Template LaTeX Wiki von BAzubis für BAzubis

Projektarbeit 1 (T3_1000)

im Rahmen der Prüfung zum

Bachelor of Science (B.Sc.)

des Studienganges Informatik

an der Dualen Hochschule Baden-Württemberg Karlsruhe

von

Yannik Schiebelhut

Abgabedatum:	04. Oktober 2021
Bearbeitungszeitraum:	01.10.2020 - 03.10.2021
Matrikelnummer, Kurs:	3354235, TINF20B1
Ausbildungsfirma:	SAP SE Dietmar-Hopp-Allee 16 69190 Walldorf, Deutschland
Betreuer der Ausbildungsfirma:	Helge Dickel
Gutachter der Dualen Hochschule:	DH-Vorname DH-Nachname

Eidesstattliche Erklärung

Ich versichere hiermit, dass ich meine Projektarbeit 1 (T3_1000) mit dem Thema:

Template LaTeX Wiki von BAzubis für BAzubis

gemäß § 5 der „Studien- und Prüfungsordnung DHBW Technik“ vom 29. September 2017 selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Die Arbeit wurde bisher keiner anderen Prüfungsbehörde vorgelegt und auch nicht veröffentlicht.

Ich versichere zudem, dass die eingereichte elektronische Fassung mit der gedruckten Fassung übereinstimmt.

Karlsruhe, den 26. August 2021

Schiebelhut, Yannik

Inhaltsverzeichnis

Abkürzungsverzeichnis

AJAX	Asynchronous Javascript and XML
API	Application Programming Interface
EWM	Extended Warehouse Management
ewm-sim	EWM Simulator
GKE	Google Kubernetes Engine
HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol Secure
IP	Internet Protocol
JSON	JavaScript Object Notation
npm	Node Package Manager
OData	Open Data Protocol
SDK	Software Development Kit
SSH	Secure Shell
WHO	Warehouse Order
WSL	Windows-Subsystem für Linux

Abbildungsverzeichnis

Tabellenverzeichnis

Quellcodeverzeichnis

1 Einleitung

1.1 Was ist der ewm-sim?

In der vierten industriellen Revolution verändert sich auch der Arbeitsalltag in Lagerhallen. Mobile Roboter finden verstärkt Einsatz, um die Arbeiter zu unterstützen. Das Projekt Extended Warehouse Management (EWM) Cloud Robotics der SAP hat das Ziel, die Integration von Robotern verschiedenster Hersteller in ein Netzwerk auf Basis von Google Cloud Robotics zu ermöglichen und dieses an ein SAP EWM-System anzubinden. Zu Demonstrations- und Entwicklungszwecken wird eine Simlationsumgebung erstellt, in der ein virtuelles Warenlager präsentiert wird, in dem Roboter beispielhafte Aufträge bearbeiten. Um nun zu vermeiden, dass ein vollständiges EWM-System für solch eine Simulation deployed werden muss, wurde der „ewm-sim“ eingeführt. Er stellt einen kleinen Web-Server dar, welcher die Schnittstelle, über die die Roboter ihre Aufträge vom EWM-System erhalten, detailgetreu nachbildet.

1.2 Kritikpunkte der ursprünglichen Implementierung

Wie bereits erwähnt, soll der ewm-sim die Schnittstelle eines EWM-Systems nachbilden. Hierbei handelt es sich um einen Open Data Protocol (OData)-Service. Für die Implementierung wurde hier auf den bestehenden Mock Server von SAPUI5 gesetzt, der normalerweise in der Frontendentwicklung dazu dient, entsprechende Schnittstellen einer Datenbank nachzubilden. Dieser ist jedoch nicht für die Backend-Entwicklung vorgesehen. Leider bringt er somit das Problem mit sich, dass er sich nur innerhalb der Laufzeitumgebung einer SAPUI5-App verwenden lässt, welche wiederum zwangsläufig in einem Browser laufen muss. In der Abbildung 1.1 ist der Aufbau des bisherigen ewm-sim veranschaulicht. Er stellt eine Art gekapseltes System dar. Die Anfragen, die an den nach außen hin geöffneten Webserver geschickt werden, landen über den WebSocket Server bei einer headless Instanz von Google Chrome, in welchem wiederum eine SAPUI5-App

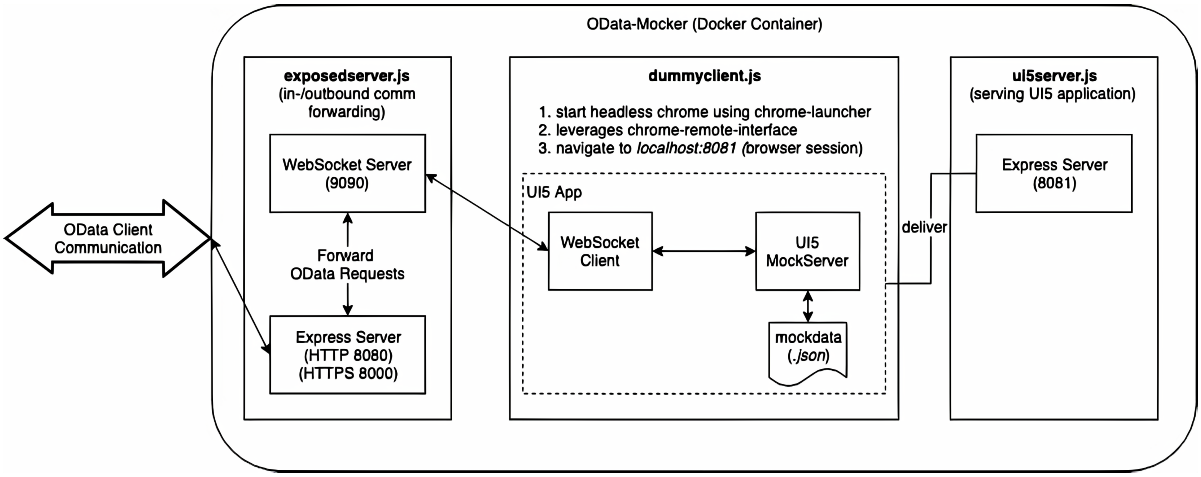

Abbildung 1.1: Aufbau des ewm-sim v1

ausgeführt wird. Diese ist mit dem SAPUI5 Mockserver verknüpft, welcher die Daten, die er bereitstellen soll, aus einer JavaScript Object Notation (JSON)-Datei einliest.

Wie gut zu erkennen ist, bringt diese Implementierung einen großen Overhead und somit mögliche Fehlerquellen mit sich. Ziel des in dieser Praxisarbeit behandelten Projekts soll es sein, das Konzept des ewm-sim zu optimieren und diesen daraufhin im Anschluss neu zu implementieren.

Verweis auf Studentenprojekt, Zusammenarbeit mit Roboter-Anbindung und Unity Simulation

2 Theoretische Grundlagen

Zur Bearbeitung dieses Projekts wurden einige Technologien eingesetzt, für die hier zunächst Vor- und Nachteile sowie theoretische Grundlagen aufgeführt werden sollen. Falls eine Entscheidung für eine entsprechende Technologie getroffen werden musste, soll diese anhand der erläuterten Kriterien nachvollziehbar dargelegt werden.

2.1 Docker

Docker ist eine freie Implementierung des Konzepts der Containervirtualisierung. Ein sogenannter Docker Container ist plattformübergreifend lauffähig. Voraussetzung ist lediglich, dass der Docker Daemon auf dem Host-System installiert ist und im Hintergrund ausgeführt wird. Damit die Container, wie beschrieben, auf alles großen Betriebssystemen laufen können, setzen sie üblicherweise auf Linux auf. Somit wird unter Windows das Windows-Subsystem für Linux (WSL) genutzt, um die Container auszuführen. Dies ist eine schlanke Integration des Linux-Kernels in das Windows-Betriebssystem, welche durch Optimierung und Reduktion auf essenzielle Bestandteile den deutlich größeren Overhead von herkömmlicher Virtualisierung erspart. [1]

Kennzeichnend für das Konzept von Docker sind die sogenannten Container. Diese stellen eine vollständige Zusammenstellung aller Komponenten dar, die eine bestimmte Anwendung zur Ausführung benötigt. Dadurch ist es sehr einfach, eine Applikation schnell und einheitlich auf einem neuen System zu deployen und dabei direkt alle Abhängigkeiten mitzuinstallieren und richtig zu konfigurieren. [2]

Docker bietet außerdem eine „Docker Hub“ genannte Plattform an. Sie bietet eine Möglichkeit, eigene Container Images (also Systemabbilder/Bauanleitungen für eine bestimmte Container-Umgebung) hochzuladen, sowie die von anderen Nutzern hochgeladenen Images zu nutzen. Diese können zum Beispiel mit einem einzigen Kommandozeilenbefehl heruntergeladen und deployed werden. Sogar als Basis für neue, eigene Images können Images von Docker Hub verwendet werden. [3]

Die angestrebte Anwendung als Docker Container zu konzipieren ist neben den oben aufgezählten Vorteilen auch vor allem deshalb naheliegend, weil die erste Version des ewm-sim bereits in dieser Form bereitgestellt wurde. Somit kann bei entsprechender Umsetzung eine identische oder zumindest sehr ähnliche Schnittstelle zur Anbindung an die anderen Softwarekomponenten des Projektes genutzt werden und Umweltfaktoren, die einen reibungslosen Betrieb verhindern würden, sind von vorneherein ausgeschlossen.

2.2 Kubernetes

Wie in 2.1 beschrieben, können mithilfe von Containern leicht Applikationen in einer neuen Umgebung eingerichtet werden. Kubernetes stellt eine quelloffene Option zur erleichterten und automatisierten Verwaltung von containerisierten Services dar. Dabei wird von Kubernetes ein deklarativer Ansatz verfolgt. Das bedeutet: Der Nutzer beschreibt den gewünschten Zustand – zum Beispiel über Konfigurationsdateien oder die Konsole – und Kubernetes ermittelt selbstständig die Schritte, die zur Erreichung und Aufrechterhaltung notwendig sind. Mittels Kubernetes ist es auch möglich, Dienste dynamisch zu skalieren (d. h. es werden abhängig von der aktuellen Anzahl der Nutzer eines Dienstes die Ressourcen einer Anwendung entweder erhöht oder wenn möglich gespart) sowie Lastausgleich und Redundanz zwischen verschiedenen Instanzen desselben Services herzustellen. [4, 5]

Instanzen eines Kubernetessystem werden auch als Cluster bezeichnet. Ein Cluster besteht aus beliebig vielen Nodes. Letztere können dabei beliebige Maschinen sein (in der Regel virtuelle Maschinen oder physische Server) auf denen die eigentlichen Anwendungen laufen. Die Interaktion mit einem Cluster findet über den sogenannten *Kubernetes Master* statt. Er stellt eine zentrale Verwaltungsstelle dar, mit den Nodes wird praktisch nie direkt interagiert. [6]

Google Kubernetes Engine (GKE) Kubernetes kann theoretisch auf jedem Heimcomputer installiert und betrieben werden. Primär findet es jedoch Anwendung im Cloud Computing. Hierbei können fertige Kubernetes Cluster gemietet werden. Dies geschieht normalerweise in Form von virtuellen Servern, deren Spezifikationen nach den eigenen Bedürfnissen gewählt werden können, ohne dass selbst Hardware angeschafft werden muss.

Somit ist es möglich, auch für kurze Projekte eine größere Infrastruktur zu deployen oder bei Bedarf und mithilfe der Skalierungsmöglichkeit von Kubernetes mit wenigen Klicks die Ressourcen einer Anwendung beziehungsweise eines Clusters anzupassen. Weiterhin ist es vorteilhaft, dass nur für die tatsächlich genutzte Rechenzeit und -last gezahlt werden muss. Google ist mit GKE einer der größeren Anbieter von Kubernetes Clustern. [7] Die Entwicklungsabteilung, im Rahmen derer dieses Projekt entwickelt wird, nutzt GKE, um dort dynamisch Testumgebungen aufbauen zu können.

2.3 Node.js

Seit der ursprünglichen Einführung im Jahre 1995 hat die interpretierte Skriptsprache „JavaScript“ rasch an Beliebtheit und Bedeutung gewonnen. Heutzutage ist sie aus der Web-Entwicklung nicht mehr wegzudenken. Node.js stellt eine Möglichkeit dar, mithilfe derer JavaScript nicht mehr nur clientseitig im Browser, sondern auch serverseitig ausgeführt werden kann. Ein großer Vorteil davon liegt darin, dass Web-Entwickler, die bereits viel mit JavaScript arbeiten und dementsprechend damit vertraut sind, nur noch eine Sprache benötigen, um sowohl Frontend als auch Backend zu entwickeln. Zudem ermöglicht Node.js die parallelisierte Ausführung von Code. Dies bedeutet, dass ein Web-Server nicht wie traditionell üblich eine Schlange von Anfragen bilden und diese nacheinander beantworten muss, sondern er die Anfragen stattdessen gleichzeitig beantworten kann.

Node Package Manager (npm) Eine weitere nützliche Funktionalität von Node.js ist der Node Package Manager. Mit ihm können sehr einfach von der Community erstellte Bibliotheken installiert und in ein Programm eingebunden werden. Auf diese Weise stehen beispielsweise fertige Frameworks für Web-Server, Unit-Tests oder Logger mit erweiterter Funktionalität zur Verfügung. npm hilft außerdem bei der strukturierten Verwaltung eines Node.js Projektes. Er stellt einen Assistenten bereit, um automatisiert eine sogenannte package.json zu generieren und auf das Projekt anzupassen. In ihr werden grundlegende Eigenschaften des Projektes verzeichnet, so zum Beispiel

- Name
- Beschreibung

- kennzeichnende Schlüsselwörter
- Lizenzen
- Autoren und Mitwirkende
- Projektwebseite und
- Quellcode-Repository.

Darüber hinaus werden hier allerdings auch die mit npm installierten Pakete inklusive deren Versionsnummer gespeichert, welche wiederum noch in allgemeine Abhängigkeiten und solche, die nur zu Entwicklungszwecken vonnöten sind, untergliedert werden. Möchte jemand anders das Projekt weiterentwickeln oder ein Deployment durchführen, so kann mithilfe von npm install im Handumdrehen die entsprechende Laufzeitumgebung dafür schaffen und Abhängigkeiten befriedigen. Des Weiteren können in der package.json auch Skripte festgelegt werden, die ein einfaches Ausführen, Testen oder Bauen des Projektes, sowie ähnliche Aufgaben ermöglichen. [8]

Die Wahl der zu verwendenden Programmiersprache fiel aus verschiedenen Gründen auf Node.js. Zunächst muss betrachtet werden, dass die simple Skriptsprache Node.js genau für den Einsatzzweck der serverseitigen Entwicklung konzeptioniert wurde. Weiterhin ist der SAPUI5 Mock Server selbst in JavaScript geschrieben. Mithilfe von einigen bereits existierenden npm-Modulen ist somit eine sehr einfache Integration mit den bereits vorhandenen Softwarekomponenten möglich. Auf Docker Hub existiert außerdem eine breite Auswahl an vorgefertigten Images für Node.js Umgebungen, sodass kein eigenständiges Containerimage von Grund auf hochgezogen werden muss.

2.4 Postman

Bei der Entwicklung eines Web-Servers ist es zu Testzwecken hilfreich, um nicht gar zu sagen unumgänglich, manuell Hypertext Transfer Protocol (HTTP)-Requests an diesen schicken zu können. Ganz grundlegende Anfragen können schon durch einen Web-Browser abgesetzt werden. Genau dies geschieht beim Aufruf jeder Website. Diese einfach im Browser abzusetzenden Anfragen stoßen jedoch schnell an ihre Grenzen, weshalb ein Programm mit erweitertem Funktionsumfang benötigt wird, bei dem detailliert Einfluss auf die Parameter von Requests genommen werden kann. Zunächst kommt ein einfaches

und auf vielen Systemen bereits vorinstalliertes Programm, welches einen solchen Funktionsumfang unterstützt, in den Sinn: *CURL*. Das konsolenbasierte Tool bietet allerdings keine Möglichkeit, Requests für spätere Verwendung zu speichern und erfordert mitunter die komplexe Kombination von Parametern, um das gewünschte Ergebnis zu erzielen. Anstelle dessen bietet sich Postman an. Das Programm stellt eine Lösung für alle oben beschriebenen Probleme von *CURL* dar. Es bietet eine übersichtliche Oberfläche, um die einzelnen Eigenschaften einer HTTP-Request bis ins kleinste Detail zu konfigurieren und ermöglicht es auch, diese zu speichern. Zur späteren Verwendung gespeicherte Anfragen können in Ordnern organisiert, mit einem Account synchronisiert oder auch als *Collection* im JSON-Format zum Austausch mit Mitarbeitern exportiert werden.

2.5 Git

Git ist ein quelloffenes Tool zur Versionsverwaltung, welches ursprünglich von Linus Torvalds zur Entwicklung des Linux-Kernels geschrieben wurde. Die klassische Bedienung erfolgt über die Kommandozeile. Mittlerweile gibt es aber auch einige grafische Bedienungsoberflächen, zudem wurde Git direkt in die gängigsten Entwicklungsumgebungen integriert. Dateien werden in sogenannten Repositories verwaltet, welchen sie durch einen *Commit* hinzugefügt oder aktualisiert werden. In einem Repository kann es zudem beliebig viele *Branches* geben, zwischen denen flexibel gewechselt werden kann. Die Dateien eines Branches sind unabhängig, d. h. es kann zum Beispiel einen Hauptbranch geben, auf dem der stabile Stand des Codes geführt wird und einen, auf dem ein neues Feature entwickelt wird. Branches können durch *mergen* ineinander überführt werden. Dies geschieht weitestgehend automatisch, es kann jedoch dabei auch zu Konflikten kommen. Beispielsweise kann dieser Fall eintreten, wenn dieselbe Zeile einer Datei in beiden Branches bearbeitet wurde. In dieser Situation muss dann manuell interveniert und der Konflikt beseitigt werden. Die Zustände des Quellcodes werden zum Zeitpunkt des jeweiligen Commits gespeichert und sind auch nachträglich noch einsehbar. So können neu hinzugekommene Fehler einfach mittels Rollbacks behoben werden.

GitHub Eine große Stärke von Git ist die Möglichkeit zur einfachen Kollaboration. Hierfür muss ein Repository auf einem Server liegen, von dem aktuelle Änderungen auf den lokalen Rechner heruntergeladen (pull) oder veröffentlicht werden können (push).

GitHub ist einer der größten Anbieter für das Hosting von Git-Repositories und bietet auf der Website noch zahlreiche weitere Möglichkeiten zum Management und der Dokumentation von Projekten. Hierzu gehören zum Beispiel ein in das Projekt integriertes Wiki, ein Tracker für Probleme (Issues), Statistiken zum Projekt und *Pull Requests* – ein Feature um Änderungen am Code anderer vorzuschlagen, auf den keine direkten Schreibrechte existieren.

2.6 Travis CI

In der professionellen Software-Entwicklung ist es üblich, automatisierte Tests (auch genannt Unittests oder Komponententests) für den Code zu schreiben, um diesen auf korrekte Funktionalität zu überprüfen. Das Tool Travis CI ist ein Dienst, der für die Durchführung genau dieser Art von Tests konzipiert wurde. In einer Konfigurationsdatei, die im Projektverzeichnis liegt, werden alle Randbedingungen (wie die genutzte Programmiersprache und das Betriebssystem, auf dem die Tests durchzuführen sind) sowie der genaue Ablauf der Tests festgelegt. Auch komplexere Abläufe, wie das Kompilieren von mehrschrittigen Builds oder parallelisierte Operationen, können mit Travis CI umgesetzt werden. Der Dienst wird dann automatisch aktiv, wenn beispielsweise neuer Code auf einem GitHub-Repository hochgeladen wird. Die Resultate der Tests können dann entweder auf der Website von Travis CI eingesehen werden oder auch als automatisch generierte Badge im Readme des GitHub-Repositorys oder auf der Projektwebsite eingebunden werden. Des Weiteren kann eine Benachrichtigung per E-Mail konfiguriert werden, sodass der für das Projekt verantwortliche direkte Rückmeldung darüber erhält, wenn etwas nach einem Update nicht mehr funktionsfähig sein sollte.

Travis CI ist eine gute und beliebte Möglichkeit, die Integrität des eigenen Codes zu garantieren. Allein deshalb schon sollte sie bei diesem Projekt zum Einsatz kommen. Ein weiterer Vorteil genau dieser Testsuite ist, dass sie bereits für EWM Cloud Robotics verwendet wird und somit die Tests des neu entstehenden ewm-sims direkt in das vorhandene Repository eingebunden werden können.

2.7 Jira

Die Abteilung in der EWM Cloud Robotics entwickelt wird, arbeitet nach der agilen Projektmanagement-Methodik *Scrum*. Dort findet Jira als Planungssoftware Verwendung. Sie erlaubt die Definition von Backlog-Items mit dazugehörigen Sub-Tasks, welche in Sprints verwaltet werden. In einem aktiven Sprint können mit wenigen Klicks Aufgaben einem Teammitglied zugewiesen oder neue Sub-Tasks erstellt werden. Die Hauptansicht zur Verwaltung ist in drei Spalten gegliedert, welche den aktuellen Status anzeigen: *To Do*, *In Progress* und *Done*. Zwischen Spalten hin und her geschoben werden Elemente eines Sprints bequem per Drag and Drop.

3 Vorbereitung

Die Abteilung, in der die Entwicklung von EWM-Cloud-Robotics vorangetrieben wird, legt einen starken Fokus auf innovative Entwicklung.

3.1 Analyse der bisherigen ewm-sim Implementierung

braucht es das hier überhaupt noch?

3.2 Einarbeitung in und Installation der Basiskomponenten

Um sich in der IT-Entwicklung mit neuen Technologien vertraut zu machen und die lokale Installation der Entwicklungsumgebung flüchtig auf korrekte Funktion zu überprüfen, empfiehlt sich in der Regel immer ein sogenanntes „Hello-World-Projekt“. Node.js und Docker stellen in dieser Hinsicht keine Ausnahmen dar und bieten beide eigenständige, geführte Einleitungen an, die den Einsteiger von der Installation bis zu den ersten sichtbaren Lebenszeichen begleiten. Beide Tools werden auch gerne in Kombination miteinander eingesetzt. Dadurch existieren praktischerweise sogar (ebenfalls von beiden Seiten aus) Anleitungen, wie ein Hello World Node.js-Server in einen Docker verpackt werden kann.

Node.js Für Node.js existieren streng genommen verschieden Ansätze für ein Hello World. Der einfachste bestünde darin, die Grußbotschaft auf der Konsole auszugeben. Da Node.js jedoch primär für den Einsatz als Webserver vorgesehen ist und die Erstellung eines Webservices auch Ziel dieser Praxisphase ist, wird hier auch das Hello World als Antwort auf eine HTTP-Request implementiert (Listing 3.1). Hierzu wird zunächst in der ersten Zeile das entsprechende HTTP-Modul aus der Standardbibliothek von Node.js geladen. Anschließend werden Hostname und Port gesetzt, unter denen der Service erreichbar sein wird. Als Hostname wird hier *127.0.0.1* in Verbindung mit Port 3000 gewählt. Die angegebene Internet Protocol (IP)-Adresse steht für den lokalen

Rechner. Üblicherweise würde für einen HTTP-Webservice der Port 80 gewählt werden. Dies bringt jedoch den Nachteil mit sich, dass die Anwendung in diesem Falle nur mit Administratorrechten ausgeführt werden kann, weshalb häufig auf alternative Ports zurückgegriffen wird. Als Nächstes wird in Zeile 6 folgende der Server initialisiert. Mit dem hier vorliegenden Aufruf der Methode *createServer* wird ein Server erstellt, der beim Aufruf jedes Pfades die folgende Callback-Funktion ausführt. Diese erhält als Parameter eine Request (eingehende Daten vom anfragenden Client) und eine Response (Antwort, die der Server an den Client zurücksendet). In den folgenden Zeilen wird zunächst der Statuscode und Inhaltstyp (hier der Einfachheit halber Klartext) der Response gesetzt und anschließend in Zeile 9 noch ihr Text „Hello, world!“ festgelegt. Mit dem Methodenaufruf in Zeile 12 wird der Server nun gestartet. Abschließend wird noch in Zeile 13 die Statusmeldung geloggt, dass der Server erfolgreich gestartet wurde und nun unter der Adresse *http://127.0.0.1:3000/* zu erreichen ist. (Die Festlegung des Inhalts dieser Statusmeldung erfolgt dynamisch und passt sich den in Zeile 3 und 4 getroffenen Einstellungen an.)

```
1  const http = require('http');
2
3  const hostname = '127.0.0.1';
4  const port = 3000;
5
6  const server = http.createServer((req, res) => {
7      res.statusCode = 200;
8      res.setHeader('Content-Type', 'text/plain');
9      res.end('Hello, world!\n');
10 });
11
12 server.listen(port, () => {
13     console.log(`Server running at ↙
           ↳ http://${hostname}:${port}/`);
14 });
```

Listing 3.1: Hello-World-Server in Node.js

Docker Nun gilt es, die eben erstellte Hello-World-Applikation in einen Dockercontainer einzubauen. Docker werden mithilfe eines sogenannten *Dockerfile*s generiert, welches

hier exemplarisch dargestellt ist (Listing 3.2). Auf Docker Hub stehen bereits dutzende Images für den Einsatz als Node.js-Server bereit. In der ersten Zeile des Dockerfiles wird mit dem Schlüsselwort *FROM* das Image referenziert, das dem neu Entstehenden als Fundament dienen soll. Im konkreten Fall heißt das Image „node“. Der hinter dem Doppelpunkt folgende Teil wird als Tag bezeichnet. Mit seiner Hilfe wird eine spezielle Version des Images referenziert, hier „14-alpine“. 14 steht für die zum Zeitpunkt dieses Projekts aktuelle Node.js-Version. Wie in der Einführung in die theoretischen Grundlagen bereits erwähnt, bauen Dockercontainer auf Linux-Systemen auf. Alpine ist eine äußerst leichtgewichtige Linux-Distribution. Dies bringt zwar unter Umständen einen verringerten Funktionsumfang mit sich, welcher allerdings bei einem stark spezialisiertem Nutzungsfall wie einem Node-Server innerhalb des Containers fast nie problematisch wird und theoretisch manuell um die benötigten Pakete erweitert werden kann, was jedoch Arbeit und Detailwissen voraussetzt. Stattdessen kann durch die Verwendung von Alpine als Basissystem jedoch die Größe des Images um ein Vielfaches reduziert werden. [9] Ist das Basisimage gewählt, wird im nächsten Schritt das Arbeitsverzeichnis innerhalb des entstehenden Containerimages festgelegt (Zeile 2), in welchem anschließend die Projektdateien eingerichtet werden. Zunächst werden hierfür die *package.json* und *package-lock.json* kopiert (Zeile 3) und gemäß den in ihnen enthaltenen Informationen die Abhängigkeiten für das Beispielprojekt im zuvor gewählten Arbeitsverzeichnis installiert (Zeile 4). Nun werden in Zeile 5 noch die restlichen Dateien in das Image kopiert. Da die Applikation nun in einem Container laufen wird und diese grundsätzlich unabhängig vom Host-System sind – auch seitens ihrer Netzwerkfunktionalität – muss in Zeile 6 noch der im Node-Server gewählte Port für die Weitergabe nach außen hin eingerichtet werden. Abschließend bleibt nur noch festzulegen, wie der Node-Server gestartet werden kann, was in der siebten Zeile erfolgt. Durch das *CMD*-Schlüsselwort, welches ein Array an Strings entgegennimmt, wird das benötigte Startkommando hinterlegt, welches beim Start des Containers auf dessen Linux-Kommandozeile ausgeführt wird.

```
1 FROM node:14-alpine
2 WORKDIR /usr/src/app
3 COPY package*.json ./
4 RUN npm install
5 COPY . .
6 EXPOSE 3000
```

```
7 CMD [ "node", "hello-world.js" ]
```

Listing 3.2: Integration des Hello-World-Servers in einen Dockercontainer

.dockerignore Im vorangegangenen Paragraphen wurden die Node-Module, die als Abhängigkeiten benötigt werden, innerhalb des Containers neu installiert, anstatt aus dem Arbeitsverzeichnis dorthin kopiert zu werden. Dass dieser Ansatz zu bevorzugen ist, hat verschiedene Gründe. Zum einen wird so gewährleistet, dass man zum Bauen des Dockerimages aus dem Quellcode keine eigene Installation von Node.js benötigt; um alle relevanten Installationsschritte kümmert sich schließlich das Node-Dockerimage. Zum anderen ist es auch einfach praktisch, da die Node-Module üblicherweise aus sehr vielen, sehr kleinen Dateien bestehen. Solche zu kopieren dauert zumeist sehr lange. Um nun zu verhindern, dass beim Schritt „den verblieben Rest in das Image kopieren“ unbeabsichtigt doch der Modul-Ordner oder andere unliebsame Dateien in das Image kopiert wird, kann eine *.dockerignore*-Datei angelegt werden. Alle in dieser Datei aufgezählten Ordner und Dateien werden dann vom Kopierbefehl ignoriert.

3.3 Suche nach einer geeigneten Basis

Kernaufgabe des ewm-sim ist die Simulation der OData-Schnittstelle eines vollständigen SAP-EWM-Systems. Grundsätzlich existieren viele Wege, über die dies erreicht werden kann. OData ist ein HTTP-basiertes Protokoll, welches eine offene Spezifikation darstellt. Also solche bieten sich grundsätzlich zwei mögliche Vorgehensweisen. Zum einen bestünde grundsätzlich die Möglichkeit, von Grund auf einen neuen Service zu implementieren, bei dem für sämtliche Requests, die an den Server gestellt werden können, das entsprechende Verhalten und mögliche Antworten abgedeckt werden müssen. Auf diese Weise wären der Entwicklung die größtmöglichen Freiheiten eingeräumt, brächte allerdings auf der anderen Seite auch eine Reihe von potenziellen Problemen und vermeidbaren Fehlern mit sich, da der OData Standard relativ umfangreich ist. Hinzu kommt, dass die Erfahrung in dieser ersten Praxisphase noch stark eingeschränkt war, weshalb dies wahrscheinlich zu einer Überforderung geführt hätte, das Projekt vermutlich nicht in der gesteckten Zeit von zwei Monaten vollständig umsetzbar gewesen wäre und sich somit insgesamt ein alternativer Ansatz empfiehlt.

Diese zweite Möglichkeit besteht in vorgefertigten Modulen, die sich speziell der Aufgabe widmen, eine einfache OData-Schnittstelle nachzubilden. Einige dieser Module finden sich beispielsweise in den npm-Repositories, wobei sie je nach Variante einen stark variierenden Funktionsumfang bieten. Da OData ein klar definierter Standard ist, waren die Kernfunktionalitäten dieser Module sehr ähnlich aufgebaut. Unterschieden haben sie sich vor allem Hinsichtlich der Speicherung abrufbaren und empfangenen Daten. Das Format, aus welchem die Services ihre Startdaten lesen, war in allen Fällen JSON. Dies bietet sich vor allem aus dem Grund an, dass es für Menschen leicht les- und schreibbar ist, jedoch auch von Maschinen aufgrund seiner Strukturierung gut verarbeitet werden kann. Der eben angesprochene Unterschied bestand nun jedoch darin, dass nur wenige der verfügbaren Module in der Lage waren, ihre empfangenen Daten – welche zur Laufzeit zunächst im Arbeitsspeicher abgelegt werden – bei der Beendung des Programms persistent zurück in die Quelldateien zu schreiben, damit sich der Server beim nächsten Programmstart wieder in genau dem Zustand befindet, in dem man zuletzt mit ihm gearbeitet hat. Würde die Implementierung des ewm-sim dies beherrschen, so wäre sie schon sehr dicht an einem tatsächlichen EWM-System, bei dem die Daten natürlich direkt aus einer Datenbank gelesen und auch dort wieder gespeichert werden. Auf der anderen Seite entstünden auch Vorteile aus einer flüchtigen Speicherung der Arbeitsdaten. Da der ewm-sim nicht nur zu Kundendemonstrationen verwendet werden soll, sondern auch in der Entwicklung Einsatz finden wird, kann es dort – etwa für die Fehlersuche – sehr hilfreich sein, wenn das Serverprogramm bei jedem Start exakt dieselbe Ausgangssituation repräsentiert. Somit kann durch Veränderung der JSON-Dateien auch gezielt ein spezieller Startzustand erzeugt und fixiert werden.

Weiterhin spielen für den ewm-sim sogenannte Function Imports eine entscheidende Rolle. Der Kern der OData-Spezifikation umfasst in erster Linie den Abruf sowie die Aktualisierung einzelner Datensätze (Entities) welche wiederum Teil sogenannter Entitysets sind. Diese Entitysets können ebenfalls angesprochen, gefiltert und hinsichtlich bestimmter Eigenschaften selektiert werden. Function Imports bieten nun die Möglichkeit, darüber hinaus spezialisierte Anfragen an den Server zu schicken. So existiert exemplarisch im ewm-sim eine Funktion „GetNewRobotWarehouseOrder“, mittels derer sich ein Roboter eine neue Warehouse Order (WHO) zuweisen lassen kann. Auf Function Imports kann in einer Simulation keinesfalls verzichtet werden, da diese, wie am obigen Beispiel anschaulich zu erkennen, für essenzielle Funktionen benötigt werden. Die verfügbaren

Module mussten somit zunächst noch alle auf die Möglichkeit untersucht werden, ob sie die Einrichtung von angepassten Function Imports unterstützen.

Im Laufe der Tests präsentierten sich zwei Kandidaten, die zwar sehr angenehm über ein Kommandozeileninterface konfiguriert werden konnten und sogar persistente Speicherung unterstützten, jedoch erfüllte leider keines von beiden die Anforderung der Function Imports. Von den betrachteten Möglichkeiten konnten diese lediglich im SAP eigenen *Mockserver* eingerichtet werden.

node-ui5 Nun kann der Mockserver allerdings, wie in der Einleitung bereits erwähnt, nur in einer *SAPUI5*-Umgebung ausgeführt werden. Mit dem npm-Modul *node-ui5* wurde eine Lösung, wie SAPUI5-Tools nicht nur clientseitig, sondern auch auf Node.js-Servern genutzt werden können, bereits vor einiger Zeit von dem kanadischen SAP-Entwickler Arnaud Buchholz entwickelt. Grundlegend funktioniert dieses Modul ziemlich gut. Leider bringt es jedoch übliche Probleme solcher in Eigenregie entwickelter „Ein-Mann-Projekte“ mit sich: Die Dokumentation ist nicht besonders ausführlich und zumeist sind andere Arbeitsaufgaben wichtiger, als das Aufspüren und beheben kleinerer Probleme. Diese Probleme stellten sich, wie zu erwarten, im Laufe des Projekts gelegentlich als kleinere oder größere Schwierigkeiten dar. Da dieses Modul jedoch das einig aktuell verfügbare ist, mit dem alle Anforderungen an den Mockserver erfüllt und eine deutliche Verbesserung der Performance erreicht werden können und die eigene Neuerstellung noch deutlich schwieriger geworden und weit über die Möglichkeiten dieser Praxisphase hinaus gegangen wären, wurde es trotzdem als Ausgangspunkt für dieses Projekt gewählt.

4 Implementierung

Nachdem die ersten Kontakte mit den beiden Kerntools erfolgt sind und eine geeignete Basis für das Projekt gefunden wurde, gilt es nun, dieses in die Tat umzusetzen. Wie in Abschnitt 3.2 kann auch für das richtige Projekt eine Aufteilung der Entwicklung in Erstellung des Servers und anschließende Einbindung in das Containerimage erfolgen, was insbesondere das Testen des Servers während der Entwicklung deutlich vereinfacht und mögliche Fehlerquellen eingrenzt.

4.1 Node-Server

Angesichts dessen, dass der Node-Server im Anschluss lediglich in einen Container eingeschnürt wird, ist die Erstellung des eigentlichen Servers der deutlich größere und wichtigere Bearbeitungsschritt. Wie bei jedem Projekt liegt auch hier der Einstiegspunkt im Errichten eines Grundgerüsts. Im konkreten Fall bedeutet das, das node-ui5 Modul erfolgreich in Node.js zu importieren, die richtigen Konfigurationsoptionen dafür zu finden, innerhalb des Moduls den SAP Mock Server anzusteuern und diesen unkonfiguriert für externe Anfragen erreichbar zu machen. In der Theorie klingt dies einfach. Allerdings tritt an dieser Stelle sowohl das Problem auf, dass node-ui5 kaum dokumentiert, als auch der Mock Server nicht für diesen Einsatz vorgesehen ist, wodurch auch die Hilfeseiten von SAP für diesen initialen Schritt nicht sonderlich hilfreich sind.

4.1.1 Grundgerüst des Mock Servers

Für das Node-Projekt wird zunächst die folgende package.json (Listing 4.1) angelegt. Hier wird zunächst ein Projektname mit Versionsnummer und Kurzbeschreibung angelegt, sowie in Zeile 5 der Einstiegspunkt für die Applikation definiert. Die drei, in Zeile 6 folgende festgelegten, Projekt-Abhängigkeiten stellen hier allerdings den primär wichtigen Teil dar. Das „body-parser“-Paket wird dafür benötigt, um den Inhalt von Anfragen an den Server sauber verarbeiten zu können. Mit den hauseigenen Bibliothe-

ken von Node.js ist zwar die Erstellung von Webservern bereits möglich, dieser Prozess wird jedoch vom „Express“-Framework stark vereinfacht. Nicht zuletzt wird noch das „node-ui5“-Modul eingebunden. Mithilfe der hier definierten Abhängigkeiten, muss nun lediglich noch *npm install* ausgeführt werden und die Implementierung des Mock Servers kann starten.

```
1  {
2    "name": "odata-mock-server",
3    "version": "1.0.0",
4    "description": "standalone odata mock server based on sapui5 ↙
       ↳ MockServer",
5    "main": "mockserver.js",
6    "dependencies": {
7      "body-parser": "^1.19.0",
8      "express": "^4.17.1",
9      "node-ui5": "^2.0.4"
10   },
```

Listing 4.1: Package.json für das Grundgerüst (gekürzt)

Nachfolgend wir in Listing 4.2 der Quellcode der mockserver.js dargestellt. Beginnend muss node-ui5 eingebunden werden (Zeile 1). Dies geschieht hier unter Verwendung der mitgelieferten „factory“, die einige Konfigurationsschritte beim Einbinden direkt übernimmt. Parameter, die sie dabei berücksichtigen soll, werden in den folgenden 3 Zeilen gesetzt, unter anderem die Variable myApp, als Referenz auf das Verzeichnis, in dem die mockserver.js liegt. Nachdem node-ui5 in Node eingebunden wurde, wird nun der gesamte folgende Code als Callback-Funktion der Einbindung ausgeführt. So enthält Zeile 7 bereits SAPUI-Code, welcher weitere SAPUI-Module einbindet (hier MockServer), welche in Zeile 9 einer anschließend automatisch ausgeführten Funktion übergeben werden.

Konfiguration des Mock Servers Ab Zeile 10 wird der eigentliche Mock Server initialisiert. Hierzu wird zunächst in der Variable *ms* ein neues Mock Server Objekt erstellt, welches für den gesamten Subpfad von „/“ zuständig ist. Zeile 14 folgende teilen dem Mock Server mit, wo er die Daten findet, die er simulieren soll (myApp/mockdata), wie diese strukturiert sind (myApp/metadata.xml) und ob automatisch Platzhalter für

fehlende Daten generiert werden sollen. Zeile 19 bis 21 nehmen schließlich noch die Einstellung vor, dass der Mock Server automatisch antwortet und in diese Antwort eine künstliche Verzögerung einbaut, um die Simulation realistischer zu gestalten, bevor der Mock Server in Zeile 24 gestartet wird.

Externe Freigabe Nun läuft der Mock Server zwar und wurde mit Datensätzen versorgt, die er simulieren kann, jedoch ist er noch nicht für Anfragen außerhalb der Node.js-Umgebung erreichbar. Zu diesem Zweck wird nun ab Zeile 26 noch ein zusätzlicher Webserver mithilfe des Express-Framework eingerichtet. Zunächst werden hierzu die mit npm installierten Module geladen und eine neue Express-Applikation initialisiert. Das zusätzlich installierte Modul für den body-parser wird nun als Standardparser für Anfragen an den Express-Server konfiguriert (Zeile 30 folgende). Anschließend wird eine Funktion festgelegt, die für jede Anfrage eines beliebigen Typs (GET, POST, PUT, ...) ausgeführt wird, die an den Express-Server geschickt werden (Zeile 35) und zwar sollen diese alle an den Mock Server weitergeleitet werden. Der Mock Server ist so implementiert, dass er automatisch alle Asynchronous Javascript and XML (AJAX)-Anfragen abfängt. Somit kann direkt basierend auf den Daten und Parametern der ursprünglichen Anfrage eine solche erstellt werden (Zeile 36-40). Der AJAX-Anfrage wird eine Callback-Funktion übergeben, die ausgeführt wird, wenn die Antwort des Mock Servers auf die Anfrage erfolgt ist. In dieser werden zunächst sämtliche Header-Daten aus der AJAX-Antwort in die Antwort des Express-Servers auf die externe Anfrage kopiert. Letztere wird anschließend noch mit dem Status-Code der AJAX-Response versehen und mit deren Text zurück an den externen Client geschickt (Zeile 41-51). Übrig bleibt nur noch das Starten des Express-Servers (Zeile 56).

```
1  require('node-ui5/factory')({
2      exposeAsGlobals: true,
3      resourceroots: {
4          myApp: __dirname
5      }
6  }).then( () => {
7      sap.ui.require([
8          "sap/ui/core/util/MockServer"
9      ], function(MockServer) {
10         var ms = new MockServer({
```

```
11              rootUri: "/"
12          });
13
14          ms.simulate(sap.ui.require.toUrl('myApp/metadata.xml'), ↙
                ↳ {
15              sMockdataBaseUrl: ↙
                    ↳ sap.ui.require.toUrl('myApp/mockdata'),
16              bGenerateMissingMockData: true
17          });
18
19          MockServer.config({
20              autoRespond: true,
21              autoRespondAfter: 10
22          });
23
24          ms.start();
25
26          const express = require('express');
27          const app = express();
28          const bodyParser = require('body-parser');
29
30          app.use(bodyParser.text({
31              type: '*/*'
32          }));
33
34          // forward HTTP-requests to MockServer
35          app.all('/*', function (req, res) {
36              window.jQuery.ajax({
37                  method: req.method,
38                  url: req.url,
39                  headers: req.headers,
40                  data: req.body,
41                  complete: jqXHR => {
42                      jqXHR.getAllResponseHeaders()
43                          .split('\n')
44                          .filter(header => header)
```

```
45                              .forEach(header => {
46                                  const pos = header.indexOf(':')
47                                  res.set(header.substr(0, ↙
                                        ↳ pos).trim(), ↙
                                        ↳ header.substr(pos + 1).trim())
48                              })
49                          res
50                              .status(jqXHR.status)
51                              .send(jqXHR.responseText)
52                      }
53                  })
54          })
55
56          app.listen(8080, () => {
57              console.log("express-app running");
58          });
59      });
60  })
```

Listing 4.2: Quellcode des Grundgerüsts (mockserver.js)

Literaturverzeichnis

[1] *Was ist das Windows-Subsystem für Linux? | Microsoft Docs*. URL: https://docs.microsoft.com/de-de/windows/wsl/about (Einsichtnahme: 14. 07. 2021).

[2] *What is a Container? | App Containerization | Docker*. URL: https://www.docker.com/resources/what-container (Einsichtnahme: 14. 07. 2021).

[3] *Docker Hub - Container Image Library | Docker*. URL: https://www.docker.com/products/docker-hub (Einsichtnahme: 14. 07. 2021).

[4] Bloß, A. *Containerorchestrierung mit Kubernetes - Teil 4 — x-cellent technologies GmbH Blog*. 2019. URL: https://www.x-cellent.com/blog/containerorchestrierung-mit-kubernetes-teil-4/ (Einsichtnahme: 15. 07. 2021).

[5] *Was ist Kubernetes? | Kubernetes*. URL: https://kubernetes.io/de/docs/concepts/overview/what-is-kubernetes/%7B%5C#%7Dwas-bedeutet-kubernetes-k8s (Einsichtnahme: 15. 07. 2021).

[6] *Konzepte | Kubernetes*. URL: https://kubernetes.io/de/docs/concepts/%7B%5C_%7Dprint/ (Einsichtnahme: 15. 07. 2021).

[7] *Kubernetes – Google Kubernetes Engine (GKE) | Google Cloud*. URL: https://cloud.google.com/kubernetes-engine (Einsichtnahme: 15. 07. 2021).

[8] *package-lock.json | npm Docs*. URL: https://docs.npmjs.com/cli/v7/configuring-npm/package-lock-json (Einsichtnahme: 14. 07. 2021).

[9] Bailey, D. *Selecting A Node.js Image for Docker*. 2017. URL: https://derickbailey.com/2017/03/09/selecting-a-node-js-image-for-docker/ (Einsichtnahme: 23. 08. 2021).