



Konzeptionierung und Implementierung einer verbesserten Version des Extended Warehouse Management Simulators

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:

Konzeptionierung und Implementierung einer verbesserten Version des Extended Warehouse Management Simulators

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 29. August 2021

Schiebelhut, Yannik

Abstract

- English -

The amount of robotic help in warehouses keeps increasing or work is even fully delegated to robots. But these robots must be developed, and the systems must be presented to customers prior to purchase and integration into their personal environment. Therefore, the possibility to simulate an entire environment in a performant way is very valuable.

This thesis deals with the creation of a service which simulates the OData interface of an SAP Extended Warehouse Management system in a detailed way. Such a service already exists although the current version suffers from great performance overhead due to its complex architecture. Therefore, the concept of the service should be rethought from scratch. Later on the service shall, among others, serve simulated robots in a Unity environment with sample orders.

Abstract

- *Deutsch* -

Roboter unterstützen vermehrt bei der Arbeit in Lagerhallen oder übernehmen diese gar vollständig. Doch diese Roboter müssen entwickelt und die Systeme Kunden präsentiert werden, bevor sich diese zum Kauf und Einrichtung in ihrer persönlichen Umgebung entschließen. Zu diesem Zweck ist die Möglichkeit unerlässlich wertvoll, eine komplette Umgebung performant simulieren zu können.

Diese Arbeit beschäftigt sich mit der Erstellung eines Services, welcher die OData-Schnittstelle eines SAP Extended Warehouse Management Systems detailgetreu nachbilden soll. Ein solcher Service existiert bereits, jedoch ist die aktuelle Fassung sehr aufwändig gestaltet und aufgrund ihrer komplexen Architektur mit großem Performance Overhead verbunden, weshalb die Software von Grund auf neu konzeptioniert werden soll. Dieser Service soll später unter anderem dazu dienen, in Unity simulierte Roboter mit Pseudoaufträgen zu versorgen.

Inhaltsverzeichnis

Abkürzungsverzeichnis	V
Abbildungsverzeichnis	VI
Quellcodeverzeichnis	VII
1 Einleitung	1
1.1 Was ist der EWM Simulator (EWM-Sim)?	1
1.2 Kritikpunkte der ursprünglichen Implementierung	1
2 Theoretische Grundlagen	3
2.1 Docker	3
2.2 Kubernetes	4
2.3 Node.js	5
2.4 Postman	6
2.5 Git	7
2.6 Travis CI	8
2.7 Jira	9
3 Vorbereitung	10
3.1 Einarbeitung in und Installation der Basiskomponenten	10
3.2 Suche nach einer geeigneten Basis	13
4 Implementierung	16
4.1 Node-Server	16
4.2 Erstellung des Dockerimages	30
4.3 Automatisierte Tests mit Travis CI	31
4.4 Deployment in Google Kubernetes Engine (GKE)	32
5 Performance-Test und Future Work	33
5.1 Locust	33
5.2 Future Work	35
Literaturverzeichnis	VIII

Abkürzungsverzeichnis

AJAX	Asynchronous Javascript and XML
EWM	Extended Warehouse Management
EWM-Sim	EWM Simulator
GKE	Google Kubernetes Engine
HTTP	Hypertext Transfer Protocol
IP	Internet Protocol
JSON	JavaScript Object Notation
npm	Node Package Manager
OData	Open Data Protocol
URI	Uniform Resource Identifier
WHO	Warehouse Order
WSL	Windows-Subsystem für Linux

Abbildungsverzeichnis

1.1	Aufbau des ursprünglichen EWM-Sim	2
4.1	SetRobotStatus im SAP Gateway Service Builder	22
4.2	Fehlercodes im Class Builder	23
5.1	Gemessene Performance der ersten Version des EWM-Sim	35
5.2	Gemessene Performance der zweiten Version des EWM-Sim	36

Quellcodeverzeichnis

3.1	Hello-World-Server in Node.js	11
3.2	Integration des Hello-World-Servers in einen Dockercontainer	12
4.1	Package.json für das Grundgerüst (gekürzt)	17
4.2	Quellcode des Grundgerüsts (mockserver.js)	18
4.3	Metadaten der Funktion SerRobotStatus	21
4.4	Node.js-Implementierung der Funktion SetRobotStatus	24
4.5	Hinzufügen des Request Handlers für die Funktion SetRobotStatus	26
4.6	Integration von HTTP-Authentifizierung in den Mock Server	27
4.7	Ausschnitt aus den Modultests	29
4.8	Dockerfile für den EWM-Sim	31
4.9	Konfigurationsdatei für automatisierte Tests des EWM-Sim mit Travis CI . .	32
5.1	Testbeschreibung für Locust	33

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 verschiedener 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 Simulationsumgebung 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

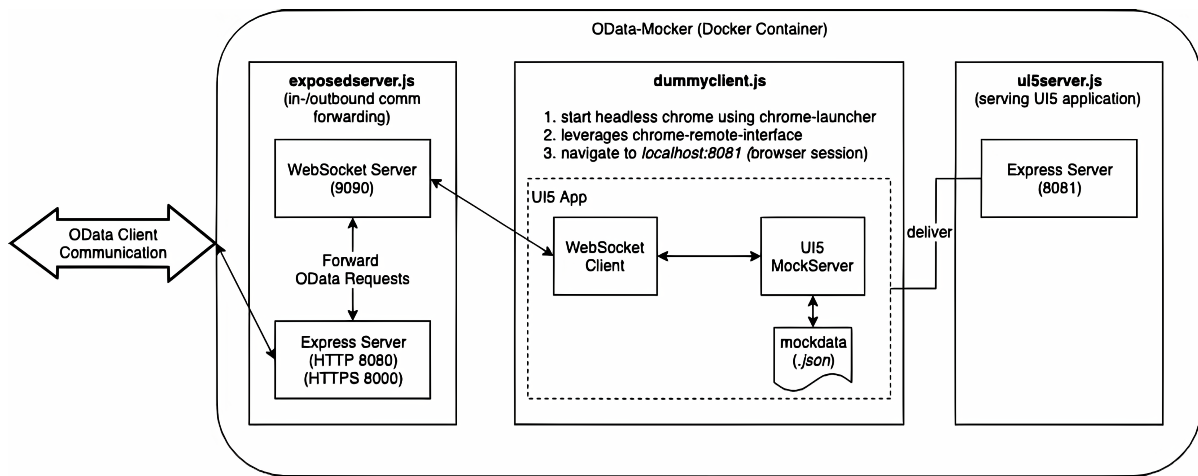


Abbildung 1.1: Aufbau des ursprünglichen EWM-Sim

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 Stu-
denten-
projekt,
Zusam-
menar-
beit mit
Roboter-
Anbindung
und
Unity Si-
mulation

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 allen 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 vornherein 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 Kubernetes-System 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]

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 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. Aus diesem Grund kommt schon dem Schritt der Vorbereitung besondere Bedeutung zu, da keine Vorgaben hinsichtlich der zu verwendenden Technologien geben wurden, abgesehen von ein paar, die Kompatibilität bedingenden, groben Rahmenbedingung und Hinweisen auf mögliche Ansätze.

3.1 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 *Dockerfiles* 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. [Bailey2017] 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.

3.2 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 Speiche-

ung 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 3.1 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 eingeschürt 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 solch einen 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 wird 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         ↵ {
16             sMockdataBaseUrl: ↵
17                 ↵ sap.ui.require.toUrl('myApp/mockdata'),
18             bGenerateMissingMockData: true
19         });
20
21     MockServer.config({
22         autoRespond: true,
23         autoRespondAfter: 10
24     });
25
26     ms.start();
27
28     const express = require('express');
29     const app = express();
30     const bodyParser = require('body-parser');
31
32     app.use(bodyParser.text({
33         type: '*/*'
34     }));
35
36     // forward HTTP-requests to MockServer
37     app.all('/*', function (req, res) {
38         window.jQuery.ajax({
39             method: req.method,
40             url: req.url,
41             headers: req.headers,
42             data: req.body,
43             complete: jqXHR => {
44                 jqXHR.getAllResponseHeaders()
45                     .split('\n')
46                     .filter(header => header)
```

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

Listing 4.2: Quellcode des Grundgerüsts (mockserver.js)

4.1.2 Function Imports

Zum aktuellen Zeitpunkt ist es bereits gelungen, einen lauffähigen OData-Service zu errichten, welcher externe Anfragen beantwortet. Damit dieser nun als neuer EWM-Sim dienen kann, muss er allerdings zunächst mit den richtigen Mockdaten versorgt werden. Zu diesem Zweck muss dem Server über die *metadata.xml* mitgeteilt werden, wie die Datenstrukturen und Zusammenhänge zwischen diesen aussehen, die er darstellen soll. Da der EWM-Sim eine genaue Nachbildung eines vollständigen EWM-Systems sein soll, sind auch die Metadaten mit einem solchen übereinstimmend. Infolge dessen kann die *metadata.xml* ohne Veränderungen direkt von einem bestehenden EWM-System übernommen werden. Auch die darzustellenden Mockdaten haben noch dieselbe Struktur wie beim EWM-Sim in der ersten Version. Um Mock- und Metadaten zu verändern, müssen die jeweiligen Dateien lediglich in die entsprechenden Verzeichnisse gelegt werden, welche in Unterabschnitt 4.1.1 konfiguriert wurden.

In den Metadaten werden für den vollständigen Funktionsumfang des Mock Servers jedoch überdies noch Function Imports definiert. Diese Funktionen stellen Anfragen dar, welche erweiterte Operationen beim Server auslösen, als ausschließlich einen Datensatz zu liefern oder zu manipulieren. Für die normalen Mockdaten benötigt der Server nicht mehr als die Daten und den entsprechenden Eintrag in den Metadaten. Bei Function Imports funktioniert das nicht so einfach, hier ist in den Metadaten lediglich festgelegt, dass eine solche Funktion zu existieren hat, auf welche HTTP-Methode diese zu reagieren hat, welche Parameter sie entgegennimmt und was sie für einen Wertetyp zurückliefert. Die eigentliche Funktionalität muss jedoch für jede Funktion einzeln von Hand implementiert werden. Wie solch eine Implementierung aussieht, soll hier am Beispiel der Funktion *SetRobotStatus* gezeigt werden.

```
1 <FunctionImport Name="SetRobotStatus" ↵
    ↵ ReturnType="ZEW_MROBCO_SRV.Robot" EntitySet="RobotSet" ↵
    ↵ m:HttpMethod="POST" sap:action-for="ZEW_MROBCO_SRV.Robot">
2   <Parameter Name="Lgnum" Type="Edm.String" Mode="In" ↵
        ↵ MaxLength="4"/>
3   <Parameter Name="Rsrc" Type="Edm.String" Mode="In" ↵
        ↵ MaxLength="18"/>
4   <Parameter Name="Exccode" Type="Edm.String" Mode="In" ↵
        ↵ MaxLength="4"/>
5 </FunctionImport>
```

Listing 4.3: Metadaten der Funktion SerRobotStatus

In Listing 4.3 ist der für die Funktion relevante Ausschnitt aus der *metadata.xml* zu sehen. Diesem kann entnommen werden, dass sich die Operation der Funktion auf das „RobotSet“ bezieht und die Funktion auf HTTP-POST-Anfragen reagieren soll. Die Funktion erwartet drei Parameter (Lgnum, Rsrc, Exccode) vom Typ String mit einer jeweils festgelegten, maximalen Länge.

Um jedoch den genauen Ablauf der Funktion im Mock Server implementieren zu können, muss die Implementierung aus dem ABAP-Backend eines richtigen EWM-Systems analysiert und in Node.js portiert werden. Dies ist überdies wichtig, da die importierten Funktionen bestimmte Fehlercodes zurückliefern, die für den Betrieb der Roboter ebenfalls essenziell sind.

```
when 'SetRobotStatus'.
  call function 'ZSET_ROBOT_STATUS'
    exporting
      iv_lgnum      = lv_lgnum
      iv_rsrc       = lv_rsrc
      iv_exccode    = lv_exccode
    importing
      es_rsrc       = ls_rsrc
    exceptions
      robot_not_found = 1
      status_not_set  = 2
      others          = 3.

case sy-subrc.
  when 0.
    move-corresponding ls_rsrc to ls_robot.
    copy_data_to_ref( exporting is_data = ls_robot
      changing cr_data = er_data ).
  when 1.
    raise exception type /iwbep/cx_mgw_busi_exception
    exporting
      textid      = /iwbep/cx_mgw_busi_exception=>business_error
      http_status_code = 404
      msg_code     = gc_error_rnf
      message      = text-001.
  when 2.
    raise exception type /iwbep/cx_mgw_busi_exception
    exporting
      textid      = /iwbep/cx_mgw_busi_exception=>business_error
      http_status_code = 404
      msg_code     = gc_error_rsn
      message      = text-004.
  when others.
    raise exception type /iwbep/cx_mgw_busi_exception
    exporting
      textid      = /iwbep/cx_mgw_busi_exception=>business_error
      http_status_code = 404
      msg_code     = gc_error_ie
      message      = text-003.
endcase. "case sy-subrc.
```

Abbildung 4.1: SetRobotStatus im SAP Gateway Service Builder

Abbildung 4.1 zeigt einen Ausschnitt aus dem *SAP Gateway Service Builder*. (Vollständige Screenshots werden aus Gründen der Lesbarkeit nicht dargestellt.) Empfängt der OData-Service den Aufruf von *SetRobotStatus*, ruft er intern im Backend die Funktion *ZSET_ROBOT_STATUS* mit den mitgegebenen Parametern auf. Die Funktion kann

zwei klar definierte und eine sonstige Exception zurückliefern. Diese Exceptions werden Exitcodes zugeordnet und anschließend wird eine Fallentscheidung anhand dieser Rückgabewerte durchgeführt. Kommt es beispielsweise zur *robot_not_found*-Exception, so wird vom OData-Service eine Antwort mit dem HTTP-Statuscode *404* zurückgegeben. Der Nachrichteninhalt wird durch die Konstante *gc_error_rnf* festgelegt. Um von dieser den eigentlichen Wert abzurufen, muss im Class Builder (Abbildung 4.2) nachgeschlagen werden. Dort steht zu jeder Konstante (Attribute) der zugehörige Wert des Fehlercodes (Initial Value).


Attribute	Description	Initial Value
GC_ERROR_RNF	 Error code: Robot not found	'ROBOT_NOT_FOUND'

Abbildung 4.2: Fehlercodes im Class Builder

Nun gilt es die Funktion in Node.js zu implementieren (Listing 4.4). Hierzu wird zunächst eine Funktion mit dem entsprechenden Namen deklariert, welche die eingehende Anfrage sowie die verwendeten URL-Parameter als Eingabewerte erhält. Zur einfacheren Verarbeitung wird aus dem zusammenhängenden String von URL-Parametern nun ein Objekt generiert, welches Schlüssel-Wert-Paare besitzt, die entsprechend die URL-Parameter und deren Werten repräsentieren (Zeile 2-8). Im nächsten Schritt wird überprüft, ob gemäß der übergebenen Parameter ein Roboter im Datensatz existiert, dessen Status gesetzt werden könnte. Ist dies nicht der Fall, muss eine *ROBOT_NOT_FOUND*-Exception zurückgegeben werden. Die Überprüfung erfolgt über eine Anfrage an das *RobotSet* desselben Servers. Aus den URL-Parametern wird der anzufragende Uniform Resource Identifier (URI) generiert und anschließend die Anfrage ausgelöst. Im Falle von Misserfolg der Anfrage wird die entsprechende Fehlermeldung zurückgegeben, ansonsten kann mit der Ausführung der Funktion fortgefahren werden (vergleiche Zeile 14-34).

In den Metadaten wurde für den Exccode eine maximale Länge festgelegt. Die Einhaltung dieser muss also überprüft werden, Überschreitung der Länge ist ein Grund zum Abbruch. Hat der übergebene Code jedoch die richtige Länge, wird erneut eine Anfrage an das servereigene *RobotSet* geschickt. Dieses Mal wird jedoch keine GET-Request geschickt, sondern mittels PUT-Request die Daten eines bestehenden Datensatzes bearbeitet. (Durch die vorangegangene Abfrage ist sicher bekannt, dass der angefragte Roboter existiert, sonst würde diese Codestelle gar nicht ausgeführt.) Als Payload werden schlicht und einfach die URL-Parameter in JSON-Form verwendet. Wird diese Anfrage

erfolgreich abgeschlossen, wird die externe Anfrage mit *200 OK* Status quittiert, andernfalls liegt ein weiterer Fehler vor. Im *SAP Gateway Service Builder* ist nur ein weiterer Fehlertyp vorgesehen. Da der Roboter an dieser Stelle erwiesenermaßen existiert, liegt nun also der *ROBOT_STATUS_NOT_SET*-Fehler vor. Dieser Fehler wird ebenfalls zurückgegeben, falls der gewünschte Status länger als die maximal erlaubte Länge sein sollte und ebenfalls in sonstigen Fehlerfällen.

```

1  var SetRobotStatus = function (oXhr, sUrlParams) {
2      logger.debug("invoking SetRobotStatus")
3      logger.debug("sUrlParams: " + sUrlParams)
4      var oUrlParams = sUrlParams.split("&").reduce(function ↵
          ↵ (prev, curr, i, arr) {
5          var p = curr.split("=")
6          prev[decodeURIComponent(p[0])] = ↵
              ↵ decodeURIComponent(p[1]).replace(/\\/g, '\\')
7          return prev
8      }, {})
9      logger.debug("oUrlParams: " + JSON.stringify(oUrlParams))
10     var uri = ""
11     var abort = false
12
13
14     // 1. Check if the robot resource exists in EWM
15     // yes: continue
16     // no: return business_error: ROBOT_NOT_FOUND
17     uri = "/odata/SAP/ZEWM_ROBCO_SRV/RobotSet(Lgnum='" + ↵
          ↵ oUrlParams.Lgnum + "',Rsrc='" + oUrlParams.Rsrc + "')"
18     logger.debug("checking if robot resource exists at: " + uri)
19     jQuery.ajax({
20         url: uri,
21         dataType: 'json',
22         async: false,
23         success: function (res) {
24             logger.debug("found that robot resource " + ↵
                ↵ oUrlParams.Rsrc + " exists")
25         },

```



```

26         error: function (err) {
27             logger.debug(JSON.stringify(err))
28             logger.debug("robot resource " + oUrlParams.Rsrc + ↵
                ↵ " does not exist")
29             oXhr.respondJSON(404, {}, { "error": { "code": ↵
                ↵ "ROBOT_NOT_FOUND" } })
30             abort = true
31         }
32     })
33     if(abort)
34         return true
35
36
37     // 2. Check if parameter "ExccodeOverall" has the right ↵
        ↵ length
38     // yes: set status for robot (and return robot?)
39     // no: return business_error: ROBOT_STATUS_NOT_SET
40     if (oUrlParams.ExccodeOverall) {
41         if (oUrlParams.ExccodeOverall.length <= 4) {
42             jQuery.ajax({
43                 url: uri, // set at 1.
44                 method: 'PUT',
45                 data: JSON.stringify(oUrlParams),
46                 async: false,
47                 success: function (res) {
48                     oXhr.respondJSON(200, {}, res)
49                     abort = true
50                 },
51                 error: function (err) {
52                     logger.debug(JSON.stringify(err))
53                     oXhr.respondJSON(404, {}, { "error": { ↵
                        ↵ "code": "ROBOT_STATUS_NOT_SET" } })
54                     abort = true
55                 }
56             })
57         }

```

```
58     }
59     if (abort)
60         return true
61
62     // else respond with error
63     oXhr.respondJSON(404, {}, { "error": { "code": 4,
64         ↪ "ROBOT_STATUS_NOT_SET" } })
65 }
```

Listing 4.4: Node.js-Implementierung der Funktion SetRobotStatus

Nun existiert zwar die Funktion, ihre Existenz ist dem Mock Server jedoch noch nicht bekannt. Um dies zu ändern, müssen zunächst die bekannten Request Handler des Mock Servers abgerufen werden. Im Anschluss kann ihnen ein neuer Handler hinzugefügt werden, der per regulärem Ausdruck den Pfad festlegt, über den seine Funktion ausgelöst wird, auf welche Art von HTTP-Requests er reagiert und welche Funktion er dann zurückliefern muss. Abschließend müssen die modifizierten Request Handler noch erneut dem Mock Server bekannt gegeben werden (siehe Listing 4.5).

```
1  var aRequests = ms.getRequests()
2  aRequests.push({
3      method: "POST",
4      path: "SetRobotStatus\\?(.*)",
5      response: SetRobotStatus
6  })
7  ms.setRequests(aRequests)
```

Listing 4.5: Hinzufügen des Request Handlers für die Funktion SetRobotStatus

4.1.3 Authentifizierung

Die OData-Schnittstelle des EWM-Sim ist dafür vorgesehen, innerhalb eines Netzwerks freigegeben zu werden. Im aktuellen Zustand wäre jeder Client in diesem Netzwerk in der Lage, nicht nur Daten von dieser Schnittstelle anzufragen, sondern diese dort auch zu manipulieren, wodurch der ordnungsgemäße Betrieb des EWM-Sim gestört werden kann. Aus diesem Grund ist es sinnvoll, eine Authentifizierung einzubauen, die den bislang offenen Zugriff beschränkt. Eine simple und verbreitete Methode hierfür ist HTTP-

Authentifizierung, welche, wie der Name schon sagt, direkt in HTTP integriert ist. Um einen Express-Server mit dieser Art der Authentifizierung zu sichern, wird lediglich ein weiteres npm-Modul benötigt – *express-basic-auth*. Des Weiteren muss die *mockserver.js* um den in Listing 4.6 dargestellten Code erweitert werden. Zunächst wird hier das Modul in den Quellcode importiert (Zeile 1). Es folgt eine Fallunterscheidung, welche überprüft, ob Zugangsdaten für den Server festgelegt worden sind. Wie im Umfeld von Containeranwendungen üblich, werden die zu nutzenden Zugangsdaten nicht in den Quellcode integriert, sondern zum Zeitpunkt des Ausführens mittels Umgebungsvariablen festgelegt. Dies sorgt zum einen dafür, dass für ein Ändern der Zugangsdaten nicht der Quellcode bearbeitet werden muss, zum anderen ist es somit auch unmöglich, den Server mit Standard-Anmeldedaten zu betreiben, was ein großes Sicherheitsmanko darstellen würde. Wurden keine Zugangsdaten per Umgebungsvariable gesetzt, bricht der Server den Start ab (Zeilen 11-13). Andernfalls wird in Zeilen 4-8 die HTTP-Authentifizierung initialisiert. Diese nutzt die *safeCompare*-Methode der importierten Bibliothek, welche bereits gegen einige mögliche Arten von Attacken abgesichert ist, und vergleicht Nutzernamen und Passwort, die der Server bei einer Anfrage erhält, mit den in den Umgebungsvariablen festgelegten. Stimmen beide Wertepaare überein, wird *true* zurückgegeben; die Authentifizierung war erfolgreich.

```
1  const basicAuth = require('express-basic-auth')
2
3  if (process.env.ODATA_USER && process.env.ODATA_PASSWD) {
4      var auth = (basicAuth({
5          authorizer: (username, password) => {
6              const userMatches = ↵
6                  ↵ basicAuth.safeCompare(username, ↵
6                  ↵ process.env.ODATA_USER)
7              const passwordMatches = ↵
7                  ↵ basicAuth.safeCompare(password, ↵
7                  ↵ process.env.ODATA_PASSWD)
8              return userMatches & passwordMatches
9          }
10     }))
11 } else {
12     logger.warn("credentials not set correctly - aborting")
13     process.exit()
```

```
14 }
```

Listing 4.6: Integration von HTTP-Authentifizierung in den Mock Server

4.1.4 Unit-Tests

Da der EWM-Sim bei der Entwicklung von Produktivsystemen eingesetzt werden soll, ist es von großer Bedeutung, dass er einwandfrei arbeitet. Um dies zu gewährleisten, ist eine umfassende Überprüfung nach jeder Änderung unerlässlich. Theoretisch könnten diese Tests händisch durchgeführt werden. Allerdings hat eine große Software wie der im Zuge dieses Projekts erstellte Mock Server einen so ausgedehnten Funktionsumfang, dass Testungen durch einen Menschen nur stichprobenartig einzelne Funktionen überprüfen könnten. Unit-Tests oder auch Modultests genannt bieten hingegen eine automatisierte Lösung des Testproblems. Softwareentwickler müssen bei dieser Variante lediglich einmal solche Tests für jede zu überprüfende Funktionalität ihres Programms schreiben. Von dort an können die Tests dann eigenständig ausgeführt werden und zeigen im Falle von Fehlern an, wo das tatsächliche Verhalten der Software vom erwarteten Ergebnis abweicht.

Modultests sind in der Softwareentwicklung sehr verbreitet, wodurch praktischerweise für eine populäre Sprache wie Node.js bereits Frameworks zur Erstellung und Durchführung solcher Tests existieren. Eines der am weitesten verbreiteten ist *Mocha* (verfügbar als npm-Modul), mit welchem auch schon die Tests für die erste Version des EWM-Sim durchgeführt wurden. Beides in Betracht ziehend fiel auch für die neue Version des EWM-Sim die Wahl auf Mocha, sodass auch einige bereits bestehende Tests übernommen werden konnten. Mocha-Tests werden einfach als normale JavaScript-Datei geschrieben und in einem „test“ genannten Ordner im Projektwurzverzeichnis abgelegt.

Listing 4.7 zeigt beispielhaften Inhalt dieser Datei. Da die Tests automatisiert laufen sollen, muss bei deren Ausführung der Mock Server ebenfalls automatisch gestartet und wieder gestoppt werden, was auch die Konfiguration der Umgebungsvariablen inkludiert und in den ersten drei Zeilen vorgenommen wird. In den Zeilen 5-7 werden für die Testdurchführung benötigte Bibliotheken geladen. Dies sind zum einen der Mock Server, eine weitere Bibliothek, die bei der Beschreibung der Testfälle benötigt wird, sowie eine aus dem ersten EWM-Sim übernommene Hilfsbibliothek, um einfach Anfragen an den Test-

server schicken zu können. Anschließend folgen die eigentlichen Tests. Testfälle in Mocha sind entfernt einer natürlichen Sprache nachempfunden. Somit wird ein Testset mit dem Schlüsselwort „*describe*“ (dt. „beschreibe“) und einer menschenlesbaren Kurzbeschreibung der nachfolgenden Tests eingeleitet (Zeile 9). Da es für automatisierte Tests üblich ist, dass gewisse Schritte vor und nach der Ausführung eines bestimmten Testfalls durchgeführt werden müssen, bringt Mocha für solches eigene Funktionen (*before* und *after*) mit. Für den vorliegenden Test erfolgt in der *before*-Funktion der Startvorgang des Mock Servers. Anschließend folgen die, durch das Schlüsselwort „*it*“ und eine genaue Beschreibung des zu prüfenden Falls gekennzeichneten atomaren Tests. Diese führen dann die zu testende Interaktion mit dem Server durch und vergleichen das erzielte Resultat (A) mittels der Funktion „*assert.deepStrictEqual(A, B)*“ mit dem erwarteten Resultat (B). Gleichen sich die Ergebnisse exakt, war der jeweilige Test erfolgreich.

```
1 process.env.ODATA_USER = "user"
2 process.env.ODATA_PASSWD = "123"
3 process.env.GEN_INT = 1000
4
5 var server = require( '../mockserver ' )
6 var assert = require( 'assert ' )
7 var tools = require( '../tools/toolbox.js ' )
8
9 describe( 'Tests for Orderroutine', () => {
10     before(() => {
11         server.initWithOrderroutine()
12     })
13
14     it( 'look if WarehouseOrderSet is initial', async () => {
15         let res = await tools.getEntity( "WarehouseOrderSet", {} )
16         assert.deepStrictEqual( res.body.d.results.length, 1 )
17     })
18
19     it( 'wait for intervall', function (done) {
20         setTimeout(() => {
21             done()
22         }, 2000)
23     }).timeout(3000)
```

```
24
25     it('check if intervall creates an order', async () => {
26         let res = await tools.getEntity("WarehouseOrderSet", {})
27         assert.deepStrictEqual(res.body.d.results.length > 1, ↵
            ↵ true)
28     })
29
30     after(() => {
31         server.stop()
32         setTimeout(() => {
33             process.exit()
34         }, 1500)
35     })
36 })
```

Listing 4.7: Ausschnitt aus den Modultests

4.2 Erstellung des Dockerimages

Nachdem der Node-Server vollständig erstellt ist und automatisiert getestet werden kann, besteht der nächste Schritt in der Einbindung in ein Dockerimage. Der Vorgang ist nur unwesentlich komplizierter als beim in Abschnitt 3.1 beschriebenen Hello-World-Docker. Als Besonderheit kommt hier allerdings dazu, dass das Zielimage möglichst klein sein soll. Aufgrund von Artefakten, die beim Generieren der Node-Module entstehen, muss hier auf ein Multi-Stage-Build zurückgegriffen werden. D. h. der Prozess der Erstellung des Dockerimages wird in mehrere Abschnitte aufgeteilt. In Listing 4.8 ist das entsprechende Dockerfile zu sehen. Die erste Buildstufe wird in den Zeilen 1-5 durchlaufen. Hierbei werden die *package.json* sowie die *package-lock.json* in das Arbeitsverzeichnis kopiert und dort mittels npm die referenzierten Pakete generiert und installiert. Anschließend wird die Buildstufe (Zeilen 7-11) gestartet. Dort werden die restlichen Projektdateien in das eigentliche Containerimage kopiert und anschließend das während der ersten Buildstufe aufgebaute Node-Modulverzeichnis aus dem ersten, temporären Image in das Zielimage kopiert. Abschließend folgt noch die standardmäßige Freigabe des Ports *8080* für den Webserver (dieser Port kann auch später noch über die Kommandozeile zur Laufzeit an-

gepasst werden) und die Festlegung der Server-Kommandozeile, welche beim Dockerstart ausgeführt werden soll, um auch den Server hochzufahren.

```
1 FROM node:lts - alpine AS build
2
3 WORKDIR /usr/src/app
4 COPY package*.json ./
5 RUN npm install --production
6
7 FROM node:lts - alpine
8
9 WORKDIR /usr/src/app
10 COPY . .
11 COPY --from=build /usr/src/app/node_modules ./node_modules
12
13 EXPOSE 8080
14 CMD [ "npm", "start" ]
```

Listing 4.8: Dockerfile für den EWM-Sim

4.3 Automatisierte Tests mit Travis CI

Wie in Abschnitt 2.6 beschrieben, ist Travis CI ein häufig verwendeter Dienst, um Modultests automatisiert ausführen zu lassen, wenn der Code im Online-Repository aktualisiert wird und somit die Integrität der Software zu gewährleisten. Für dieses Projekt wurde Travis-CI.org (inzwischen zugunsten der neuen Version Travis-CI.com eingestellte Variante) verwendet. Um Tests für ein GitHub-Repository in Travis CI einzurichten, müssen zwei Dinge erledigt werden. Zum einen muss eine Registrierung mit dem GitHub-Account, dem das Repository gehört, auf der Website von Travis CI erfolgen. Hierbei erteilt man dem Dienst Zugriffsberechtigungen für die dem Account zugehörigen Repositories. Anschließend wird einem von Travis CI eine Liste der auf dem Account gefundenen Repositories angezeigt, in der man mit nur einem weiteren Klick die automatische Verarbeitung für ausgewählte Repositories aktivieren kann.

Als weiterer Schritt muss Travis CI nun noch eine Konfiguration erhalten, wie die Tests für das registrierte Repository durchzuführen sind. Diese Konfiguration erfolgt mittels

der Datei „*.travis.yml*“, welche im Projektwurzelverzeichnis angelegt wird. In Listing 4.9 ist die Konfigurationsdatei für den EWM-Sim zu sehen. Hier wird zunächst festgelegt, in welcher Programmiersprache das Projekt geschrieben wurde und welche Version der Sprache zu verwenden ist (Zeilen 1-3). Die *install*-Sektion in Zeile 4 definiert, welche Kommandos zur Vorbereitung der Testumgebung ausgeführt werden müssen. Im konkreten Fall müssen beispielsweise die Abhängigkeiten des Node-Servers installiert werden (Zeile 5). Es können auch weitere Hook festgelegt werden, so zum Beispiel der in Zeile 6 aufgeführte *after_success*-Hook, der nach einer erfolgreichen Ausführung des Tests angewandt wird. Hier wird nach dem Test noch die „Coverage“ (dt. Abdeckung) bestimmt (Zeile 7). Diese gibt an, wie groß der Anteil des Quellcodes ist, welcher von den Tests abgedeckt wird. Eine geringe Coverage kann somit darauf hindeuten, dass nicht ausreichend Testfälle definiert wurden, um das korrekte Verhalten der Applikation zu verifizieren. Ebenfalls in der Konfigurationsdatei festgelegt werden Benachrichtigungen (Zeile 8 fort folgende). Für den EWM-Sim wurde dort eine E-Mail-Adresse angegeben, die im Falle von fehlgeschlagenen Tests automatisch benachrichtigt wird.

```
1 language: node_js
2 node_js:
3   - 14.15.0
4 install:
5   - npm install
6 after_success:
7   - npm run coverage
8 notifications:
9   email:
10    - y.schiebelhut@sap.com
```

Listing 4.9: Konfigurationsdatei für automatisierte Tests des EWM-Sim mit Travis CI

4.4 Deployment in GKE

mach ich
das über-
haupt
noch?..

5 Performance-Test und Future Work

Nach Fertigstellung der Neuauflage des EWM-Sim stellen sich nun natürlich die Fragen: „Ist es gelungen, den EWM-Sim im Vergleich zur Vorgängerversion wesentlich zu verbessern? Wenn ja, wie groß ist der Vorsprung?“ und „Wie geht es mit den Ergebnissen weiter? Was kann noch verbessert werden?“.

5.1 Locust

Die einfachste Methode, um die Performance eines Webservices zu bewerten, ist ein sogenannter Load Test. Hierbei wird eine hohe Last an Anfragen an den Service angelegt und währenddessen unter anderem protokolliert, wie lange dieser zur Beantwortung der Anfragen dauert und wie viele Anfragen dabei verloren gehen. Locust ist eine Python-Bibliothek, um solche Tests ohne großen Aufwand durchführen zu können. Man muss diese hierfür lediglich installieren und in einer kleinen Datei die möglichen Tests beschreiben, die durchgeführt werden sollen. Zum Vergleich der beiden Service-Implementierungen wurde die in Listing 5.1 dargestellte Locustdatei angelegt. Die möglichen Anfragen an den Server werden als Methoden der Klasse „ODataUser“ implementiert. Eine Methode wird durch die Annotation „@task(x)“ als Test gekennzeichnet. Über „x“ kann hierbei eine Wahrscheinlichkeit angegeben werden, mit der diese Aktion von einem simulierten User ausgeführt wird. Um die Performance des EWM-Sim zu testen, werden verschiedene Typen von Anfragen simuliert – etwa Anzeigen aller Roboter, Erstellen eines neuen Roboters, Löschen eines Roboters oder Anzeigen aller WHOs.

```
1 from locust import HttpUser, task, between
2 import time
3 import json
4
5 odataUser = "root"
6 odataPasswd = "123"
7
8 class ODataUser(HttpUser):
```

```

9      wait_time = between(0.25, 15)
10     @task(1)
11     def getHealthz(self):
12         self.client.get("/healthz")
13
14     @task(10)
15     def getWhoSet(self):
16         self.client.get(
17             "/odata/SAP/ZEWM_ROBCO_SRV/WarehouseOrderSet/?$top=20", ↵
18             ↵ auth=(odataUser, odataPasswd))
19
20     @task(10)
21     def getRobotSet(self):
22         self.client.get("/odata/SAP/ZEWM_ROBCO_SRV/RobotSet/?$top=20",
23             ↵ auth=(odataUser, odataPasswd))
24
25     @task(5)
26     def createRobot(self):
27         robot = {"Lgnum": "0815", "Rsrc": "Isaac"}
28         self.client.post("/odata/SAP/ZEWM_ROBCO_SRV/RobotSet",
29             ↵ auth=(odataUser, odataPasswd), ↵
30             ↵ json=robot)
31
32     @task(6)
33     def deleteRobot(self):
34         count = ↵
35         ↵ self.client.get("/odata/SAP/ZEWM_ROBCO_SRV/RobotSet/$count",
36             ↵ auth=(odataUser, ↵
37             ↵ odataPasswd)).json()
38
39         if (int(count) > 0):
40             robot = ↵
41             ↵ self.client.get("/odata/SAP/ZEWM_ROBCO_SRV/RobotSet/?$top=
42             ↵ auth=(odataUser, ↵
43             ↵ odataPasswd)).json()
44
45         robot = ↵
46         ↵ json.loads(json.dumps(robot))['d']['results'][0]

```

```

38         self.client.delete("/odata/SAP/ZEWM_ROBCO_SRV/RobotSet(Lgnum='0)
39                             auth=(odataUser , odataPasswd), ↵
                                   ↳ json=robot )

```

Listing 5.1: Testbeschreibung für Locust

Anschließend kann Locust per Aufruf in der Kommandozeile gestartet werden. Dies stellt ein kleines Webinterface auf dem lokalen Host zur Verfügung, über welches unter anderem die Anzahl der zu simulierenden Nutzer eingestellt und der eigentliche Load Test gestartet werden kann.

Der Load Test wurde mit 1000 simulierten Nutzern durchgeführt, welche die jeweiligen Services gleichzeitig mit Anfragen kontaktieren. Das Ergebnis ist überraschend konkret. Während die ursprüngliche Version des EWM-Sim durchschnittlich 16,6 Anfragen pro Sekunde beantwortet (Abbildung 5.1) sind es bei der neu erstellten Variante rund 104,0 Anfragen pro Sekunde (Abbildung 5.2). Zwar ist die absolute Fehlerrate bei der zweiten Version mit 4,9 Fehlern pro Sekunde deutlich höher als bei der ersten Version mit 0,5 Fehlern pro Sekunde, dies relativiert sich jedoch in Anbetracht des deutlich höheren Durchsatzes der Anfragen.

Method	Name	# Requests	# Fails	Average (ms)	Min (ms)	Max (ms)	Average size (bytes)	RPS	Failures/s
GET	/healthz	139	0	6	1	122	0	0.9	0.0
GET	/odata /SAP/ZEWM_ROBCO_SRV/	83	83	44617	30576	71669	113	0.5	0.5
GET	/odata /SAP/ZEWM_ROBCO_SRV /RobotSet	976	0	45088	4493	71886	308377	6.0	0.0
POST	/odata /SAP/ZEWM_ROBCO_SRV /RobotSet	498	0	44366	28348	71739	316	3.1	0.0
GET	/odata /SAP/ZEWM_ROBCO_SRV /WarehouseOrderSet	996	1	45319	3	71869	21088	6.1	0.0
	Aggregated	2692	84	42698	1	71886	119668	16.6	0.5

Abbildung 5.1: Gemessene Performance der ersten Version des EWM-Sim

5.2 Future Work

Trotz der nachweislich besseren Performance kam es bei der längeren Verwendung der neuen Version des EWM-Sim zu Probleme. Diese bestehen darin, dass eine Funktionalität endlos neue Daten generiert. Langfristig führt dies jedoch dazu, dass sich der

Method	Name	# Requests	# Fails	Average (ms)	Min (ms)	Max (ms)	Average size (bytes)	RPS	Failures/s
GET	/healthz	411	18	1074	1	7244	0	3.9	0.2
GET	/odata	433	21	1537	4	8753	228	4.1	0.2
GET	/SAP/ZEWM_ROBCO_SRV/ /odata /RobotSet	4007	190	1598	5	11473	261105	38.1	1.8
POST	/SAP/ZEWM_ROBCO_SRV/ /odata /RobotSet	2041	92	1569	4	9056	373	19.4	0.9
GET	/SAP/ZEWM_ROBCO_SRV/ /odata /WarehouseOrderSet	4060	197	1545	4	9023	1430	38.6	1.9
	Aggregated	10952	518	1551	1	11473	96139	104.0	4.9

Abbildung 5.2: Gemessene Performance der zweiten Version des EWM-Sim

Arbeitsspeicher viel zu stark ausgelastet wird, was schlussendlich zu einem Absturz des Simulators führen kann.

Zukünftig ist also auf jeden Fall dieser Missstand noch zu beheben. Generell war im Rahmen dieser Projektarbeit keine Zeit für ausreichende Langzeittests oder eine Verknüpfung mit den Systemen, die später auf den EWM-Sim zurückgreifen sollen. Diese Schritte müssen somit mit gebotener Sorgfalt durchgeführt werden, um auch langfristig einen reibungslosen Betrieb zu gewährleisten.

Des Weiteren wird momentan mit node-ui5 auf ein nicht aktiv entwickeltes und gepflegtes Produkt zurückgegriffen, welches überdies die Komponenten von SAPUI5 in einer Art und Weise verwendet, welche von SAP nie vorgesehen war. Die aktuelle Implementierung auf Basis von node-ui5 eignet sich zwar schon deutlich besser, als die ursprüngliche, welche innerhalb eines headless Browsers lief, jedoch ist weiterhin ein Overhead vorhanden. Aus diesen Gründen sollte es langfristig in Betracht gezogen werden, die node-ui5-Basis durch eine eigene OData-Implementierung zu ersetzen.

Allerdings muss bei alldem auch betrachtet werden, dass es sich bei dem EWM-Sim eben um eine Simulationsumgebung handelt und nicht um ein Produktivsystem. High Level Performance ist somit nicht kritisch und aufgrund des erheblichen Arbeitsaufwands, welcher mit den vorgeschlagenen Verbesserungen verbunden ist, ist die Umsetzung der meisten davon vermutlich nicht sinnvoll.

Der vollständige Quellcode des Projekts kann unter <https://github.com/SAP/ewm-cloud-robotics/tree/0353baf3532041ceffcc4243ae52d78a467d5cf/docker/ewm-sim> (Abgabezeitpunkt) und

<https://github.com/SAP/ewm-cloud-robotics/tree/master/docker/ewm-sim> (aktueller Entwicklungsstand) eingesehen werden.

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).