auf jeder Docker unterstützenden Plattform betrieben werden kann.

1.2 Aufbau der Arbeit

Das erste Kapitel erläutert kurz das Ziel dieser Arbeit. In Kapitel 2 wird auf die technischen Grundlagen eingegangen. In diesem Rahmen wird zunächst die Technik der virtuellen Maschinen erläutert. Aufbauend darauf wird anschließend die allgemeine Funktionsweise und Struktur von Docker thematisiert.
Kapitel 3 beinhaltet die Analyse, die bei der Umsetzung des Docker Images berücksichtigt werden muss. Zudem umfasst es Erklärungen dazu, welche Software vorausgesetzt wurde. Anschließend werden mögliche Lösungansätze veranschaulicht.
In Kapitel 4 wird detailliert illustriert, wie dies umgesetzt wird und wie sich der Aufbau gestaltet.
Im letzten Kapitel werden die Ergebnisse und die Erkenntnisse der Arbeit zusammengefasst.

1.3 Ziel

Das Ziel dieser Arbeit ist es, einen Docker Container mit einer vollständig konfigurierten IDE zu erstellen. Darüber hinaus soll ermöglicht werden, dass diese so angewendet werden kann, als wäre sie lokal auf dem PC installiert.
Dadurch wird die Konfiguration einer IDE und die Installation dessen Abhängigkeiten zentralisiert und automatisiert. In dieser Arbeit wird eine Machbarkeitsanalyse eines solchen Docker Images sowie dessen Umsetzung angestrebt.
Sobald der User den Container startet soll sich die Benutzeroberfläche der IDE öffnen, als wäre diese Lokal installiert. Des Weiteren soll die Benutzung eines persistenten Speichers gewährleistet sein. Zudem müssen auf dem PC keine weiteren Programme als der Docker Deamon installiert werden. Allein dieser Docker Deamon wird zum Starten und Verwalten von Docker Images und Containern ausreichend sein. Dies hat den positiven Effekt, dass das System selber nicht mit Software belastet wird. Hervorzuheben ist ebenfalls, dass eine Deinstallation seitens des Benutzers nicht erfolgen muss, falls die Software nicht mehr benutzt werden möchte.

2 Grundlagen

In diesem Kapitel werden die technischen Grundlagen dieser Arbeit dargelegt. In diesem Zusammenhang werden zum einem die allgemeinen Funktionen und der Nutzen der virtuellen Maschinen veranschaulicht. Zum anderen werden die Grundlagen von Docker und dessen Unterschiede zur klassischen Virtualisierung skizziert. Zuletzt folgt eine kurze Beschreibung von IDEs sowie deren möglichen Abhängigkeiten.

2.1 Virtuelle Maschinen

2.1.1 Allgemein

Eine virtuelle Maschine (kurz VM) ist die Virtualisierung einer physischen Maschine, was die Möglichkeit schafft einen simulierten PC in einer abgeschotteten Umgebung auf einem realen PC laufen zu lassen. Die Virtualisierung wird durch Software (Hypervisor) realisiert, die einen Teil der Ressourcen, wie Grafikkarte, CPU, Speicher, Netzwerkkarte und Festplatte des realen PCs übernimmt und soweit aufbereitet, dass diese von einem Betriebssystem angewendet werden kann. Auch das BIOS wird virtuell simuliert. Zusätzlich können auch andere Hardware Komponenten virtuell zu Verfügung gestellt werden, beispielsweise optische Laufwerke.
Aus der Sicht des Betriebssystems ist es nicht möglich zu sagen, ob es auf einer virtuellen oder physikalischen Maschine läuft. Realisiert wird das durch die Hardware Abstraktion, die bewirkt, dass sich ein Betriebssystem in beiden Fällen identisch verhält. Man unterscheidet zwischen einem Gastbetriebssystem (betrieben auf einer virtuellen Maschine) und einem Hostsystem (betrieben auf einer physikalischen Maschine) (vgl. Zimmer (2012)).

2.1.2 Hypervisor

Die Software, die mit der die Abstrahierung befasst, nennt sich Hypervisor oder auch *Virtual Machine Monitor* (kurz VMM). Der Hypervisor leitet die Maschinenbefehle des Gastsystems an das Hostsystem weiter und übersetzt diese gegebenenfalls. Er teilt die Ressourcen des Hostsystem ein und ordnet sie dem Gastsystem zu. Durch einen Hypervisor ist es möglich, mehrere grundsätzlich unterschiedliche Betriebssysteme auf einer Maschine zu installieren und parallel zu betreiben. So ist es möglich auf einer Linux Distributionen eine VM zu erstellen, auf der ein Microsoft Windows System installiert ist. Man klassifiziert zwei Typen von Hypervisor, die sich in ihrer Architektur unterscheiden.

Hypervisor Typ 1

Der Typ1 Hypervisor ist der performantere von beiden Typen. Dies wird dadurch erreicht, dass dem Hypervisor kein Betriebssystem unterliegt. Somit läuft es direkt auf der Hardware und teilt die Ressourcen den Gastsystemen zu, wie in der Abbildung 2.1 auf Seite 4 dargestellt. Der Hypervisor muss dafür mit den entsprechenden Treibern für die Hardware ausgestattet werden.

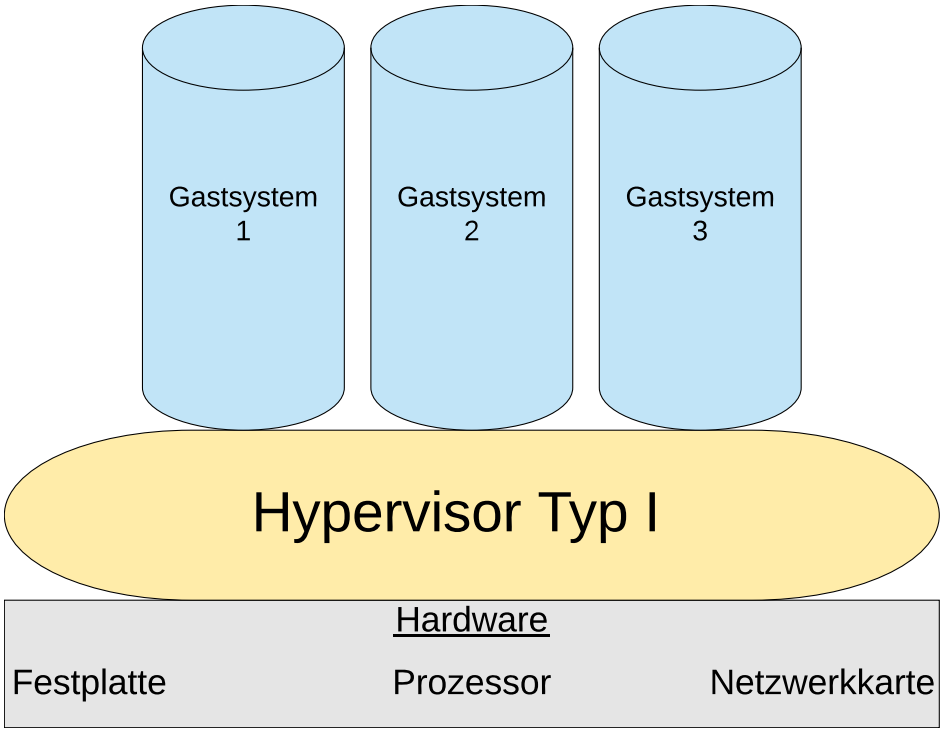


Abbildung 2.1: Hypervisor Typ 1

Hypervisor Typ 2

Bei dem Typ 2 Hypervisor handelt es sich um eine Software, die auf ein bereits bestehendes Betriebssystem installiert wird. Er kann dadurch die Treiber des Betriebssystem benutzen, um die Hardware anzusprechen und zu unterteilen (siehe Abbildung 2.2 auf Seite 5). Die unerwünschte Folge daraus ist, dass Performance verloren geht da neben der Virtualisierung auch das Host Betriebssystem sowie gegebenenfalls andere Programme Ressourcen in Anspruch nehmen.

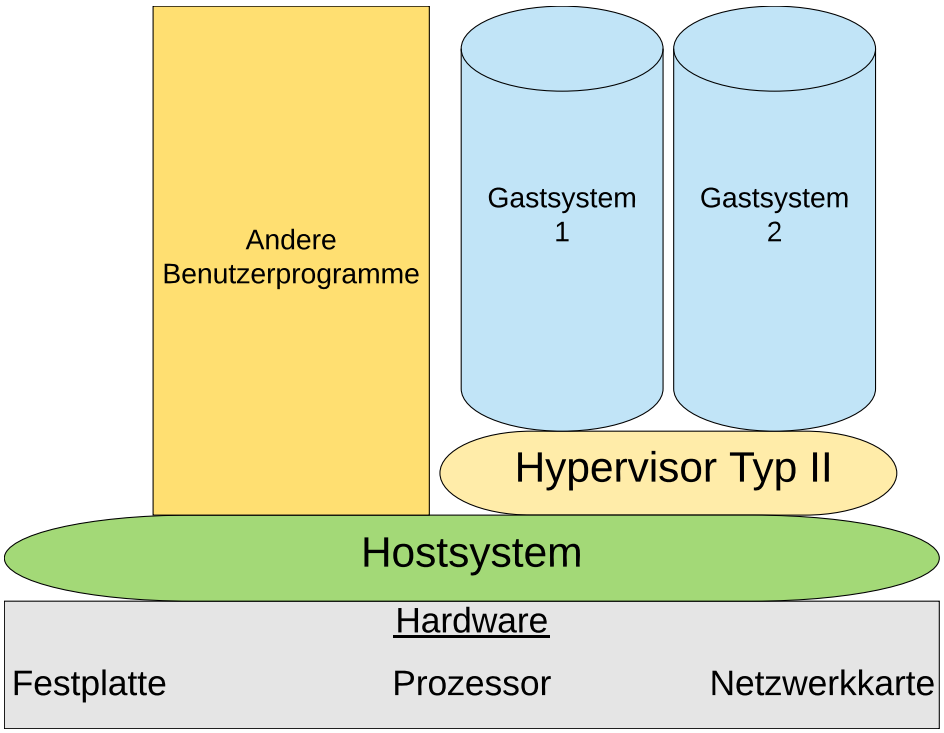


Abbildung 2.2: Hypervisor Typ 2

2.2 Docker

2.2.1 Grundgedanke und Aufbau

Grundgedanke

Im Transportwesen sind viele verschiedene Güter in allen möglichen Formen vorhanden. Es existieren feste Güter in Kisten, flüssige in Tonnen oder Kanistern, granulare in Säcken und viele mehr. Jedes dieser Elemente bedarf eine andere Art der Aufbewahrung und Verstauung. Diese Umständlichkeit wurde mit der Einführung von Containern gelöst. Container besitzen genormte Maße und sind von der äußerlichen Handhabung identisch. Es ist unerheblich, was der Container beinhaltet, um ihn zu transportieren.
Dieses Prinzip wurde von Docker mit den Docker-Containern in der IT Welt umgesetzt. Ein Docker Container ist ein in sich abgeschlossenes System, vergleichbar mit einer vollwärtig simulierten Maschine. Es umfasst ein Betriebssystem und ist in der Lage Applikationen zu betreiben und zu verwalten. Weiterhin enthält es einheitliche Schnittstellen, um den Docker Container anzusprechen.

Aufbau

Docker basiert auf Techniken, die im Linux Kernel vorhanden sind. Der Kernel ist der zentrale Bestandteil eines Betriebssystems und stellt Schnittstellen zur Hardware zur Verfügung, auf die das Betriebssystem selber zurückgreift. Docker macht sich die Funktionsweisen des Kernels zu Nutzen. Indem abgeschottete Prozesse direkt auf dem Hostsystem laufen werden die Container realisiert (Abbildung 2.5 auf Seite 11). Docker besteht aus drei elementaren Komponenten, das ist die Dockerfile, das Docker Image und der Docker Container.
Die Funktionalitäten werden vom Docker Daemon gesteuert. Der Daemon muss als Prozess des Benutzers root gestartet werde,n um seine Funktionen bereitstellen zu können. Als Dateisystem nutzt Docker eines der Unification File Systems. Dadurch können mehrere Dateisysteme übereinander geschichtet und für den Container als eines dargestellt werden (Abbildung 2.4, Seite 9) (vgl. Hykes (2014)).

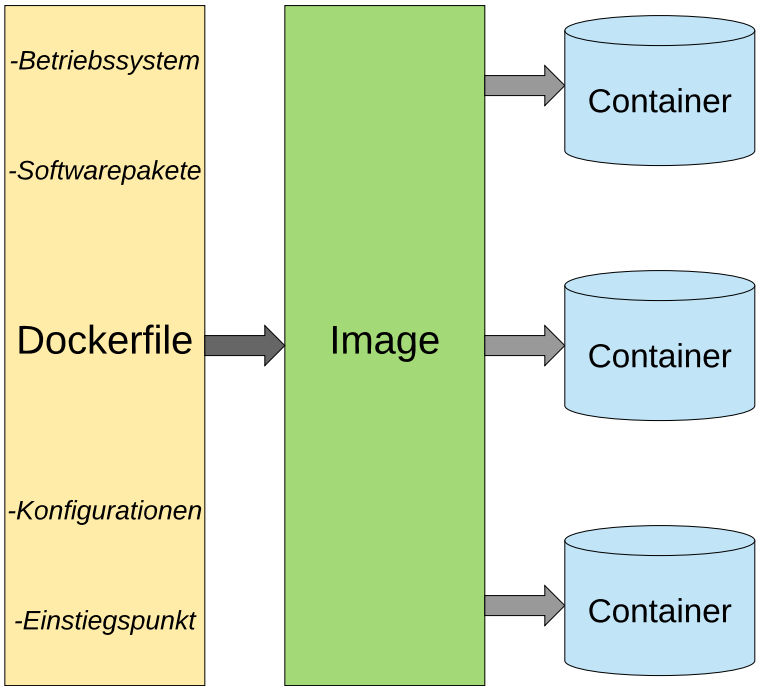


Abbildung 2.3: Docker Aufbau

2.2.2 Dockerfile

Am Anfang steht die Dockerfile. Sie enthält alle Informationen in Form einer Liste von Instruktionen, die sequenziell abgearbeitet werden. Die Dockerfile umfasst dasjenige Betriebssystem, das dem Image zugrunde liegt, sowie die zu installierende Software. Des Weiteren können grundsätzliche Konfigurationen und zusätzlich Befehle definiert werden, die beim Start des Containers Anwendung finden.
Eine Dockerfile setzt bestimmte Befehle voraus, damit daraus ein Image entstehen kann. Im Folgenden werden bestimmte Befehle kurz erläutert, welche für diese Arbeit eine Rolle spielen (vgl. Docker-Inc. (2017)).

FROM

An erster Stelle steht der FROM Befehl. Er gibt an auf welches Basisimage das resultierende Image aufbauen soll. Hier kann ein Betriebsystem stehen oder auch ein Image, welches von einer anderen Quelle erstellt wurde. Viele Images besitzen sogenannte *Tags* mit denen sie sich unterteilen lassen, beispielsweise in Form von Versionsnummern.

MAINTAINER

Hier steht der Autor der Dockerfile.

RUN

Hier werden Parameter übergeben, die als Shell Befehl ausgeführt werden. Es wird aufgelistet, welche Pakete installiert und welche Konfigurationen gesetzt werden sollen. Jeder RUN Befehl erzeugt eine neue Schicht im Image und verbraucht dadurch Speicherkapazität. Um ressourcensparend vorzugehen, sollten möglichst viele Shell Befehle hinter einem Run Befehl stehen.

CMD/ENTRYPOINT

CMD bietet einem Container eine Standardaktion die beim Start ausgeführt wird. Diese kann beim Aufruf auch überschrieben werden. Es kann nur einen CMD Befehl pro Docker Image existieren. Für den Fall, dass mehrere definiert werden, ist nur der zuletzt ausgeführte von Relevanz.

ENTRYPOINT verhält sich ähnlich wie CMD und stellt einen Einstiegspunkt beim Start eines Containers dar. Jedoch kann ENTRYPOINT nicht von CMD beim Aufruf überschrieben werden, was hingegen bei CMD möglich ist. Um einen Container mit einem anderen Befehl zu starten, muss der ENTRYPOINT mit dem ENTRYPOINT Befehl überschrieben werden. Der ENTRYPOINT ist, wenn nicht anders angeben, mit */bin/sh -c* gesetzt. Wenn CMD definiert wird, wird es dem ENTRYPOINT angefügt.

EXPOSE

Dieser Befehl dient dazu, die Ports des Containers freizugeben.

ADD/COPY

ADD und COPY sind zwei ähnliche Befehle, die eine Datei oder ein Verzeichnis von dem Hostsystem in das Dateisystem des Images einfügen. ADD ist dabei mächtiger als COPY.

WORKDIR

Jeder RUN, CMD, ADD und COPY Befehl geht zunächst einmal von dem root Verzeichnis aus. Sind Änderungen erwünscht so kommt der WORKDIR Befehl zum Einsatz. Dieser setzt das neue Referenzverzeichnis für alle folgenden Befehle beziehungsweise bis zum nächsten Aufruf von WORKDIR.

2.2.3 Docker Image

Das Docker Image wird mit den Informationen aus dem Dockerfile gebaut. Es ist eine Art Schnappschuss eines Systems. Images sind intern in Schichten aufgebaut, wobei jede Schicht

ein eigenständiges Image sein kann. So ist zum Beispiel das Betriebssystem ein Image (Basisschicht oder in englisch base layer), das bei dem Bau eines weiteren Images geladen wird. Wird nun ein weiteres Image erstellt, das auf der gleichen Basisschicht basiert, muss dieses nicht neu geladen werden. Es ist ausreichend, dass auf die Basisschicht verwiesen wird. Dies ist besonders bei mehreren auf gleichen Schichten basierenden Images ressourcensparend. Zusammenfassend ist das Image jenes Element von Docker, das portabel ist und verbreitet werden kann.

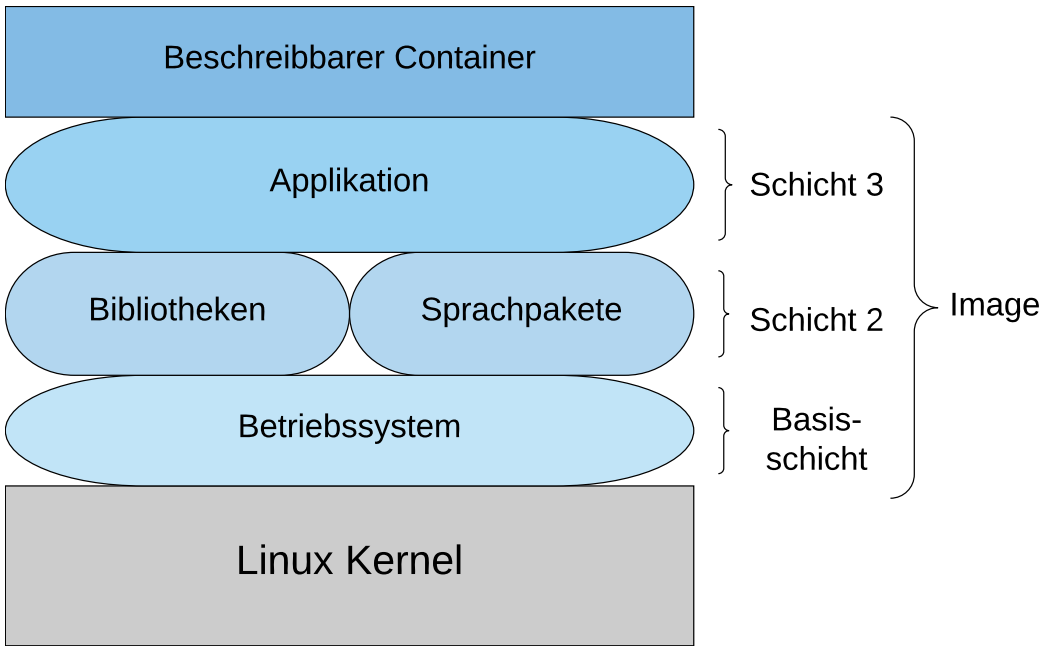


Abbildung 2.4: Image Schichten

2.2.4 Docker Container

Der Docker Container ist die laufende Instanz eines Docker Images. Wird ein Container geschlossen und erneut gestartet, so wird er jedes mal wieder aus dem Image erstellt. Möchte man dauerhaft Änderungen am Container vornehmen, so müssen sie mittels eines *commit* Befehls gespeichert werden. Es wird dann eine neue Schicht erstellt, die auf das Image gelegt wird und somit ein neues Image darstellt. Es ist möglich aus einem Image mehrere Container zu starten, die parallel laufen.

2.2.5 Docker-Registry

Docker Hub

Docker Hub dient als Onlinedienst dazu eine Registry zu pflegen. Diese kann mit Docker Images und Repository verwendet werden. Hinsichtlich der Registry kann zwischen einem öffentlichen und einem privaten Part differenziert werden. Erst genannter dient dazu, dass die Benutzer ihre eigenen Images hochladen können, sodass andere Nutzer diese öffentlich zugänglichen Images einsehen und verwenden können. Auch Linux Distributoren stellen offizielle Images zur Verfügung. Im Gegensatz dazu sind die Images im privaten Teil nicht öffentlich einsehbar. Vielmehr können hier Images hochgeladen und für einen ausgewählten Benutzerkreis verteilt werden (vgl. Wiki03. (2017)).

Versionsverwaltung

Die in Docker integrierte Versionsverwaltung ermöglicht es ein Image zu speichern, das den aktuellen Stand des Containers wiedergibt. Dieses Image kann in diesem Zusammenhang auf das Docker Hub geladen werden. Weiterhin ermöglicht die Versionsverwaltung, dass die Unterschiede zwischen dem aktuellen Zustand des Containers und dem ursprünglichen Image dargestellt werden. Letztlich kann sie realisieren, dass die Historie eines Images angezeigt wird (vgl. Wiki03. (2017)).

2.2.6 Unterschiede zur virtuellen Maschine

Der Unterschied zu den vorherig genannten Arten der Virtualisierung ist der, dass auf eine zusätzliche Software, nämlich den Hypervisor verzichtet werden kann. Dies ist damit begründet, dass die Aufgabe direkt vom Kernel ausgeführt wird, wie in der Abbildung 2.5 auf Seite 11 dargestellt. Es wird lediglich ein Programm benötigt, um die Container zu verwalten und zu steuern. Dadurch ist es nicht mehr nötig Hardware-Komponenten zu virtualisieren, was zu

einem Ressourcen einspart und zum anderen deutlich weniger Latenzen aufweist. Daraus resultiert, dass Docker Container sehr flexibel einsetzbar sind.

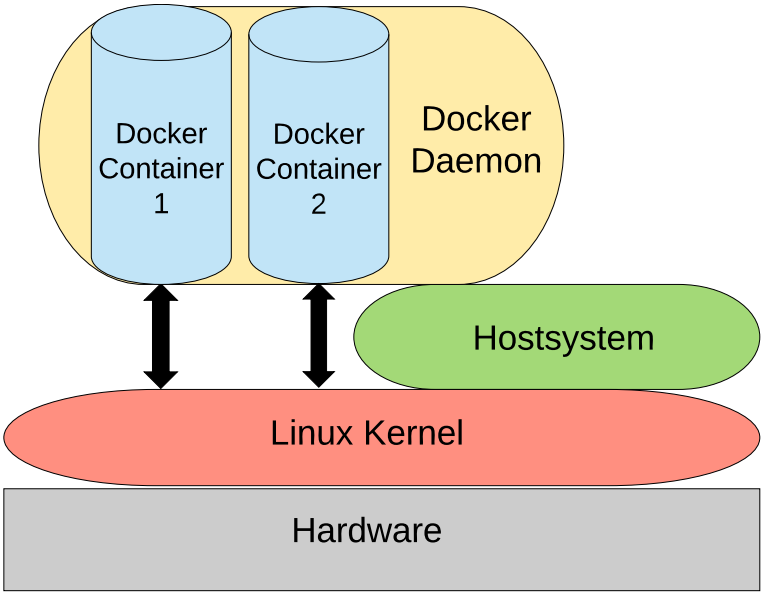


Abbildung 2.5: Docker

2.3 IDE

Eine IDE (Abkürzung für englisch integrated development environment, auf deutsch Integrierte Entwicklungsumgebungen) ist ein Werkzeug, welches für das Entwickeln von Software angewendet wird. Oft stellt die IDE selber eine Sammlung an Programmen dar, die um weitere Programme oder Bibliotheken erweitert werden kann, je nach Anforderungsprofil des jeweiligen Entwicklers (vgl. Wiki06 (2017)).

2.4 X-Server

Der X-Server (auch X Version 11, X11, X) ermöglicht es, die Fenster auf dem Großteil der unixoiden Betriebssystemen und OpenVMS darzustellen. Es hält eine Reihe von standartisierten Baukästen sowie das Protkoll bereit, um dies umzusetzen. In den gängigen Betriebssysten ist solch ein X-Server implementiert. Dem Servernamen entsprechend werden die Anwendungsprogramme als X-Clients bezeichnet (vgl. WikiU01 (2017)).

3 Analyse

Es soll ein Image erstellt werden, das alle erforderlichen Faktoren abdeckt, um in einer bestimmten Entwicklungsumgebung arbeiten zu können. Es besteht der Anspruch, dass neben dem Image keine weiteren Programme und Konfiguration zum Entwickeln notwendig sind. Alle nötigen Sprachpakete und Bibliotheken sollen im Image eingebunden sein. Dabei soll der Container sich so starten lassen, dass sich lediglich die Benutzeroberfläche der IDE öffnet und bedienen lässt. In diesem Zusammenhang soll es möglich sein, dass sich Projekte persistent außerhalb des Containers speichern lassen und sich Projekte in dem Container öffnen lassen, die unabhängig von diesem erstellt wurden. Des Weiteren soll gewährleistet sein, dass das Image innerhalb einer Arbeitsgruppe verteilt beziehungsweise zur Verfügung gestellt werden kann.

3.1 Software

IDE

Die im Image enthaltene IDE heißt Rider und ist von dem Software-Unternehmen JetBrains entwickelt worden. "JetBrains Rider is a new .NET IDE based on the IntelliJ platform and ReSharper "JetBrains (2017). Mit der IDE lassen sich Projekte basierend auf das .NET sowie das Platform übergreifende .NET Core Framework von Microsoft entwickeln.

.NET/Core Framework

Das Framework ist ein Teil von Microsofts Software-Plattform .NET und dient der Entwicklung und Ausführung von Anwendungsprogrammen. Das .NET Framework besteht aus einer Laufzeitumgebung, in der die Programme ausgeführt werden, sowie einer Sammlung von Klassenbibliotheken, Programmierschnittstellen und Dienstprogrammen (vgl. Wiki02 (2017)).

3.2 Herausforderung

Docker wird in der Regel dazu verwendet Anwendungen mithilfe von Betriebssystemvirtualisierung in Containern zu isolieren. Diese Anwendungen sind in den meisten Fällen keine Desktop-Applikationen sondern Anwendungen die keine grafische Benutzeroberfläche aufweisen. Jeder Container hat standardisierte Netzwerkschnittstellen und als Basis eine Linux-Distribution, womit auch die Bedienung durch Shell Befehle oder auch Skripte möglich ist. Durch die kurzen Latenzen und die generelle Sparsamkeit der Container, können so zum Beispiel Datenbanken ohne größeren Aufwand ausgetauscht werden. Auf diese Weise lassen sich Anwendungen und Services bequemer und übersichtlicher verwalten und steuern. Anstatt auf eine kohärente Architektur zurückzugreifen, die mehrere Services anbietet, läuft jeder Service in einem für sich individuellen Container.
In dieser Arbeit sollen diese Vorteile genutzt werden, um diese für Desktop-Applikationen gebräuchlich zu machen. Docker Container wurden nicht dazu entwickelt eine grafische Schnittstelle zu besitzen oder Daten permanent zu speichern. Die Herausforderung besteht darin, die Docker Technologie und Prinzipien anzuwenden, wobei der Service eines Containers eine voll konfigurierte IDE ist.

3.3 Mögliche Ansätze

Es gibt mehrere Ansätze um die Probleme der persistenten Speicherung und der Übertragung der Benutzerfläche zu lösen. Auf diese wird nun detaillierter eingegangen.

3.3.1 Xvfb mit VNC

X Window Virtual Framebuffer (kurz Xvfb) ist ein X-Server, der bildschirmlos arbeiten kann. Stattdessen wird ein virtueller Framebuffer angewendet. Das bedeutet, dass die grafischen Darstellungen nicht direkt auf dem Monitor erstellt werden, sondern sie existieren virtuell im Arbeitsspeicher. Für die Funktionen des Clients in dem X-Server Model haben diese Gegebenheiten keinen Einfluss (vgl. Wiki05 (2017)).
VNC (abgekürzt für Virtual Network Computing) steht für eine Software, die eine Interaktion zwischen einem als Server fungierenden externen Rechner und einem lokalen Rechner (Client) ermöglicht. Die Software zeigt den Bildschirminhalt auf dem lokalen Rechner an und Tastatur und Mausbewegungen des lokalen Rechners werden gleichzeitig an den Server transportiert. VNC funktioniert plattformunabhängig, da es mit dem Remote Framebuffer Protocol arbeitet (vgl. Wiki04. (2017)).
Anstatt VNC nun auf einem nicht lokalen Computer zu installieren, ist es auch möglich VNC in einem Docker Container, auf welchem auch Xvfb installiert ist, einzusetzen. Die Funktionsweise von VNC ändert sich nicht und der Docker Container übernimmt die Rolle des VNC-Servers. Er überträgt seinen Bildschirminhalt auf den Host Rechner (VNC-Client), welcher nun per Tastatur und Maus den Container bedienen kann. Damit der Host über VNC mit dem Container kommunizieren kann, muss dieser einen bestimmten Port freigeben.
Somit generiert der Container den virtuellen Bildschirminhalt und stellt diesen dem Host über VNC zur Verfügung. Im Gegenzug kann der Host über VNC Tastatur- und Mauseingaben an den Container weitergeben.

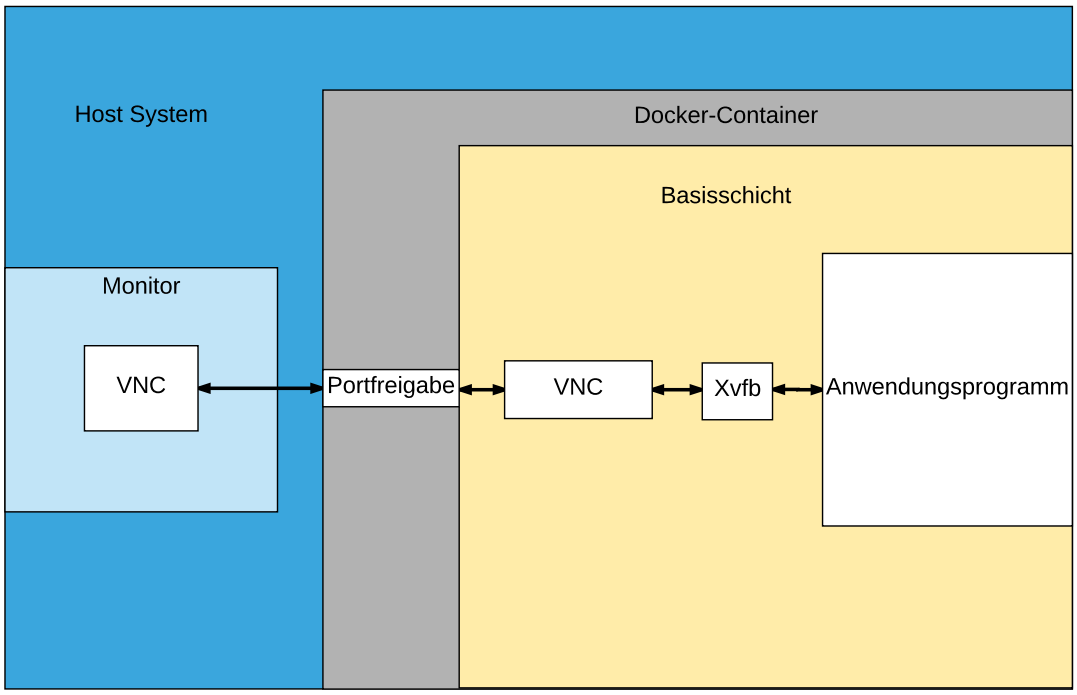


Abbildung 3.1: VNC

3.3.2 Docker-Volume-Plugins

Es existiert mittlerweile eine Vielzahl von Lösungen für das dauerhafte Speichern von Daten aus einem Docker Container. Um ein größeres Spektrum an Funktionalitäten von Docker zu erreichen, wurden die Volume-Plugins eingeführt. Es wurde eine API entwickelt, womit die Netzwerk- und Volume-Möglichkeiten erweitert werden können (vgl. Docker-Inc.. (2017)). Dadurch ist es zudem möglich nicht nur lokal eine persistente Speicherung zu erstellen, sondern auch extern Daten aus einem Container zu speichern und zu verwalten.

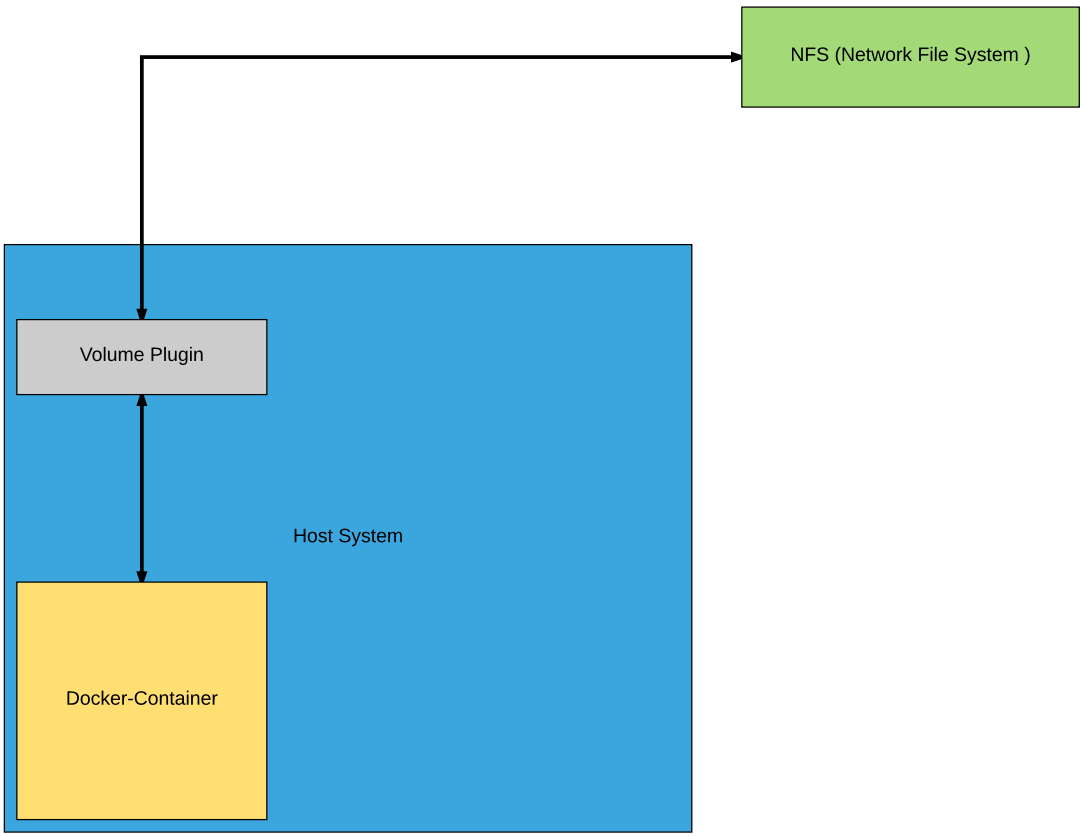


Abbildung 3.2: Docker-Volume-Plugins mit NFS

3.3.3 Basisschicht

Die Größe des Images dieser Arbeit wird überwiegend von der IDE und den geforderten Sprachbibliotheken beeinflusst, wie in Abbildung 4.9 auf Seite 37 visualisiert. Deswegen wurde bei der Umsetzung des Konzepts ein Base Image genommen, welches den geringsten Mehraufwand bei der Umsetzung hat. Es wurde als Basischicht das Docker Base-Image Ubuntu 16.04 verwendet. Es gibt jedoch für Docker mittlerweile eine Menge an Basisschichten, basierend auf Linux, die je nach Anwendungsfall entworfen wurden. Das Ziel eines Images sollte es sein, eine möglichst kleine Größe zu haben, da es normalerweise aus einer nicht lokalen Quelle und damit über das Netz geladen wird. Natürlich bedeutet ein schmaleres Base-Image auch, dass es weniger Pakete und Bibliotheken beinhaltet, die gegebenenfalls manuell nachgepflegt werden müssen.

3.4 Möglichkeiten mit Windows

Docker wurde ursprünglich unter Linux entwickelt und basiert auf nativen Linuxfeatures, wie im Abschnitt 2.2.1 auf Seite 6 beschrieben. Mittlerweile arbeiten jedoch die Entwickler von Docker zusammen mit Microsoft um die Container Technologie auch unter Windows zu integrieren. Gegenwärtig steht die Container-Technologie für die Betriebsysteme Windows Server 2016 und Windows 10 zur Verfügung (vgl. Thomas Joos (2017)).
Im weiteren Abschnitt wird auf mögliche Wege eingegangen, um die Anforderungen auch unter Windows zu realisieren.

3.4.1 Weiterleitung der Benutzeroberfläche

Cygwin/x

Software, die für Systeme wie zum Beispiel Unix, Linux oder BSD entworfen wurden, lassen sich mit Cygwin auf Windowsplattformen lauffähig machen. Umgesetzt wird das durch einen Adapter, der die Unix-API bereitstellt. Dadurch ist es möglich, dass Unix-Programme unter Windows übersetzt werden können.
Mit Cygwin/X existiert auch eine Portierung des X.Org-Servers auf die Cygwin-Umgebung. Dadurch ist es möglich unter Microsoft Windows einen kompletten X-Server bereit zu stellen. Durch den portierten X-Server ist es möglich, entweder UNIX/Linux-Programme, die für Windows kompiliert wurden, lokal auf dem Windows-Rechner auszuführen, oder aber Programme, die auf einem Unix- oder Linuxrechner ausgeführt werden, auf Windows darzustellen. Außerdem gibt es einem die Möglichkeit sich, ausgehend von dem Windows-Rechner, auf einem Unix-Rechner einzuloggen (vgl. Wiki04 (2017)).
Das ermöglicht ein, ähnliches Vorgehen wie in dem Start Skript 4.10 auf Seite 28. Wie im folgenden Skript dargestellt wird auch hier der Socket des X-Servers in den Docker Container eingebunden und der Monitor soweit übergeben, dass dieser vom Container angesprochen werden kann.

```
$ export DISPLAY=*your-machine-ip*:0.0
$ startxwin -- -listen tcp &
$ xhost + *your-machine-ip*
$ eval "$(docker-machine env *your-machine-name*)"
$ docker run --rm -e DISPLAY=$DISPLAY -v /tmp/.X11-unix:/tmp/.X11-
    unix firefox
```

Listing 3.1: Windows Skript

Quelle: https://manomarks.net/2015/12/03/docker-gui-windows.html

Babun

Babun baut auf einem vorkonfigurierten und mit Erweiterungen ausgestattetem Cygwin auf. Es bedient sich folglich an dem gleichen Prinzip, dass ein X-server auf einem Windows Host aufgesetzt wird. Babun verfügt darüber hinaus unter anderem über Plugin Erweiterungen und besitzt einen eigenen Paketmanager (vgl. Bujok (2016)).

Xvfb mit VNC

Wie in dem Abschnitt 3.3.1 auf Seite 15 beschrieben, kümmert sich bei dieser Konfiguration der Container alleine um das Erstellen der Benutzeroberfläche mit Hilfe von Xvfb. VNC dient dazu den virtuellen Bildschirm des Containers auf den Host zu übertragen und im Gegenzug die Maus-und Tastatureingaben an den Container zu senden. Damit kann der Container von jeder Plattform aus bedient werden, auf der es möglich ist VNC zu installieren.

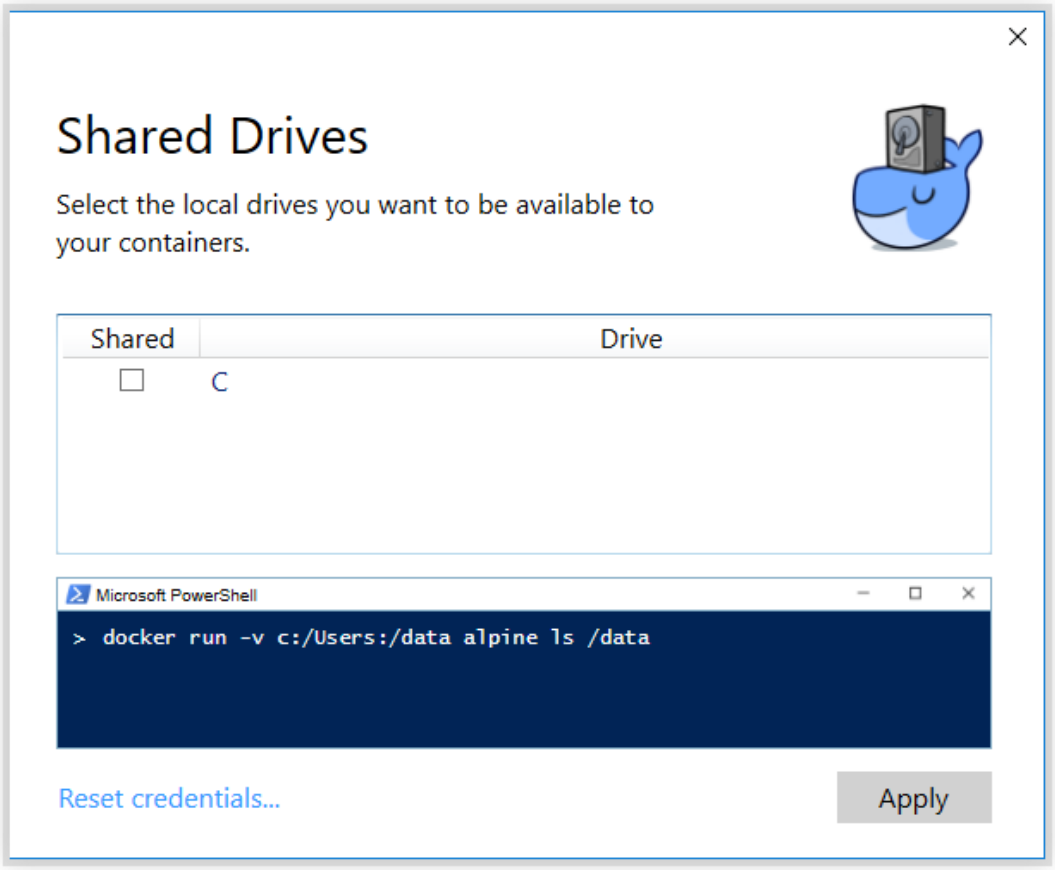
3.4.2 Speicher

Docker-Volume-Plugins

Auch unter Windows lassen sich die Docker-Plugins einsetzen. Die Plugins realisieren eine echte Plattformunabhängigkeit dadurch, dass der Speicher nicht mehr auf dem Host liegt (Abschnitt 3.3.2 auf Seite 16). Dies hat zur Folge, dass es unerheblich ist, welches Betriebssystem dem Host zugrunde liegt.

Docker Volume

Auch unter Windows ist es mittlerweile möglich, Verzeichnisse vom Host mit einem Docker Container zu teilen. Realisiert wird dies zum einem durch die von Microsoft eigens entwickelte Virtualisierungstechnik *Hyper-V* und zum anderen durch die sogenannte Ordnerfreigabe. Docker besitzt ein hauseigenes Werkzeug für Windows mit dem sich solche Einstellungen über eine grafische Oberfläche bedienen lassen (Abbildung 3.4.2 auf Seite 20). Zu beachten ist hier lediglich, dass der Container Zugangsdaten eines Windows Benutzerprofil benötigt, welches Zugriffsrechte auf das Verzeichnis besitzt (vgl. Docker-Inc.. (2016))



Quelle: https://docs.docker.com/docker-for-windows/#shared-drives

Abbildung 3.3: Docker Volume unter Windows

4 Umsetzung

Im Folgenden wird darauf eingegangen, wie die Anforderungen umgesetzt wurden. Dabei wird zuerst auf den grundsätzlichen Aufbau eingegangen. Darauf folgend wird der praktische Teil detailliert erläutert, der sich dabei in zwei Abschnitte unterteilen läst. Das ist zum einen die Dockerfile und zum anderen das Start Skript. Die Dockerfile enthält alle Informationen, die für das Docker Image benötigt werden. Das Start Skript hingegen ist neben dem Aufruf des Containers für die Kommunikation verantwortlich, die zwischen dem Container und dem Host Computer stattfindet.

4.1 Vorgehensweise

Es muss einerseits ein Dockerimage erstellt werden, das die geforderte Software beinhaltet und andererseits soll die Benutzbarkeit realisiert werden. Die Benutzbarkeit wird durch ein Shell Skript erreicht. Das Skript startet einmal den Container und sorgt für die Anzeige der Benutzeroberfläche sowie für das persistente Speichern. Die Techniken und Wege, die im Rahmen dieser Arbeit angewendet wurden, wurden mit der Priorität gewählt besonders ressourcensparend zu sein und einen geringen Mehraufwand zu besitzen.

4.2 Aufbau

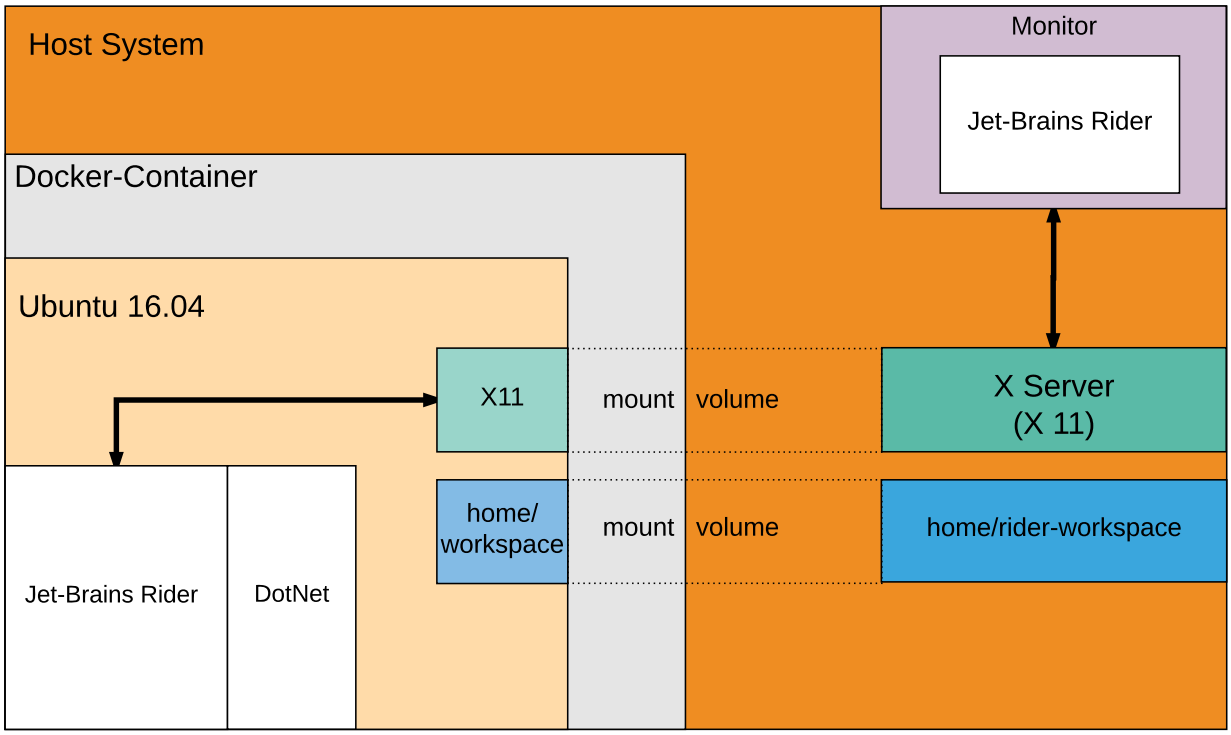


Abbildung 4.1: Aufbau Container-Host

Auf der Abbildung 4.1 auf Seite 22, wird das Host System mit den Schlüsselkomponenten dargestellt, die für diese Arbeit eine Rolle spielen. Wie zu sehen ist, wird das Einbinden von Verzeichnissen, benutzt um eine persistente Speicherung zu ermöglichen und um die Benutzeroberfläche zu übetragen. Auf dem Docker Container läuft die Linux-Distribution Ubuntu mit der Version 16.04, welche wiederum die IDE-Softtware Jet-Brains Rider sowie die Bibliothek DotNet installiert hat. Damit Projekte, die mit der IDE erstellt worden sind, langfristig gespeichert werden können, wurde das Verzeichniss *rider-workspace* von dem Host System unter den Namen *workspace* in dem Container eingebunden.
Für die grafische Übertragung ist der X-Server des Host Systems verantwortlich. Dieser besitzt ein Socket, dessen Verzeichnis wie das vorherige in dem Container eingebunden wird. Dadurch ist es möglich, die grafische Ausgabe auf den an den Host angeschlossenen Bildschirm zu übertragen.

Im Folgenden wird näher auf die einzelnen Komponenten eingegangen, die im Laufe dieser Arbeit erstellt wurden. Das ist zum einem das Docker Image beziehungsweise Die Dockerfile und zum anderen das Shell Skript, welches zum Starten des Containers benötigt wird.

4.3 Aufbau der Dockerfile

Listing 4.1: Dockerfile

```
1  FROM ubuntu:16.04
2
3  MAINTAINER Jonny Niebuhr <jonny.niebuhr@haw-hamburg.de>
4
5  RUN apt-get -y update \
6   && apt-get -y install libxext6 \
7      libxtst6 \
8      libxi6 \
9      libstdc++6 \
10     libxrender1 \
11     libfreetype6 \
12     libunwind8 \
13     wget \
14     tzdata \
15  && wget --progress=bar:force -O dotnet.tar.gz https://go.microsoft.
       com/fwlink/?LinkID=835021 \
16  && mkdir -p /opt/dotnet && tar zxf dotnet.tar.gz -C /opt/dotnet \
17  && ln -s /opt/dotnet/dotnet /usr/local/bin \
18  && rm dotnet.tar.gz
19  RUN wget --progress=bar:force -O rider.tar.gz https://download.
       jetbrains.com/resharper/riderRS-171.4089.466.tar.gz \
20  && mkdir -p /opt/rider && tar zxf rider.tar.gz -C /opt/rider \
21  && rm rider.tar.gz \
22  && apt-get -y clean \
23  && apt-get -y autoremove
24
25 ENTRYPOINT ["/opt/rider/Rider-171.4089.466/bin/rider.sh"]
```

Beim Bau eines Docker Images besteht der Anfang aus der Dockerfile. Die Dockerfile enthält sämtliche Informationen, die für den Bau des Images und damit für die Instanziierung eines Containers nötig sind.
Die Dockerfile kann mit jedem Texteditor erstellt werden. Damit der Docker Daemon die Datei lesen und als Dockerfile erkennen kann, darf diese keine Endung besitzen ("Dockerfile"nicht "Dockerfile.txt") und muss zudem als "Dockerfile"bezeichnet werden. Auf Seite 23 ist die Dockerfile zu sehen, die die Instruktionen für das Image enthält.
Im Folgenden wird auf die einzelnen Befehle eingegangen.

Listing 4.2: FROM

```
1 FROM ubuntu:16.04
```

Das Base Image ist Ubuntu 16.04. Im Gegensatz zur normalen Desktop Installation handelt es sich bei dem Docker Image um eine Minimal-Installation von Ubuntu. Das heißt, dass unter anderem nur die nötigsten Sprachpakete und Bibliotheken installiert sind.

Listing 4.3: MAINTAINER

```
1 MAINTAINER Jonny Niebuhr <jonny.niebuhr@haw-hamburg.de>
```

Hier steht lediglich der Autor der Dockerfile.

Listing 4.4: RUN/Update

```
1 RUN apt-get -y update \
```

apt-get -y Mit dem Befehl *apt-get -y* (*apt* kurz für englisch Advanced Packaging Tool) wird das Packetmanagment angesprochen und der Parameter *-y* dient der automatischen Bestätigung des Befehls.
Update aktualisiert die Paketliste und vollzieht keine Veränderung am System.

Listing 4.5: Bibliotheken

```
1  && apt-get -y install libxext6 \
2     libxtst6 \
3     libxi6 \
4     libstdc++6 \
5     libxrender1 \
6     libfreetype6 \
7     libunwind8 \
```

Zusehen sind die Bibliotheken, die für das Image benötigt werden und von dem Paketmanagement zur Verfügung gestellt werden. Alle Pakete die mit *libx* beginnen, dienen der Darstellung der Benutzeroberfläche auf dem Monitor. Es sind Bibliotheken für das X Window System (auch X Version 11, X11 oder nur X), das unter anderem für grafische Benutzeroberflächen verantwortlich ist. Darauf wird in Abschnitt 4.5.1 auf Seite 28 ausführlicher eingegangen.
Das Paket *libstdc++6* ist die C++ Standard Bibliothek und wird für die Projekte benötigt, die man mit der Rider IDE erstellen kann. Das Paket *libfreetype6* ist eine Bibliothek für diverse Schriftarten, die zur Texterstellung von Benutzeroberflächen erforderlich sind. Das Ziel der Bibliothek *libunwind8* ist eine portable und effiziente C Programmierschnittstelle (API kurz für englisch application programming interface), um die Aufrufketten eines Programmes zu bestimmen.

Listing 4.6: Download/Zeit

```
1     wget \
2     tzdata \
```

In diesem Arbeitsschritt wird das Paket *wget* installiert, um die IDE *Rider* sowie das *dotnet* Framework aus dem Internet zu laden. Das ist erforderlich da beide Programme nicht im Paketmanagement geführt werden. *Wget* dient unter anderem dazu ohne Benutzerinteraktion Dateien von Servern zu laden. Die Steuerung erfolgt über Parameter, die beim Aufruf mitgegeben werden.
Das Paket *tzdata* enthält eine Software, die unter anderem für die lokale Systemzeit verantwortlich ist und wird von der IDE Rider benötigt um Log Dateien zu erstellen.

Listing 4.7: IDE und Framework

```
1 && wget --progress=bar:force -O dotnet.tar.gz https://go.microsoft.
      com/fwlink/?LinkID=835021 \
2 && mkdir -p /opt/dotnet && tar zxf dotnet.tar.gz -C /opt/dotnet \
3 && ln -s /opt/dotnet/dotnet /usr/local/bin \
4 && rm dotnet.tar.gz
5 RUN wget --progress=bar:force -O rider.tar.gz https://download.
      jetbrains.com/resharper/riderRS-171.4089.466.tar.gz \
6 && mkdir -p /opt/rider && tar zxf rider.tar.gz -C /opt/rider \
7 && rm rider.tar.gz \
```

Hier werden die IDE sowie das Framework geladen und im System eingebettet. Bei dem Befehl mit dem Parameter *wget –progress=bar: force -O <Dateiname> <URL>* handelt es sich um die Downloads. Der Parameter *–progress=bar:force* forciert dabei einen Progressbalken,

der den Verlauf des Downloads anzeigt. Diese Anzeige erscheint beim Bauen des Images. Bei *-O <Dateiname>* wird das von dem Server Geladene in den angegebenen Dateinamen geschrieben, sodass der Umgang mit dieser Datei erleichtert wird. Im Folgenden werden die Schritte veranschaulicht, die bei der IDE sowie bei dem Framework benutzt werden:

mkdir -p /opt/dotnet
Es wird das Verzeichnis erstellt, in dem die Software installiert werden soll.

tar zxf dotnet.tar.gz -C /opt/dotnet
Es wird mit *-C* in das vorher erstellte Verzeichnis gewechselt, in das die Datei entpackt wird.

rm dotnet.tar.gz
Das Paket wurde nun entpackt, das heißt, dass die Software installiert wurde und somit gelöscht werden kann.

Für das Framework wird noch eine symbolische Verknüpfung mit dem Befehl **ln -s /opt/dotnet/dotnet /usr/local/bin** erstellt. Das Verzeichnis ist für Programme gedacht, die man an der Paketverwaltung vorbei installieren möchte. Dies ist erforderlich, damit die IDE auf das Framework zugreifen kann.

Der Zweite **RUN** Befehl stellt eine weitere Image Schicht dar. Auf die Funtkionsweise des Images hat er keinen Einfluss. Will man jedoch lediglich eine neuere Version der IDE in das Image einpflegen oder gar eine komplett andere IDE benutzen, so könnte Docker die Schichten lokal laden, die schon gebaut wurden.

Listing 4.8: Clean/Autoremove

```
1 && apt-get -y clean \
2 && apt-get -y autoremove
```

Am Ende des Run Abschnitts stehen die Befehle *clean* und *autoremove*. Der Befehl *clean* löscht die bereits heruntergeladenen Installationsdateien aus dem Paket-Cache. *Autoremove* deinstalliert nicht mehr benötigte Pakete, die keine Abhängigkeiten mehr aufweisen.

Listing 4.9: ENTRYPOINT

```
1 ENTRYPOINT ["/opt/rider/Rider-171.4089.466/bin/rider.sh"]
```

Der *ENTRYPOINT* eines Images stellt die Aktion dar, die beim Instantiieren eines Containers ausgeführt wird. In diesem Fall ist es das Startskript von der IDE Rider. Somit wird beim Starten des Containers als Abbild des Images direkt die Rider IDE gestartet und es öffnet sich die Benutzeroberfläche.

4.4 Docker Image erstellen und in die Registry einfügen

Mit dem Befehl '*docker build -t nijo/rider:latest .*' lässt sich nun das Image bauen. Dabei ist '*docker build <Parameter> .*' der eigentliche Build-Befehl und die Parameter *-t nijo/rider:latest* sind für die Namensgebung verantwortlich. Diese enthalten in diesem Fall direkt die Struktur, die im Docker-Hub erforderlich ist. Hier ist *nijo* der Name des Kontos, *rider* der Name des Repositories und *:latest* der sogenannte 'Tag'. Wird kein Tag angegeben, wird *latest* als Standard verwendet. Das gilt auch beim Laden eines Images aus dem Docker Hub.
Anschließend kann man das Image direkt mit dem Befehl *docker push nijo/rider:latest* in die Registry laden.

```
Username: nijo
Password:
Login Succeeded
root@ba-VirtualBox:/home/ba/jb-rider-ubuntu-latest# docker push nijo/rider:latest
The push refers to a repository [docker.io/nijo/rider]
ff7100942335: Pushing [==================================================>]  1.18 GB
73e5d2de6e3e: Mounted from library/ubuntu
08f405d988e4: Mounted from library/ubuntu
511ddc11cf68: Mounted from library/ubuntu
a1a54d352248: Mounted from library/ubuntu
9d3227c1793b: Mounted from library/ubuntu
```

Abbildung 4.2: Docker Push

4.5 Start-Skript

Listing 4.10: Startskript für den Container

```
#non-network local connections being added to access control list of
    the X server
xhost +local:root

#run the container and gives the graphic output to the host via the
   x11 socket
#also its mounting the directory /home/rider-Workspace to the
   container for saving projects on the hosts filesystem
docker run -ti \
       -e DISPLAY=$DISPLAY
       --volume="/home/rider-workspace:/home/workspace" \
       -v /tmp/.X11-unix:/tmp/.X11-unix \
       nijo/rider


#non-network local connections being removed from access control
   list of the X server
xhost -local:root
```

Im Folgenden wird das Shell Skript vorgestellt, mit dem der Container gestartet wird. Das Skript erledigt dabei zwei Aufgaben. Zum einen sorgt es dafür, dass der Container in der Lage ist die Benutzeroberfläche an das Hostsystem zu übergeben und zum anderen verknüpft es ein Verzeichnis zwischen Container und Hostsystem. Letzteres hat die Wirkung, dass Daten direkt auf dem Host gespeichert werden können und von dem Host gelesen werden können.

4.5.1 Weiterleitung der Benutzeroberfläche

Eines der Schlüsselelemente der Kommunikation zwischen Host System und dem Docker Container ist der X-Server.
Der X-Server pflegt eine Liste mit Benutzern. Diese kann man mit dem xhost Befehl verändern, in dem man den gewünschten Host beziehungsweise Benutzer in dieser Liste hinzufügt.

Listing 4.11: X Server

```
xhost +local:root
xhost -local:root
```

Mit den beiden Befehlen, wobei der eine am Anfang des Skriptes steht und der andere am Ende, wird nun ein leerer String mit dem Befehl *xhost +local:root* in diese Liste eingetragen. Dies ermöglicht allen lokalen Benutzern und Systemen sich mit dem X-Server zu verbinden. Also auch dem Docker Container. Der Befehl am Ende sorgt dafür, dass der leere String wieder aus der Liste entfernt wird. Das geschieht, wenn der Container geschlossen wird und somit die Operation der Ausführung damit beendet ist (Zeile 6 bis 11 in Listing 4.10).

4.5.2 Speicher

Docker Container weisen keinen persistenten Speicher auf. Ein Container wird auf der Grundlage eines Images erstellt. Wird der Container geschlossen, so werden auch alle neuen Daten oder Veränderungen seit dem Start des Containers verworfen. Bei dem Aufrufen eines Images wird jedes mal ein neuer Container erstellt. Um Projekte, die unter der IDE erstellt wurden oder weiterentwickelt werden sollen, dauerhaft zu speichern und auf diese zuzugreifen wird ein Verzeichnis in dem Container gemounted. Das Start-Skript erstellt ein Verzeichnis auf der Home-Ebene des Host Systems. Dieses Verzeichnis wird dazu verwendet, um Projekte, die mit der IDE erstellt wurden, direkt auf dem Host zu speichern. Sollte das Verzeichnis schon bestehen wird es nicht überschrieben und direkt gemounted. Damit wird die dauerhafte Speicherung von Daten ermöglicht. Zudem können auch schon bestehende Projekte weiterentwickelt werden.

Listing 4.12: Docker Aufruf

```
1 docker run -ti \
2        -e DISPLAY=$DISPLAY
3        --volume="/home/rider-workspace:/home/workspace" \
4        -v /tmp/.X11-unix:/tmp/.X11-unix \
5        nijo/rider
```

Im Listing 4.12 auf Seite 29 sehen wir den Aufruf des Docker Images. Der eigentliche Aufruf besteht aus dem Befehl *docker run -ti rider*. Die Parameter *-t* und *-i* sorgen dafür, dass es dem Container möglich ist in der Shell zu antworten, in der er gestartet wurde. Dazwischen stehen bestimmte Aufrufparameter, auf die jetzt näher eingegangen wird.

Listing 4.13: Übergeben des Bildschirms

```
1        -e DISPLAY=$DISPLAY
```

Hier wird die System Variable *DISPLAY* gleichgesetzt mit der des Hostsystems . Damit ist der

Container in der Lage den Bildschirm des Host System anzusprechen, um zu bestimmen, wo die Benutzeroberfläche angezeigt werden soll.

Listing 4.14: Einbinden des Verzeichnisses

```
1       --volume="/home/rider-workspace:/home/workspace" \
```

Hier wird das Verzeichnis des Hosts in dem Container eingebunden, wobei vor dem Doppelpunkt der Pfad des Hosts steht und danach der Pfad, auf dem es im Container eingebettet wird.

Listing 4.15: Einbinden des X-Server Sockets

```
1       -v /tmp/.X11-unix:/tmp/.X11-unix \
```

Damit der Container auch mit dem X-Server kommunizieren kann, wird der Socket des X-Serves auch in dem Container integriert. Somit ist es möglich für einen X-Client mit dem X-Server zu kommunizieren, ohne dass es für eine Seite ersichtlich ist, dass sie sich auf zwei verschiedenen Systemen befindet.

Listing 4.16: Aufruf des Images

```
1       nijo/rider
```

Hier steht der Imagename, von dem der Container abstammen soll. Wobei *rider* der Name, oder auch TAG, des Repositories beziehungsweise des Images ist und *nijo* der Name des Kontos, auf welchem sich das Repository befindet (Abbildung 4.8 auf Seite 36). Docker überprüft zuerst, ob das Image schon lokal vorhanden ist. Ist dies nicht der Fall, wird im Docker Hub nachgeschaut aus welchen Schichten das Image besteht und gegebenenfalls werden die fehlenden Schichten geladen. Hat man zum Beispiel das Docker Ubuntu 16.04 Base Image schon lokal auf dem System, muss dieses nicht noch einmal geladen werden.

Terminal-Skript

Damit es möglich ist den Container zu starten, ohne dass umgehend die IDE startet, muss man lediglich die Zeile des Aufrufes ändern. Um den Container nun so zu starten, dass man sich direkt im Terminal befindet, muss man den ENTRYPOINT (englisch für Einstiegspunkt) überschreiben.

Listing 4.17: Überschreiben des ENTRYPOINTS

```
1       --entrypoint /bin/bash nijo/rider
```

So ist es möglich den Container anzupassen und mit dem Befehl *docker commit [OPTIONS] Container ID [repositorie[:tag]]* ein neues Image zu erstellen, das auf das alte aufbaut. Zu bedenken dabei ist lediglich, dass dadurch auch der *ENTRYPOINT* neu gesetzt wurde und zwar mit */bin/bash*. Damit das Image wieder direkt die IDE aufruft sobald es gestartet wird, muss also wieder der *ENTRYPOINT* überschrieben werden. Die Veränderung wird sodann mit dem *commit* Befehl dauerhaft gespeichert.

4.6 Konfiguration

Das resultierende Image, welches aus der Dockerfile hervorgeht, beinhaltet nun sämtliche geforderte Software. Ruft man nun das Image über das Shell Skript auf, wird ein Container gestartet und es öffnet sich ein Fenster von Rider als wäre es gerade lokal installiert worden. Damit sich beim Starten des Containers direkt ein Fenster öffnet, das ein direktes Arbeiten ermöglicht, müssen noch einige Arbeitsschritte absolviert werden.
Im Folgenden werden die einzelnen Schritte aufgezeigt, die dem bestehenden Image beigefügt werden.

4.6.1 Rider

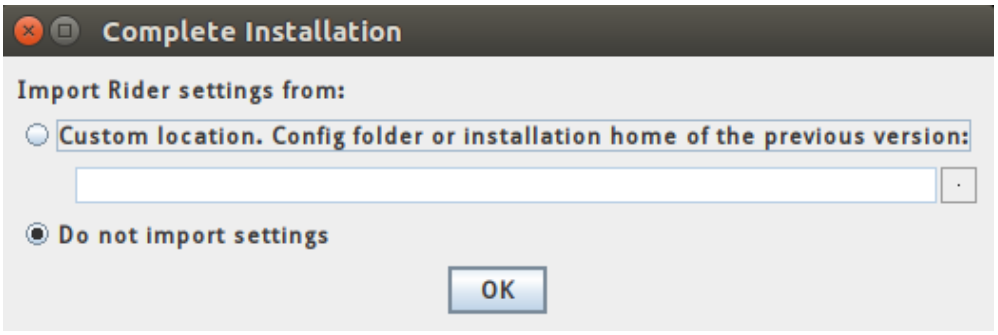


Abbildung 4.3: Rider-Container erster Start

Hier hat man nun die Möglichkeit Einstellungen zu importieren oder eigenständig zu definieren. Da keine Einstellungen aus vorherigen Versionen vorhanden sind, wird mit der letzteren Option fortgefahren.

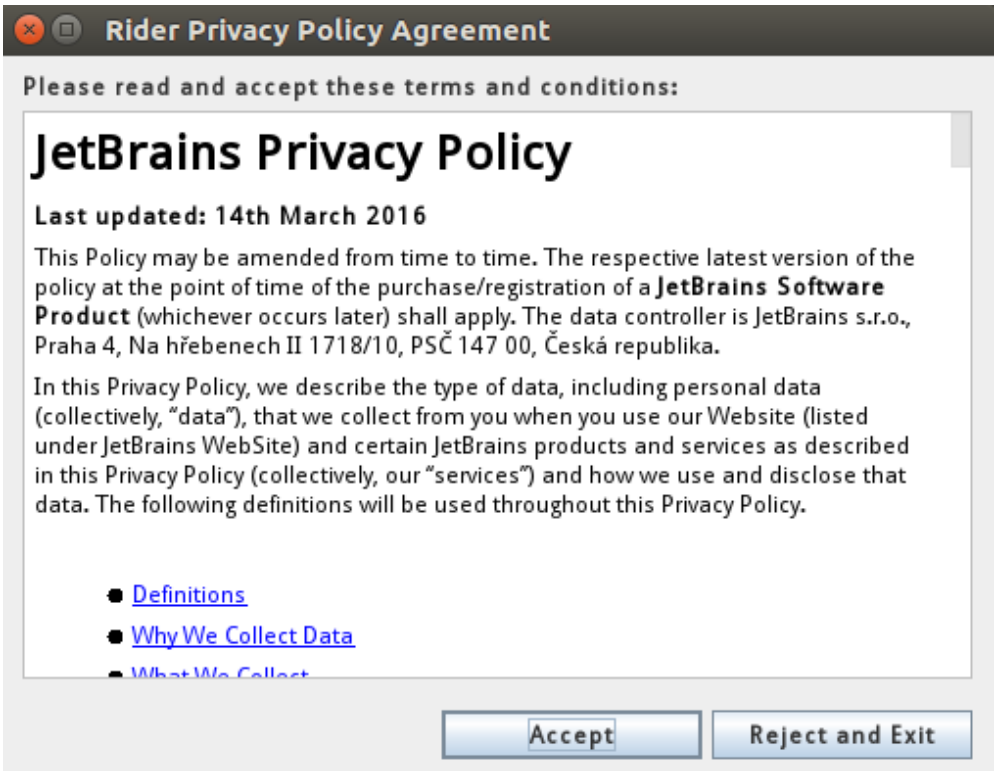


Abbildung 4.4: Rider Geschäftsbedingungen

Als nächstes wird man dazu aufgefordert die Geschäftsbedingungen zu akzeptieren.

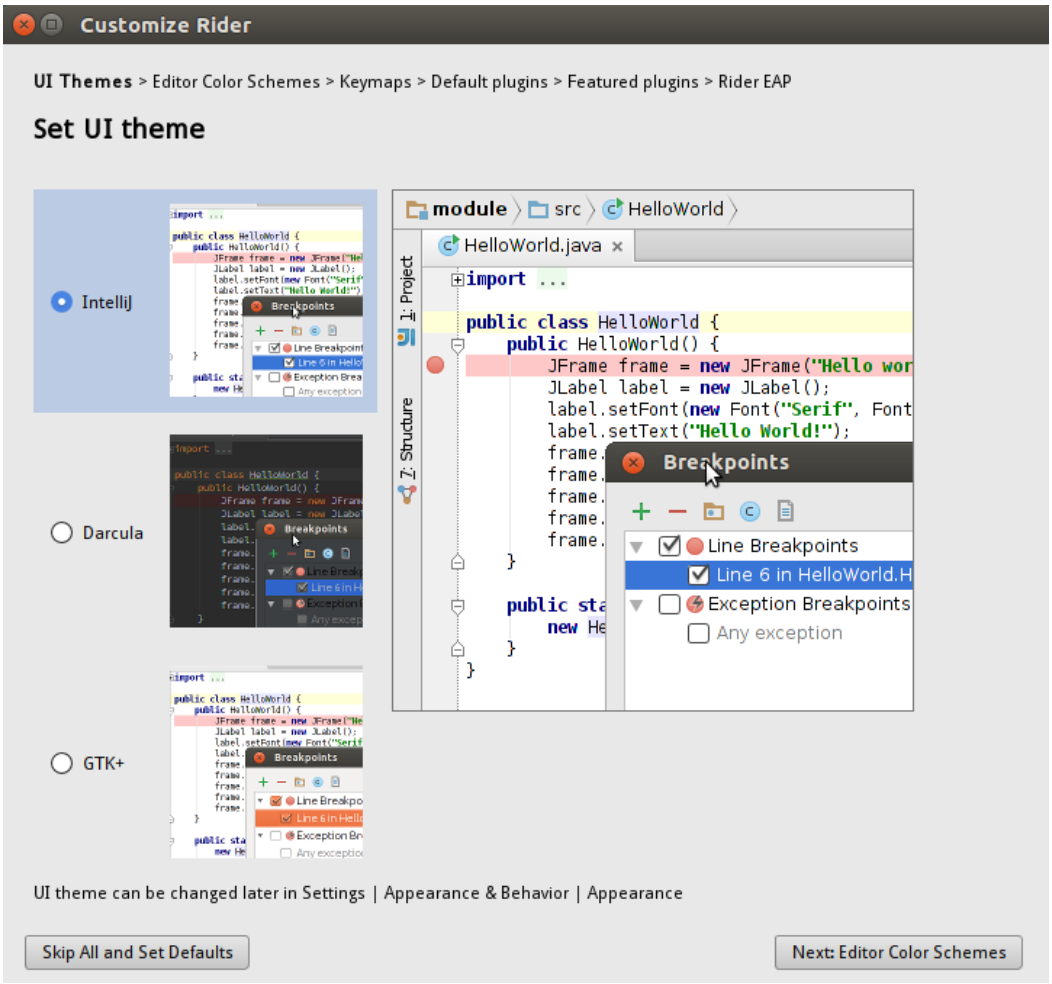


Abbildung 4.5: Rider Anpassung

Anschließend besteht die Option Rider nach seinen eigenen Belieben anzupassen. Hier werden diese Einstellungsmöglichkeiten überpsrungen und die Option *Skip All and Set Defaults* gewählt.

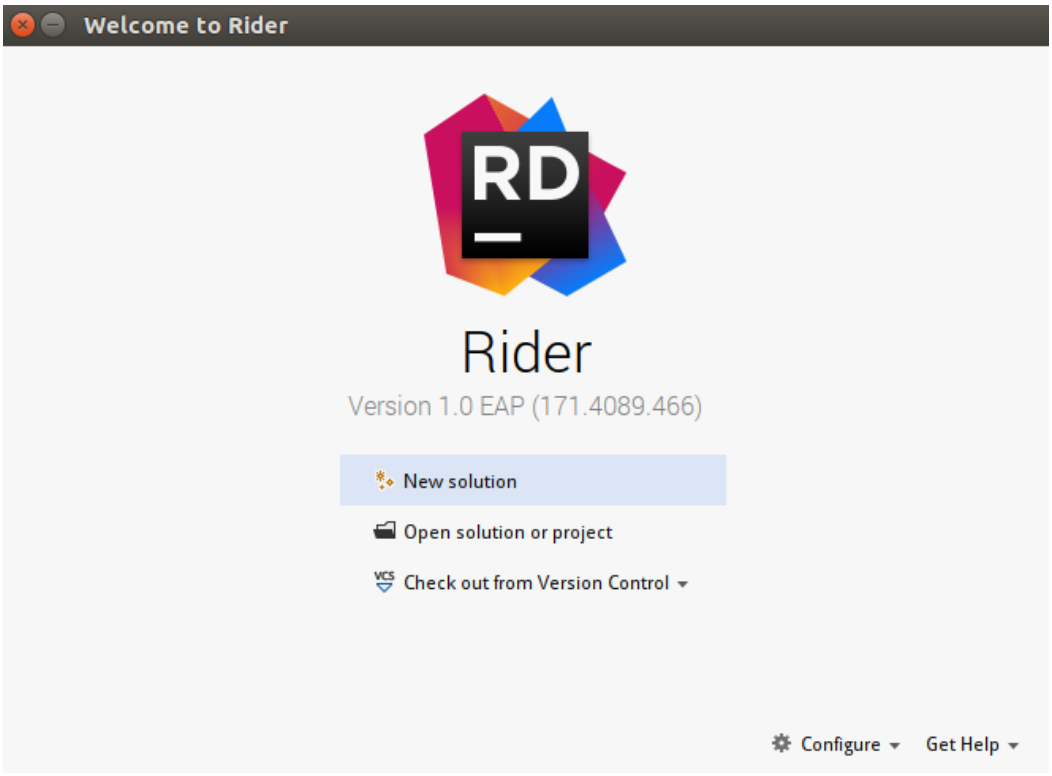


Abbildung 4.6: Rider Start Standard

Bei diesem Fenster hat man die Wahl ein neues Projekt zu starten oder ein schon bestehendes zu öffnen.
In der Version des Images ist der Stand des Containers, dass bei jedem neuen Aufruf diese Prozessschritte durchlaufen werden müssen. Die Dateistruktur bei Docker ist in Schichten organisiert. Dadurch ist es mit einem *commit* Befehl möglich den Container auf das Image zu schreiben. Dieser bildet dann eine neue Schicht und damit dann ein neues Image, welches auf dem originalen Image aufbaut.

4.6.2 Finales Docker Image

Der Container, der wie eine weitere Schicht auf dem Image liegt (siehe Abbildung 2.4 auf Seite 9), hat jetzt den Zustand, der für das Image gelten soll. Anschließend wird die oberste Schicht, der beschreibbare Container, auf das schon existierende Image geschrieben. Der Befehl dafür lautet
docker commit -m="default configuration"6d22081aa1a2 nijo/rider:default.
Die Zeichenfolge *6d22081aa1a2* ist dabei die ID des Containers, die zufällig generiert wird und das neue Image trägt den gleichen Namen wie jenes, auf dem es basiert. Dies lässt sich dadurch erreichen, dass es einen anderen *Tag* bekommen hat, nämlich *:default*. Lädt man nun dieses Image in das gleiche Repository wird nur die neue Schicht hochgeladen.

```
root@ba-VirtualBox:/home/ba/jb-rider-ubuntu-latest# docker push nijo/rider:default
The push refers to a repository [docker.io/nijo/rider]
ace49755661d: Pushing [==================================================>] 64.75 MB
ff7100942335: Layer already exists
73e5d2de6e3e: Layer already exists
08f405d988e4: Layer already exists
511ddc11cf68: Layer already exists
a1a54d352248: Layer already exists
9d3227c1793b: Layer already exists
```

Abbildung 4.7: Rider Push

Im Docker Hub, der Registry von Docker, sieht es dann wie folgt aus.

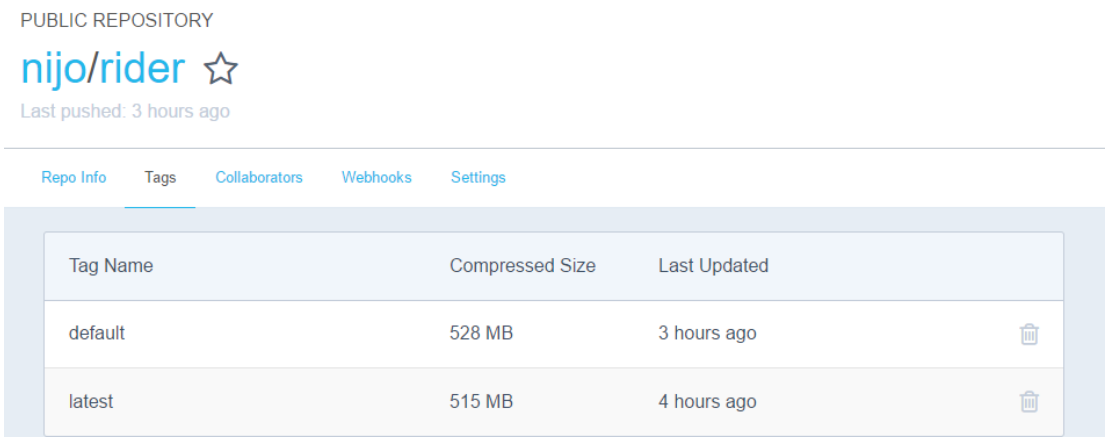


Abbildung 4.8: Docker Hub

Dadurch ist es möglich, die Versionierung und auch eine gewisse Variation eines Images zu realisieren. In diesem Fall liegt ein Image mit dem Tag "latest"vor, welches keine Konfiguration nach der Installation von Rider beinhaltet. Zusätzlich gibt es ein Image mit einer weiteren Schicht mit dem Tag "default". Dieses enthält die Standardkonfigurationen, die nötig sind, um direkt an einem Projekt zu arbeiten oder eines neu zu erstellen.
Im Folgenden kann man eine Abbildung sehen, die die Schichten des endgültigen Images visualisiert und anzeigt, welche Größe jede Schicht und das Image als Gesamtes besitzt.

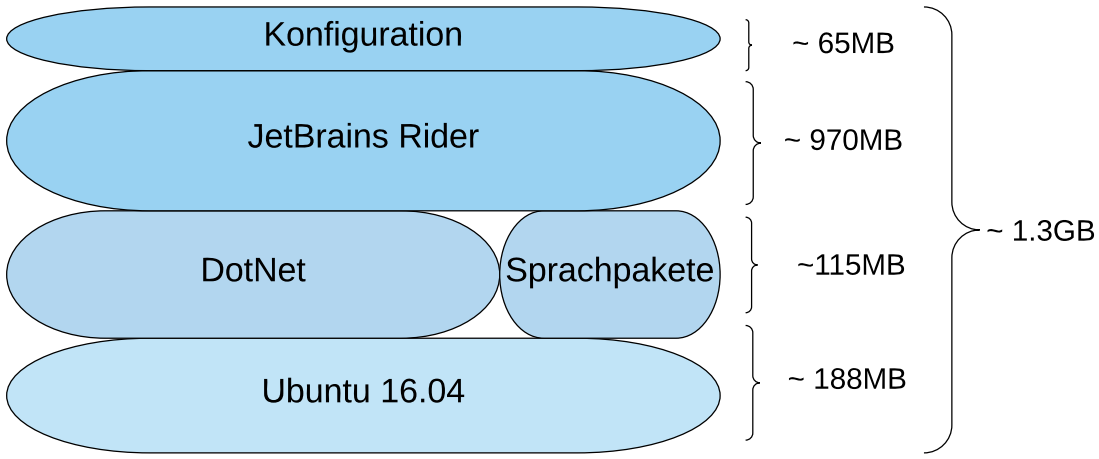


Abbildung 4.9: Image Schichten und Größen

5 Ergebnis

5.1 Zusammenfassung

Diese Arbeit beschäftigte sich mit der Konzeption und der möglichen Umsetzung einer IDE in einem Docker Container. Es wurde nach einer Möglichkeit gesucht eine Integrierte Entwicklungsumgebung in einen Docker Container zu implementieren und diese über die eigene grafische Benutzeroberfläche bedienbar zu machen. Dazu sollte es möglich sein aus dem Docker Container heraus Daten dauerhaft zu Speichern und zugreifbar zu machen. In diesen Zusammenhang wurde nach Wegen gesucht, um für die grafische Benutzeroberfläche sowie für das dauerhafte Speichern von Daten jeweils eine Schnittstelle zu realisieren.
Im Verlauf dieser Arbeit wurde eine praktische Umsetzung realisiert. Bei dieser Umsetzung war es die Priorität, dass sie innerhalb des Zeitrahmens dieser Arbeit fertiggestellt werden könnte.
Es stellte sich heraus, dass das Konzept einer Softwareeentwicklungsumgebung in einem Docker Container realisierbar ist. Das Ziel wurde erreicht, indem die grafischen Informationen an den Host Computer übermittelt wurden und Daten dauerhaft in die Verzeichnisstruktur des Host Computer geschrieben wurden. Die Voraussetzung dafür schafft ein Skript, das Shell Befehle beinhaltet. Diese Befehle umfassen zum einen die geforderten Schnittstellen für grafische Informationen und den persistenten Speicher sowie zum anderen den Aufruf des Containers selber.

Im Verlauf dieser Arbeit wurde nach alternativen Lösungsansetzen recherchiert. Das ergab, dass bei der Schnittstelle für persistenten Speicher sowie bei der grafischen Schnittstelle jeweils ein weiterer Ansatz existiert. Das persistente Speichern von Daten mittels Docker Containern wurde neben der hier angewendeten Technik auch mit Docker-Volume-Plugins realisiert. Es existieren für diverse Anwendungsfälle verschiedene Plugins.
Ein anderer Ansatz für die grafische Schnittstelle ist der, dass der Container mit Hilfe einer Software seinen eigenen Bildschirminhalt virtuell generiert und selber einen X-Server bereitstellt. Außerdem benötigt der Container noch die Fernwartungssoftware VNC. Der virtuelle Bild-

schirminhalt kann mittels der VNC Software auf den Host Computer übertragen werden und damit auf einen physischen Bildschirm dargestellt werden. VNC ermöglicht es im Gegenzug das Tastatur- und Mauseingaben an den Container gesendet werden.

5.2 Evaluation

Es hat sich herausgestellt, dass bei dieser Arbeit vordergründig zwei Schnittellen im Fokus stehen. Das ist die Schnittstelle für persistentes Speichern und die Schnittstelle für die Benutzeroberfläche. Beide Ansätze, die auf Docker Volume beziehungsweise Docker Volume Plugins basieren, sind bewährt und zudem plattformunabhängig. Es stellt sich hier lediglich die Frage, ob die Daten lokal oder extern gespeichert werden sollen.
Bei der Umsetzung der grafischen Schnittstelle zeigte sich jedoch, dass das Docker System nicht dafür konzipiert ist eine Schnittstelle für Grafik zu besitzen. Das hat zur Folge, dass beide Lösungsansätze Techniken verwenden, die für diese Anwendung nicht erdacht waren.
Der Lösungsansatz, der in dem praktischen Teil dieser Arbeit veranschaulicht wurde, eignet sich nur für Systeme, die einer Linux Distribution zu Grunde liegen. Damit geht die Plattformunabhängigkeit verloren. Anzumerken ist, dass die Schnittstelle unter Umständen eine Sicherheitslücke darstellen kann.
Der alternative Lösungsansatz sichert zum einem die Platformunabhängikeit und schafft eine sichere Verbindung, zum anderen erfordert er wiederum die Implementation weiterer Software in dem Container. Dies ist mit einem Mehraufwand in der Konfiguration und Wartung des Images verbunden.

Literaturverzeichnis

[Bujok 2016] Bujok, Tom: *Babun - a windows shell you will love*. http://babun.github.io/. 2016. – [Online; Zugriff 23.05.2017]

[Docker-Inc.. 2016] Docker-Inc..: *Shared Drives*. https://docs.docker.com/docker-for-windows/#shared-drives. 2016. – [Online; Zugriff 25.05.2017]

[Docker-Inc. 2017] Docker-Inc.: *Dockerfile reference*. https://docs.docker.com/engine/reference/builder/#dockerignore-file. 2017. – [Online; Zugriff 02.05.2017]

[Docker-Inc.. 2017] Docker-Inc..: *Plugins*. https://docs.docker.com/engine/extend/legacy_plugins/. 2017. – [Online; Zugriff 20.05.2017]

[Hykes 2014] Hykes, Solomon: *DOCKER 0.9: INTRODUCING EXECUTION DRIVERS AND LIBCONTAINER*. https://blog.docker.com/2014/03/docker-0-9-introducing-execution-drivers-and-libcontainer/. 2014. – [Online; Zugriff 01.05.2017]

[JetBrains 2017] JetBrains: *JetBrains Rider*. https://www.jetbrains.com/rider/. 2017. – [Online; Zugriff 01.05.2017]

[Thomas Joos 2017] Thomas Joos, Stephan A.: *Plugins*. http://www.dev-insider.de/docker-und-andere-container-unter-windows-a-572054/. 2017. – [Online; Zugriff 23.05.2017]

[Wiki02 2017] Wiki02: *.NET Frameworkr*. https://de.wikipedia.org/wiki/.NET_Framework#Verh.C3.A4ltnis_zu_Mono. 2017. – [Online; Zugriff 01.05.2017]

[Wiki03. 2017] Wiki03.: *Docker Hub*. https://de.wikipedia.org/wiki/Docker_(Software)#Docker_Hub. 2017. – [Online; Zugriff 10.05.2017]

[Wiki04 2017] Wiki04: *Cygwin*. https://de.wikipedia.org/wiki/Cygwin. 2017. – [Online; Zugriff 23.05.2017]

[Wiki04. 2017] Wiki04.: *VNC*. https://de.wikipedia.org/wiki/Virtual_Network_Computing. 2017. – [Online; Zugriff 15.05.2017]

[Wiki05 2017] Wiki05: *Xvfb*. https://de.wikipedia.org/wiki/Xvfb. 2017. – [Online; Zugriff 23.05.2017]

[Wiki06 2017] Wiki06: *Integrierte Entwicklungsumgebung*. https://de.wikipedia.org/wiki/Integrierte_Entwicklungsumgebung. 2017. – [Online; Zugriff 15.05.2017]

[WikiU01 2017] WikiU01: *X Server*. https://wiki.ubuntuusers.de/XServer//. 2017. – [Online; Zugriff 04.04.2017]

[Zimmer 2012] Zimmer, Dennis: *VMware vSphere 5: Das umfassende Handbuch*. Galileo Press GmbH, 2012. – ISBN 383621847X

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

Hamburg, 09. Juni 2017 Jonny Niebuhr