Ausarbeitung zum Projekt Investment Management System (IMS)

Seminararbeit

vorgelegt am 29.04.2020

Fakultät Wirtschaft

Studiengang Wirtschaftsinformatik

Kurs Verteilte Systeme

von

Maxim Bauer
Lena Teresa Simon
Lukas Spieß

DHBW Stuttgart:
Benjamin Salchow

Inhaltsverzeichnis

Abkürzungsverzeichnis

API	Application programming Interface
CORS	Cross-Origin Resource Sharing
IMS	Investment Management System
JS	JavaScript
REST	Representational State Transfer
VM	Virtual Machine

Abbildungsverzeichnis

1 Einleitung

„An der Börse ist alles möglich. Auch das Gegenteil.“ Dieses Zitat des US-amerikanischen Journalisten André Kostolany führt auf das verteilte System mit dem Namen „Investment Management System“ (IMS) hin. Dieses System sollte von diesem Projektteam im Rahmen der Vorlesung „verteilte Systeme“ entwickelt werden. Die Grundidee des Onlinebrokers lautet wie folgt: Jeder Nutzer hat ein Depot und die Möglichkeit die aktuellen Aktienwerte und den aktuellen Depotwert einzusehen, Aktien zu kaufen oder Aktien zu verkaufen. Die Aktienwerte werden automatisch aktualisiert. Um die Möglichkeit des Kaufens und des Verkaufens zu realisieren, wird außerdem eine Bank eingerichtet, die die Geldabbuchungen und Geldeinzahlungen verwaltet.

Ziel des Projektes war es, ein verteiltes System aufzubauen, welches die im Kapitel 2.1 beschriebenen Anforderungen erfüllt. Dieses Ziel sollte durch den Einsatz verschiedenster geeigneter Technologien auf der einen Seite und einer sinnvollen Trennung in einzelne Komponenten und Microservices auf der anderen Seite, erreicht werden.

In der vorliegenden Ausarbeitung wird das Ergebnis des Projektes vorgestellt. Dafür wird zunächst in Kapitel 2 die Architektur vorgestellt. Anschließend wird In Kapitel 3 auf die Umsetzung eingegangen. Abschließend erfolgt in Kapitel 4 eine Reflexion.

2 Architektur

Dieses Kapitel dient der Beschreibung der Architektur, die für dieses verteilte System ausgewählt wurde. Es wird insbesondere darauf eingegangen, wie die Systemkomponenten miteinander interagieren und welche funktionalen und nicht funktionalen Anforderungen sie zu erfüllen haben. Außerdem erfolgt eine Detailbeschreibung der einzelnen Komponenten.

2.1 Anforderungen

2.1.1 funktionale Anforderungen

Login: Es soll ein Login implementiert werden. Der Nutzer soll, unter Angabe seiner Anmeldedaten, Zugang zu seinem Depot erhalten.

Depotübersicht: Der Nutzer soll durch eine Tabelle eine Übersicht über den Inhalt seines Depots erhalten. Dabei sollen Gesamtwert und Tagesveränderungen der Positionen ersichtlich sein.

Aktienkurse: Es soll eine Aktienübersicht bereitgestellt werden, die die täglichen und aktuellen Kursbewegungen graphisch sowie tabellarisch abbildet.

Kaufprozess: Es soll dem Nutzer möglich sein, Aktien zu dem aktuellen Kurs zu kaufen. Dabei muss jedoch darauf geachtet werden, dass der Wert der gekauften Aktien nicht den Wert des Kontostands übersteigt.

Verkaufsprozess: Es soll dem Nutzer möglich sein, Aktien, die er im Depot hält, zu verkaufen. Hierbei soll darauf geachtet werden, dass er nicht mehr Aktien verkaufen kann, als er im Besitz hat.

Datenbeschaffungsprozess: Es sollen in regelmäßigen Abständen aktuelle Marktdaten geladen werden.

2.1.2 nicht funktionale Anforderungen

Transparenz: Der Nutzer soll die Applikation als eine Schicht wahrnehmen

(Nutzer nimmt nur den Reactjs-Client wahr, jedoch nicht das gesamte verteilte System).

Wartbarkeit: Durch die strukturierte Trennung von Komponenten soll eine verbesserte Wartbarkeit gewährleistet werden.

(Komponenten mit eindeutigen Zuständigkeiten in Reactjs, eindeutige API-Schnittstellen mit getrennten Aufgaben, getrennte Aufgabengebiete).

High Availability: Es soll eine möglichst hohe Service Availability gewährleistet werden.

(z.B. das Auslagern der Datadelivery.js. Stürzt dieser Microservice ab, führt dies nicht zum kompletten Absturz des Systems, wie dies bei einem Monolithen der Fall wäre).

Flexibilität/Unabhängigkeit: Durch eine eindeutige Trennung soll eine hohe Flexibilität gewährleistet werden. (z.B. Einführung einer Influxdatenbank + Datadelivery.js, anstelle die Alphavantage Schnittstelle direkt zu integrieren → bei einem Ausfall der Schnittstelle kann einfacher auf andere Anbieter gewechselt werden)

Dynamik/Performance: Das System soll insgesamt eine hohe Performance, bei gleichzeitig niedrigen Ressourcenverbrauch, besitzen. Änderungen sollen dem Nutzer direkt und ressourcenschonend angezeigt werden. (Reactjs zeichnet sich durch hohe Performance aus)

Nebenläufigkeit: Es soll gewährleistet werden, dass mehrere Nutzer gleichzeitig das System nutzen können. Dabei sollen verschiedene Anweisungen zur gleichen Zeit verarbeitet werden können.

2.2 Aufbau des verteilten Systems

Folgende Abbildung 1 zeigt den modularen Aufbau des verteilten Systems.

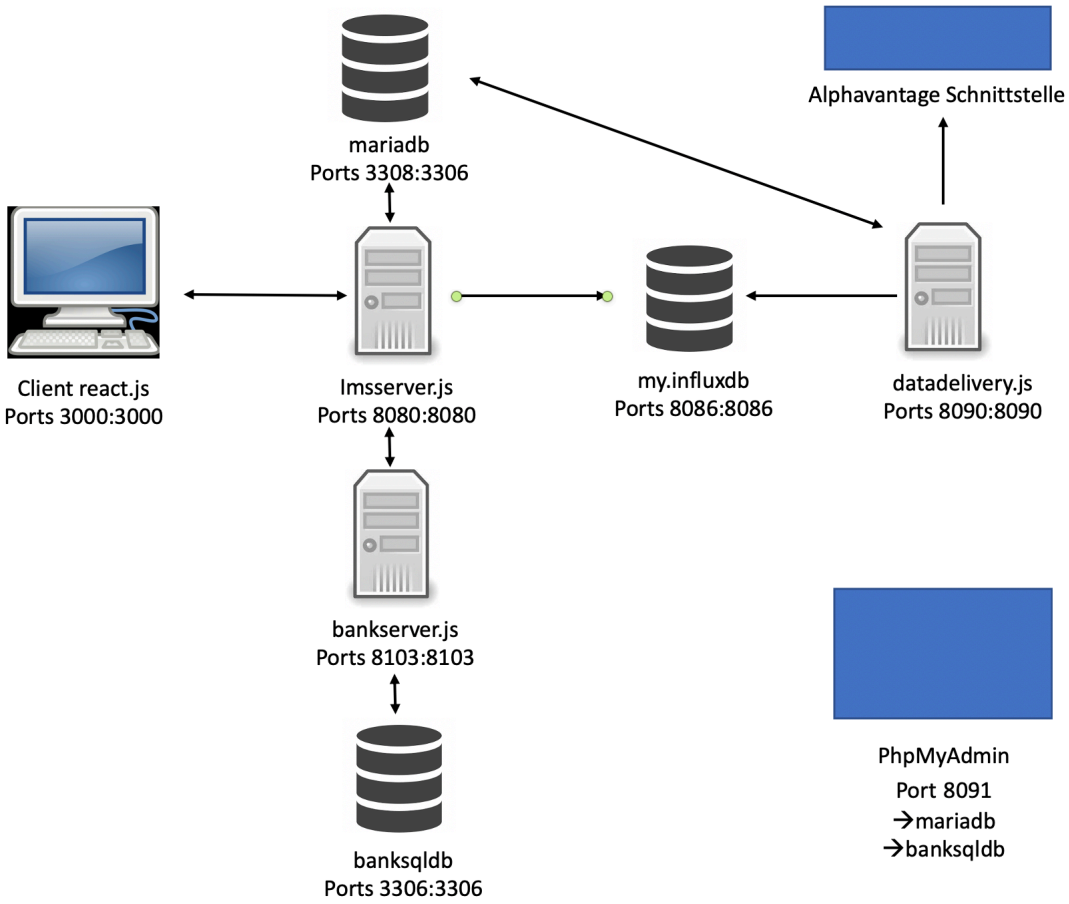

Abbildung 1: Aufbau des verteilten Systems

Für dieses Projekt war die Verwendung einer Microservicearchitektur angedacht. Unter einer Microservicearchitektur kann ein Architekturstil verstanden werden, der als Sammlung von

unterschiedlichen unabhängig deploybaren Services strukturiert ist[1]. Die Verwendung einer Microservicearchitektur bietet viele Vorteile. Auf einige wird im Folgenden eingegangen. Ein bedeutender Vorteil ist die Modularisierung, die dazu führt, dass Komponenten leicht ersetzbar sind[2]. Darüber hinaus ist es leichter, verschiedene Technologien für verschiedene Komponenten zu verwenden. Dadurch können Vorteile unterschiedlicher Technologien ausgenutzt und Nachteile von Technologien begrenzt werden. Auch bieten Microservices in Themen Skalierbarkeit und Ausfallsicherheit deutliche Vorteile gegenüber Monolithen. Die Frage, ob die Architektur des IMS die Eigenschaften einer Microservicearchitektur erfüllt, wird im weiteren Verlauf dieses Abschnittes geklärt.

Jede in der Abbildung 1 dargestellte Komponente läuft in einem eigenen Docker-Container. Dies bringt verschiedene Vorteile mit sich, auf einige wird im Folgenden eingegangen. Im Vergleich zu großen Servern verbrauchen Container wesentlich weniger Ressourcen[3]. Im Vergleich zu virtuellen Maschinen sind Docker-Container extrem schnell bereitgestellt[4]. Darüber hinaus sind Container optimal für die Aufteilung eines Systems in getrennte Komponenten bzw. den Aufbau einer Microservicearchitektur[5] und damit flexibel einsetzbar. Docker-Container bieten damit eine praktische Kapselung und sind betriebssystemübergreifend einsetzbar[6]. Da dies die Erfüllung der nicht funktionalen Anforderungen Flexibilität/ Performance/ und Wartbarkeit (besser aufgrund klarer Trennung) begünstigt, ist die Entscheidung auf den Einsatz des Containermodells gefallen.

Da jede Komponente des Systems in den folgenden Abschnitten detailliert beschrieben wird, beschränkt sich die folgende Auflistung auf die grundlegenden Aufgaben der Komponenten:

- Imsserver.js : Zentrale Komponente des Systems, bietet 4 essentielle Schnittstellen an, und koordiniert zahlreiche Softwareprozesse
- Bankserver.js: simuliert eine Geschäftsbank, kümmert sich um Zahlungsabwicklung
- Datadelivery.js: Aufgabe ist die regelmäßige Beschaffung von Aktiendaten aus einer öffentlich zugänglichen Schnittstelle
- Mariadb: Relationale Datenbank für den Imsserver.js
- Influxdb: Zeitreihenbasierte Datenbank. Zentrale Aufgabe ist die Aufbewahrung von Aktiendaten
- Banksqldb: Relationale Datenbank für Bankserver.js

[1] Vgl. Richardson o.J.
[2] Vgl. Momot 2016
[3] Vgl. Joos 2019
[4] Vgl.Buchanan/Rangama/Bellavance 2020, S. 5
[5] Vgl. Buchanan/Rangama/Bellavance 2020, S. 5
[6] Vgl. Buchanan/Rangama/Bellavance 2020, S. 2

Die Interaktion der Systemkomponenten soll anhand folgender Softwareprozesse veranschaulicht werden (eine detailliertere Beschreibung findet sich in der Beschreibung der einzelnen Komponenten).

Loginprozess

Bei einem Loginversuch stellt der Client react.js einen Request an den Imsserver. Unter Einsatz eines Datenbankkonnektors werden benötigte Daten bei der Mariadb abgefragt und anschließend an den Client weitergeleitet, wo eine Validierung erfolgt (nicht praxistauglich).

Kaufprozess

Bei einem Kaufprozess stellt der Client eine Anfrage an den Imsserver. Nach Erhalt des Requests wird eine Abbuchungsabfrage an den Bankserver gestellt. Ist das Konto ausreichend gedeckt, bestätigt der Bankserver die Abbuchungsabfrage und aktualisiert mittels eines Datenbankkonnektors die Banksqldb. Nach Erhalt der Bestätigung von dem Bankserver wird der Kauf in die Datenbank aufgenommen und der Depotinhalt des Nutzers aktualisiert. Anschließend erhält der Nutzer die Rückmeldung, ob der Kauf erfolgreich war. Ist das Konto nicht ausreichend gedeckt, erhält der Nutzer eine geeignete Fehlermeldung.

Verkaufsprozess

Bein einem Verkaufsprozess stellt der Client eine Anfrage an den Imsserver. Nach Erhalt des Requests wird eine Gutschreibungsabfrage an den Bankserver gestellt. Der Bankserver nimmt anschließend die Gutschreibungsabfrage entgegen und aktualisiert den Kontostand. Der Imsserver erhält anschließend vom Bankserver die Bestätigung der Gutschreibung und aktualisiert seinerseits die Mariadb (Depotinhalt und Verkaufstabelle werden aktualisiert), und der Nutzer wird über den erfolgreichen Verkauf und seinen neuen Kontostand informiert.

Datenbeschaffungsprozess

Mit diesem Prozess ist speziell die Einspeisung der Aktiendaten durch die Datadelivery.js gemeint. Datadelivery.js fragt bei der Mariadb mittels Datenbankkonnektor die angebotenen Aktien ab. Anschließend wird für jede angebotene Aktie ein Request an die Alphavantage-Schnittstelle gesendet. Die erhaltenen Daten werden verarbeitet und in die Influxdatenbank eingespeist.

Prozesse der Datenabfrage

Um dem Nutzer stets eine Übersicht über die Entwicklung seines Depots / aktueller Aktien zu bieten und notwendige Daten für die Transaktionsprozesse zu liefern, laufen im Hintergrund zahlreiche Softwareprozesse zur Datenabfrage ab. Dabei stellt der Client react.js zahlreiche Requests an den Imsserver. Dieser nimmt die Requests entgegen und ermittelt die angefrag-

ten Daten aus den jeweiligen Datenbanken und gibt diese als Result zurück. Die gelieferten Daten werden auf Clientseite verwendet/kombiniert und weiterverarbeitet.

Abschließend kann gesagt werden, dass das System IMS einen modularen Aufbau besitzt. Es sind Microservices für kleinere Aufgaben implementiert (bankserver.js, datadelivery.js, …). Alle Module können unabhängig voneinander deployed werden. Dies lässt den Schluss zu, dass das System die Eigenschaften einer Microservicearchitektur erfüllt. Der Umstand, dass der Imsserver.js eine große Ansammlung an Funktionalitäten besitzt und damit mehrere Aufgaben erfüllt (entspricht nicht der Definition eines Microservice), ändert nichts an dem grundlegenden Aufbau und der Einordnung des Systems als Microservicearchitektur. Jedoch kann an dieser Stelle bereits die Aussage getroffen werden, dass eine weitere Aufspaltung des Imsserver.js zu einem besseren modularen Aufbau führen würde. Dieser Punkt wird in der Reflexion nochmals aufgegriffen.

2.3 Einführung in die Komponenten

2.3.1 Imsserver.js

Der IMS-Server („imsserver.js“) ist die zentrale Komponente des Systems. Bei „imsserver.js“ handelt es sich um einen Node.js-Server. Node.js ist eine eventbasierte JavaScript Laufzeitumgebung und wurde für die Entwicklung von skalierbaren Netzwerkanwendungen entwickelt[7]. Der Server lässt sich in 2 Bestandteile einteilen. Zum einen in die Logik zum Starten und Aufbauen des lauffähigen Servers (inklusive Einfügen der Testdaten für die Mariadb) und zum anderen in die Logik für die REST API-Schnittstellen[8]. Wird das System gestartet und der Server hochgefahren, wird ein Listener auf die angegebene Adresse und Port gesetzt, wodurch es möglich wird, Anfragen gegen den Server zu senden. Anschließend wird eine Verbindung zu den verwendeten Datenbanken aufgebaut und, falls nicht vorhanden, Testdaten eingefügt. Darüber hinaus wird Axios und CORS eingebunden. CORS (Cross-Origin Resource Sharing) ist ein Mechanismus, der cross-origin HTTP-Anfragen zwischen Webanwendungen unterschiedlichen Ursprungs erlaubt[9]. Dies wird eingebunden, um die Kommunikation zwischen dem React.js-Client und dem IMS-Server zu ermöglichen. Der Server stellt 4 API-Schnittstellen bereit:

„/transaktion“

Die „/transaction“-Schnittstelle beinhaltet die Logik für einen Kauf oder einen Verkauf einer Aktie. Sind in der Abfrage alle notwendigen Daten vorhanden (UserID, Kontonummer, Betrag, Transaktionsart, Aktiensymbol, Anzahl) wird die Transaktion verarbeitet. Dabei werden

[7] Vgl. auch im folgenden: Node.js o.J.
[8] Bei einer REST API handelt es sich um eine Programmierschnittstelle die eine Kommunikation zwischen Servern und Clients gewährleisten soll
[9] Vgl. auch im folgenden: o. A. 2020a

die nötigen Prüfungen (z.B. Abfrage des Kontostands bei dem Bankserver) getätigt und die Datenbank aktualisiert. Anfragen gegen den Bank-Server werden dabei mit Hilfe von Axios gestellt. Es erfolgt eine Rückmeldung im JSON-Format. Die Rückmeldung beinhaltet die Information, ob eine Transaktion erfolgreich war.

„/login“

Die „/login“-Schnittstelle gibt die Daten des Users unter Angabe des Usernames zurück. Die Logik, ob der Login erfolgreich ist, wird anschließend beim Client durchgeführt. Den Login beim Client zu validieren, stellt ein großes Sicherheitsrisiko dar. Das Sicherheitsrisiko wurde aus Zeitgründen nicht mehr behoben. Die Rückmeldung erfolgt im JSON-Format.

„/fetch“

Die „/fetch“-Schnittstelle bedient alle Anfragen bezüglich der Aktienkurse. Unter Angabe des Zeitintervalls und der Aktie werden nachfolgende Daten verschickt. Je nach Angabe des Zeitintervalls werden unterschiedliche Werte im JSON-Format zurückgemeldet:

-Array mit allen Daten des RealtimeShares-Measurements der letzten 150 Tage

-Array mit allen Daten des DailyShares-Measurements der letzten 4 Tage

-Objekt mit aktuellem Wert und Tagesänderung der Aktie

Das Intervall der 2. Anfrage führt an bestimmten Zeiten zu einem Bug, der in der Anwendungsanleitung näher erläutert wird. Um den Bug zu beheben, müsste eine Prüfung auf Wochentag und Tageszeit erfolgen. Dies wurde aufgrund von Zeitgründen unterlassen.

„/fetch_depotinhalt“

Diese Schnittstelle gibt unter Angabe der DepotID den Depotinhalt im JSON-Format zurück. Wie in Abb. 04 zu sehen ist, handelt es sich hier jedoch ausschließlich um die Anzahl und das Symbol der Aktien.

2.3.2 Datadelivery.js

Dieser Microservice kümmert sich um die regelmäßige Beschaffung von Aktiendaten. Datenlieferant ist Alphavantage. Alphavantage ist ein Provider von kostenlosen API-Schnittstellen für Echtzeit- und historische Daten[10]. Je nach API-Schnittstelle liefert Alphavantage unter Angabe des amerikanischen Aktiensymbols und Zeitintervalls ein JSON zurück[11]. Um die Schnittstelle nutzen zu können, wurde ein eigener API_Key beantragt. Die Nutzung des API_Key´s ist jedoch auf 5 Requests die Minute bzw. 500 Requests am Tag beschränkt[12]. Aufgrund dessen stehen im Programm ausschließlich 2 Aktien (IBM / SAP) zur Verfügung.

[10]Vgl. AlphaVantage 2020
[11]Vgl. Alphavantage 2020b
[12] Vgl. Alphavantage 2020a

Die Kurse werden in Dollar angegeben. Auch bei der späteren Nutzung der Daten wird das ursprüngliche Datum verwendet und auf eine Zeitkonvertierung verzichtet. Bei datadelivery.js handelt es sich ebenfalls um einen Server, der auf Node.js basiert ist. Dieser Server läuft nach dem Start unter der angegebenen Host- und Portadresse. Beim Start wird eine Verbindung zur Influxdatenbank und zur Mariadb aufgebaut. Beim erstmaligen Aufbau des Systems wird in der Influxdatenbank eine Datenbank erstellt (inklusive Erstellung von zwei Retention Policys, dies wird in Abschnitt 2.3.4 näher erläutert). Darüber hinaus wird in der Mariadb die Tabelle „sharesymbols“ erstellt und mit Testdaten befüllt. Mit der deklarierten Funktion loadJSON(file,callback) lässt sich ein XMLHttpRequest stellen. Um Aktiendaten in die Influxdatenbank einzuspielen, werden zunächst die Aktiensymbole der Tabelle „sharesymbols“ abgefragt und anschließend für jedes Symbol ein Request gestellt. Das erhaltene JSON wird anschließend in einer For-Schleife durchlaufen, ein Array mit IPoints[13] erstellt und diese abschließend in die Influxdatenbank geschrieben. Dieser Prozess wird für historische Daten im Tagesintervall (Tagesintervall wurde ausgeklammert) und für Echtzeitdaten im Minuteninterval durchlaufen.

2.3.3 React.js-Client

React.js ist eine ursprünglich von Facebook eingeführte JavaScript-Softwarebibliothek zum Aufbauen von interaktiven User-Interfaces[14]. React.js basiert hier auf der Verwendung von sogenannten Komponenten, in denen eine render()-Methode implementiert ist[15]. Herkömmliche Webapplikationen müssen bei einer Änderung ein komplettes Rerendering durchführen, während bei einer Reactanwendung ausschließlich die render()-Methode der betroffenen Komponente aufgerufen werden muss. Dies führt zu einer deutlich besseren Performance. Generell führt das Herunterbrechen der Funktionalität auf Komponenten zu einer deutlich besseren Übersichtlichkeit. Im Folgenden wird die Komponentenstruktur des IMS-Clients genauer erläutert16. Die Komponenten befindeen sich im Verzeichnis „IMSdocker/stockfolder/src/“.

Index.js

Diese Komponente ist standardmäßig vorhanden. Index.js wurde durch die Einführung eines Browserrouters erweitert, dies ermöglicht die Implementierung eines Routings in der App.js Komponente.

App.js

[13] Ein IPoint ist ein von dem Node-influx-Datenbankkonnektor standardisiertes Objektformat. Vgl. Node-Influx 2017
[14] Vgl. auch im folgenden Infopulse 2020
[15] Vgl. auch im folgenden React.js 2020
[16] Der grundlegende Aufbau einer React-App wird hier nicht genauer beleuchtet, da dies den Rahmen dieser Arbeit sprengen würde.

Diese Komponente ist standardmäßig vorhanden. In dieser Komponente wird zusätzlich ein Routing implementiert. Dies geschieht durch die Einführung von Router, Switch und Route. Je nach Pfad(‚/', ‚/Home', ‚/Stock') werden unterschiedliche Komponenten aufgerufen. Das Routing funktioniert mit Hilfe der History-Komponente.

History.js

Diese Klasse beschäftigt sich mit dem Browserverlauf, durch Veränderung der History-Komponente kann das Routing gesteuert werden.

Cookie.js

Damit wichtige Daten (z.B. UserID, DepotID) komponentenübergreifend verfügbar sind, wurde ein provisorischer Cookie, in Form eines globalen Objekts, eingeführt. Um auf die Attribute zuzugreifen, muss der Cookie importiert werden.

LoginApp.js

Diese Komponente kümmert sich um den Loginprozess. Es wird zunächst ein einfaches Login-Formular gerendert. Änderungen des Formularinhalts durch Usereingaben müssen durch eine Eventverarbeitung verarbeitet werden. Erfolgt ein Anmeldeversuch durch ein submit, wird mittels Axios ein Request an die Login-API des Servers gestellt. Anschließend wird der Anmeldeversuch validiert. Ist der Anmeldeversuch erfolgreich, werden die Daten des Cookies befüllt. Anschließend wird der Nutzer durch die Veränderung der History-Komponente, zur Homeapp.js weitergeleitet.

HomeApp.js

Diese Komponente ist einfach gehalten. Sie beinhaltet den Komponentenaufruf von Table1.js (Aktuelle Kurse) bzw. Table2.js (Deine Aktien).

Table1.js

Diese Komponente kümmert sich um die Darstellung der aktuellen Kursübersicht aller angebotenen Aktien in tabellarischer Form. Zunächst werden über die Methode fetchsymbols() alle angebotenen Aktiensymbole ermittelt. Anschließend werden für jedes Symbol mittels der Methode fetchdata() der aktuelle Kurs sowie die Tagesveränderung ermittelt. Abschließend wird der Array sharedata[] (Bestandteil des State) aktualisiert. Die Funktionen renderTableData() und renderTableHeader() kümmern sich um das dynamische Zusammenstellen einer Tabelle aus dem benannten Array. Beim Rendern einer Zeile wird ein Button „Kaufen" hinzugefügt, der den Nutzer beim Betätigen zur Stockapp.js weiterleitet und dabei das Symbol in der URL mitgibt.

Table2.js

Diese Komponente kümmert sich um die tabellarische Darstellung des Depotinhalts eines Nutzers und ist ähnlich aufgebaut wie Table1.js. Hier wird über die Methode fetchdepotinhalt() der Depotbestand eines Nutzers abgefragt. Der Depotinhalt wird mit dem Aufruf der Methode fetchdata() mit den aktuellen Kursen kombiniert und somit der Gesamtwert der einzelnen Positionen berechnet. Abschließend wird der Array des State aktualisiert. Beim Rendern einer Zeile wird ein Button „Verkaufen“ hinzugefügt. Wird der Button betätigt, wird ein Popupfenster geladen und die Attribute Aktie und Anzahl übergeben.

Popup2.js

Diese Komponente kümmert sich um den Verkaufsprozess. Der aktuelle Wert der Aktie wird abermals über die Methode fetchdata() ermittelt. Die Nutzereingabe (erlaubt sind nur Integer) wird mit dem ermittelten Wert multipliziert und ergibt den Verkaufspreis (die Nutzereingabe ist hier durch die übergebene Anzahl beschränkt). Bei einem Verkauf wird in der Methode onVerkaufClicked() ein Request an die API-Schnittstelle „/transaction“ des IMS-Servers mittels Axios übermittelt und der Nutzer über das Ergebnis informiert.

StockApp.js

Bei dieser Komponente handelt es sich um die Einzelübersicht einer Aktie und beinhaltet die Komponenten Stock.js, Table.js, und Popup.js. StockApp.js rendert darüber hinaus 2 Buttons, der Button Kaufen steuert das Popup für den Kaufprozess, der 2. Button leitet den Nutzer zur Depotübersicht weiter.

Stock.js

Diese Komponente kümmert sich um die graphische Darstellung eines Aktienkurses. Hierzu wurde die Bibliothek plotly.js importiert. Je nach gewünschtem Zeitintervall werden über die Methode fetchdata() beim imsserver.js die benötigten Aktiendaten abgefragt und dem State übergeben. Mittels der Daten rendert die importierte Komponente Plot einen Graphen. Der Switch erfolgt über 2 Buttons, die jeweils den State „time“ verändern, dadurch wird gesteuert, welche Daten abgefragt werden.

Table.js

Entspricht einer vereinfachten Form von Table1.js,

Popup.js

Diese Komponente kümmert sich um den Kaufprozess. Der aktuelle Wert der Aktie wird abermals über die Methode fetchdata() ermittelt. Die Nutzereingabe wird mit dem ermittelten Wert multipliziert und ergibt den Kaufpreis. Bei einem Kauf wird die Methode onKaufClicked() ein Post an den Server mittels Axios versenden und der Nutzer über das Ergebnis informiert.

2.3.4 Bankserver.js

Eine weitere Komponente dieses verteilten Systems ist der Bankserver („bankserver.js“). Dabei handelt es sich ebenfalls um einen Node.js-Server. Wird der Server hochgefahren, wird ein Listener auf die angegebene Adresse und Port gesetzt. Dieser Server beinhaltet zum einen die Logik zum Aufbauen der Verbindung zu einer MySQL-Datenbank mit der Database „banksqldb“, wobei die Tabellen „Kunde“ und „Konto“ aufgebaut und Testdaten eingefügt werden, falls die Tabellen nicht vorhanden sind. Zum anderen ist die Logik für die Abbuchung eines Dollarbetrags von der banksqldb im Falle eines Kaufes bzw. die Gutschreibung eines Betrags im Falle eines Verkaufes enthalten. Wird ein Kauf oder Verkauf getätigt, wird vom Imsserver eine Anfrage mittels Axios gestellt. Dabei wird sowohl der Abbuchungs- bzw. Zubuchungsbetrag, als auch die Kontonummer übermittelt. Die Anfrage wird in ein Objekt umgewandelt, so dass mittels eines Selects, der die Kontonummer enthält, eine Anfrage an die banksqldb gestellt werden kann, um den aktuellen Kontostand zu erfahren. Dies ist vor allem bei dem Kauf wichtig, um zu prüfen, ob genügend Geld auf dem jeweiligen Konto vorhanden ist. Um den Betrag abzubuchen bzw. gutzuschreiben, wird ein Update gemacht. Anschließend wird eine Nachricht im JSON-Format an den Imsserver.js zurückgemeldet, ob der Kauf bzw. Verkauf erfolgreich war oder nicht.

2.3.5 Datenbanken

Für das verteilte System wurden 3 Datenbanken verwendet, die die benötigten Daten speichern. Im Folgenden werden auf Typ, die Aufgaben und Tabellenstrukturen der Datenbanken eingegangen:

My_influxdb

Für die Verwaltung der Aktienkurse wird eine Influxdatenbank verwendet. Bei einer Influxdatenbank handelt es sich um eine Time Series Database (TSDB)[17], dies ist eine für die Verwendung von zeitreihen- bzw. zeitstempelbasierten Daten optimierte Datenbank. Das Speichern von Daten erfolgt hier durch das Erstellen von Messungen, sogenannten Measurements. Folgende Abbildung zeigt den Ausschnitt eines Measurements im laufenden Betrieb (siehe Abb. 02). Die Struktur orientierte sich hier an den von der Alphavantage Schnittstelle gelieferten Daten:

```
> select * from DailyShares
name: DailyShares
time                close    high     low      open     symbol timezone   volume
----                -----    ----     ---      ----     ------ --------   ------
1575244800000000000 132.9100 134.5000 132.4800 134.4500 IBM    US/Eastern 3066813
1575244800000000000 133.4500 134.9300 132.3500 134.9300 SAP    US/Eastern 628177
```

Abbildung 2: Ausschnitt eines Measurements

[17] Vgl. auch im folgenden Influxdata 2020

Jeder Datensatz hat damit folgende Attribute[18]:

Time – der Zeitstempel des Datensatzes (amerikanische Zeit)

Symbol – Symbol der Aktie

Timezone – Zeitzone der Börse, an der die Aktie gehandelt wird

Volume – Handelsvolumen der Aktie

Close – Schlusskurs der Aktie

Open – Eröffnungskurs der Aktie

High – Höchstwert der Aktie

Low – Tiefstwert der Aktie

Im laufenden Betrieb sind 2 Measurements (RealtimeShares/ DailyShares) vorhanden. Bei „RealtimeShares“ werden die aktuellen untertägigen Kursdaten gespeichert, hier erfolgt die Messung im 1-Minutenintervall. Bei „DailyShares“ werden die historischen Daten der letzten 100 Tage gespeichert, die Messung erfolgt hierbei im Tagesintervall. Beiden Measurements liegt die gleiche Struktur zugrunde, und sie unterscheiden sich ausschließlich im Intervall der Zeitstempel. Bei einer Influxdb lässt sich über sogenannte Retention Policys die Aufbewahrungsdauer definieren. Bei der Initialisierung der Datenbank werden 2 Retention Policys erstellt („150d“ für 150 Tage, „4d“ für 4 Tage).

Mariadb:

Für die Verwaltung von nicht zeitreihenbasierten Daten des IMS wird eine MariaDB verwendet. Bei der MariaDB handelt es sich um ein verbreitetes OpenSource-Datenbank-System, welches ursprünglich aus der MySQL-Entwicklung hervorgegangen ist[19]. Diese Datenbank zeichnet sich durch ihre hohe Kompatibilität und den Support aus[20]. Darüber hinaus ist diese bei größeren Datenmengen performanter als SQL. Um die funktionalen Anforderungen zu erfüllen (z.B. Login/ Transaktionstracking/ Depotführung) wurde folgende Tabellenstruktur entworfen (siehe Abb.03):

[18] Anmerkung: die Werte gelten für das betrachtete Intervall
[19] Vgl. auch im folgenden: o.A. 2020b
[20] Vgl. auch im folgenden Beliaev 2014

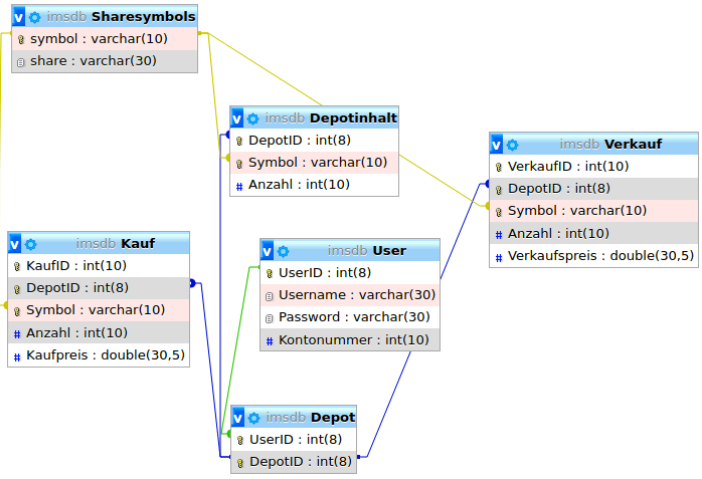

Abbildung 3: Db-Tabellenstruktur Mariadb

An dieser Stelle ist noch einmal wichtig anzumerken, dass die Tabelle „Sharesymbols“ definiert, welche Aktien beim IMS angeboten werden. Das Hinzufügen von Werten würde das Portfolio der angebotenen Aktien erweitern.

Banksqldb

Bei der simulierten Bank wurde eine frühere Version einer MySQL-Datenbank verwendet. Grund hierfür ist die fehlende Kompatibilität zwischen dem verwendeten Node.js - Datenbankkonnektor und der neusten MySQL-Version 8.0. Die Daten der simulierten Bank beschränken sich hier auf 2 Tabellen (siehe Abb. 04):

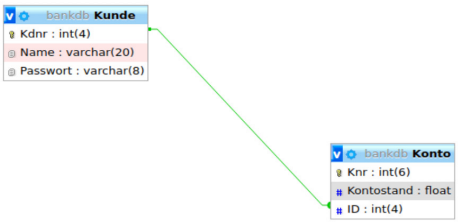

Abbildung 4: Db-Tabellenstruktur der Banksqldb

Die Verwendung einer veralteten Version einer MySQL-Datenbank geht mit Sicherheitseinbußen einher. Die Entscheidung, 3 unterschiedliche Datenbanken in das verteilte System einzubinden, erfolgte neben den beschriebenen Vorteilen unter anderem aus der Motivation heraus, den Lerneffekt durch die Verwendung unterschiedlichster Technologien zu maximieren.

3 Anwendungsanleitung

3.1 Testdaten

Im Folgenden wird auf die bereitgestellten Testdaten eingegangen, die bei der erstmaligen Initialisierung des Systems vorliegen. Folgende Übersicht zeigt, welche Daten in den jeweiligen Datenbanken hinterlegt sind:

MariaDb (für IMS):

User

UserID	Username	Password	Kontonummer
1	testuser1	securepassword	1111
2	testuser2	securepassword	2222

Depot

UserID	DepotID
1	1
2	2

Depotinhalt

DepotID	Symbol	Anzahl
1	IBM	10
2	SAP	12

Kauf

KaufID	DepotID	Symbol	Anzahl	Kaufpreis
1	1	IBM	12	1464
2	2	SAP	16	1936

Verkauf

VerkaufID	DepotID	Symbol	Anzahl	Verkaufspreis
1	1	IBM	2	244
2	2	SAP	4	484

sharesymbols

symbol	share
IBM	IBM
SAP	SAP

Abb. 02: Testdaten Mariadb

Banksqldb:

Konto

Knr	Kontostand	ID
1111	2000.99	111
2222	5000	222

Kunde

Kdnr	Name	Passwort
111	Achim Alt	111
222	Berta Brot	222

Abb. 03: Testdaten Banksqldb

Wie Abb. 01 zu entnehmen ist, sind initial 2 Testuser angelegt. Ein Login ist unter Angabe der Benutzernamen („testuser1“/ „testuser2“), bzw. Passwörter („securepassword“) möglich. Die Anzahl der Aktien, die erworben werden können, ist durch das Guthaben der hinterlegten Bankkonten beschränkt (2000.99$ bei Testuser1/ 5000$ bei Testuser2). Es besteht die Möglichkeit, die Testdaten zu verändern. Für diesen Zweck wurde die Phpmyadmin-Suite integriert. Zugang zu den Datenbanken MariaDB und Banksqldb erhält man unter der Angabe folgender Logindaten:

MariaDB: Server: „mariadb“, Username:”secureuser”, Password: “securepassword”

Banksqldb: Server: “banksqldb”, Username:”securesuer”, Password: “securepassword”

Die Influx-Db wird über die Datadelivery.js direkt mit aktuellen Daten der Alphavantage-Schnittstelle versorgt. Durch Eingabe folgender Befehle im Terminal kann auf diese Daten zugegriffen werden:

- docker exec -it my_influxdb influx
- use aktiendb
- select * from RealtimeShares/DailyShares

3.2 Was ist zu beachten?

1. Bitte beim erstmaligen Starten mindestens 2 min warten, bis alle Services hochgefahren und alle Daten (1min Intervall bei Intraday) geladen sind.
2. Es ist zu überprüfen ob alle vorgesehenen Services laufen. Falls dies nicht der Fall ist, sollten durch ein erneutes Ausführen des Befehls „docker-compose up“ alle verbliebenen Services starten.
3. Die Eingabefelder beim Kauf/Verkauf von Aktien akzeptieren nur ganze Zahlen
4. Es können nicht mehr Aktien verkauft werden, als im Besitz sind.
5. Es können nur so viele Aktien gekauft werden, deren Wert von dem hinterlegten Konto gedeckt sind
6. Es kam bereits vor, dass Alphavantage nicht zeitgemäße Daten bereitstellt

3.3 bekannte Bugs

1. Die Buttons, um zwischen Intraday- und Dailyansicht zu wechseln, müssen 2 mal betätigt werden.
2. Schlägt ein Kauf fehl, muss das Popupfenster erneut geöffnet werden

3. Werden die Intraday-Aktien für den Graphen geladen, muss das Zeitintervall (aufgrund von Wochenende/Feiertagen) mindestens 4 Tage betragen. Dies führt zu bestimmten Uhrzeiten/ nach einer gewissen Zeitspanne dazu, dass der Intraday-Graph Daten von mehr als einem Tag anzeigt.
4. Der Depotwert aktualisiert sich nach einem Verkauf erst beim Wechsel der Seite. Es war nicht bekannt, wie die Methode update() der Komponente Headerline.js in der Komponente Popup2.js aufgerufen werden kann.

4 Umsetzung

4.1 Herausforderungen

Hier ist anzumerken, dass auch größere Herausforderungen und Probleme im Zusammenhang mit dem Projekt als natürlicher Bestandteil des Lernprozesses angesehen wurden.

In diesem Abschnitt werden Herausforderungen, die während des Projekts aufkamen, beleuchtet. Anschließend wird erörtert, wie auf Herausforderungen reagiert bzw. wie Herausforderungen gelöst wurden.

Zunächst wird auf generelle Herausforderungen in Bezug auf Rahmenbedingungen und anschließend auf programmierspezifische Herausforderungen eingegangen.

4.1.1 Herausforderung bei den Rahmenbedingungen

Linux Performance:

Im Rahmen des Projektes gab es immer wieder Probleme mit der Linux Performance. Zeitweise kam es zu Abstürzen/ großen Verzögerungen. Da dies in Summe doch einiges an Zeit gekostet hat, bleibt dies hier nicht unerwähnt.

Einarbeiten in neue Technologien:

Eine große Herausforderung im Projekt, die zu nennen ist, ist das Einarbeiten in bzw. die Verwendung und Integration von vielen verschiedenen neuen, bis dato noch unbekannten Technologien. Wichtig waren unter anderem die folgenden Fragestellungen:

-Was zeichnet diese Technologie aus? (z.B. Influxdatenbank optimiert für das Arbeiten von zeitserienbasierten Daten, wie Aktien)

-Wie funktioniert die Technologie generell? (z.B. grundlegende Unterschiede InfluxDb zu normalen Relationen Datenbanken, Idee von einer single-page application)

-Einarbeiten in die Syntax: Javascript, Node.js, Axios, Reactjs, Influxdbsyntax, Datenbankkonnektoren, JSON waren grundlegend neu, das Einarbeiten und Einbinden war eine große Herausforderung.

Vernetzung von Komponenten

Die Abstimmung und Vernetzung zwischen insgesamt 7 zentralen Containern erforderte eine große Abstimmung und Kommunikation. Auch die Verwendung von Axios/JSON stellte eine große Herausforderung dar. Es musste hier geklärt werden, welche Schnittstellen und Aufgaben die einzelnen Komponenten haben sollen, und wie diese miteinander interagieren.

4.1.2 Programmspezifische Herausforderungen:

API-Schnittstelle/ Datadelivery.js

Bei der API Schnittstelle von Alphavantage musste auf die Dynamiken der amerikanischen Börsen geachtet werden. Die Schnittstelle lieferte ein JSON. Die Daten des JSON-Objekts mussten unter Berücksichtigung der Zeitverzögerung, Wochenenden/ Feiertagen und „Öffnungszeiten“ richtig in die Datenbank eingespeist werden.

Datenbanken bei Docker-Compose

Es bestand lange Zeit das Problem, dass die Javascript Container beim Starten innerhalb eines docker-compose.yml-files keine Verbindung zu den Datenbanken aufbauen konnten.

Unhandled-promise-rejections bei imsserver.js

Ein weiteres großes Problem bestand darin, dass bei Imsserver.js nach einer gewissen Zeit unhandled promise-rejections aufgetreten sind. Dies geschah bei Funktionen, die in den Minuten/Sekunden davor noch funktionsfähig waren . Hierbei musste der Imsserver.js immer wieder manuell abgebrochen und neugestartet werden, um die volle Funktionalität der Webanwendung wiederherzustellen.

Axios beim Client

Die Verwendung von Axios bei einem Client.js verlief nicht zur Zufriedenheit des Projektteams. Nachdem Axios durch einen Import eingebunden wurde, konnten für das Html-file angedachte Funktionen nicht mehr verwendet werden.

Kommunizieren von React.js und Node.js funktionierte nicht

Beim lKommunizieren zwischen Reactjs und Nodejs traten cross-side rejections auf, und eine Kommunikation über Axios war im ersten Schritt nicht möglich.

Reactjs: die bisher unbekannte Idee von einer Single-Page-Application

Die Dynamik von ReactJs mit Komponenten, dynamisches Routing und Renderfunktionalität waren für das Projektteam neu. Viele Fragen waren zu Anfang ungeklärt. Wie funktionieren Serverabfragen? Wie können bei einer Single-Page-Applikation „Seiten“ gewechselt werden? Wie werden Informationen übergeben?

4.2 Reaktion und Lösungen

Linux-Performance

Die Linux Performance hat zwar zu Zeitverlusten geführt, jedoch hat die Verwendung einer LinuxVM in Verbindung mit Docker Performance- und Kompatibilitätsvorteile. Das passive Problem wurde bewusst in Kauf genommen.

Einarbeiten in neue Technologien

Einerseits war es bei dieser Herausforderung sehr wichtig, sich hauptsächlich auf die gebrauchten Features zu konzentrieren, andererseits war es aber auch wichtig, sich einen Überblick über die Technologie als Ganzes zu verschaffen, um möglichst viele Vorteile und Möglichkeiten der Technologie zu nutzen. In Anbetracht des begrenzten Zeitraums ist aus Sicht des Projektteams hier eine gute Balance gelungen.

Vernetzung von Komponenten

Bei den vielen Komponenten des verteilten Systems war eine umfassende Teamkommunikation notwendig. Es musste genau abgesprochen werden, wie die Schnittstellen designed werden müssen, und wie der Kommunikationsfluss aussehen muss. Die entstandene Lösung basiert hier vorwiegend auf der Verwendung von JSON/REST API-Schnittstellen/ Axios und den jeweiligen Datenbankkonnektoren.

API-Schnittstelle/Datadelivery

Diese Problematik wurde mit einer umfassenden Logik gelöst, die durch Abfrage von Datum, Uhrzeit, Wochentag an jedem Tag die richtigen Daten der Alphavantageschnittstelle in die Influxdb einspeist.

Datenbanken bei Docker-Compose

Die Lösung kam mit der Erlangung eines Grundverständnisses von der Funktionsweise von Ports/Hosts in Docker. Bei dem docker-compose.yml-file mussten die Datenbanken den Javascript-containern bei dem Attribut „links:“ bekannt gemacht werden und der Datenbankhost und Port bei dem Aufbauen der Datenbankverbindungen in die dockerspezifischen Adressen und Ports umgeändert werden. Ein weiteres Problem, dass Datenbankserver zu langsam starten, wurde hier mit einem einfachen statischen sleep gelöst.

Unhandled-promise-Rejections bei imsserver.js

Die Ursache für die Problematik, dass funktionierende Funktionen nach einiger Zeit „unhandled promise rejections“ werfen, konnte nicht gefunden werden. Gelöst wurde dieses Problem mit einem Workaround. Bei diesem Workaround wurden die Möglichkeiten, die docker-compose und node.js bieten, zu Nutze gemacht: Zum einen wurde der Befehl zum Starten des imsserver.js („node“) durch die Option „--unhandled-rejections = strict“ im Dockerfile und docker-compose.yml-file erweitert. Dies sorgt dafür, dass der imsserver.js bei einem Fehlerfall automatisch abbricht. Zum anderen wurde im docker-compose-file die Option „restart: always“ hinterlegt, wodurch der Server im Fehlerfall direkt neu startet.

Axios beim Client

Im Scrum-Meeting wurde eine Transition zu Reactjs beschlossen. Trotz der Herausforderung einer neuen Technologie und dem damit verbundenen Einarbeitungsaufwand funktionierte hier die Kommunikation mittels Axios nach der Einbindung von CORS einwandfrei.

Kommunizieren von Reactjs und Nodejs funktionierte nicht

Bei den ersten Versuchen einer Kommunikation mittels Axios zwischen React.js und Node.js kam es zu einer „cross-origin -rejection“. Dies rührte daher, dass standardmäßig eine Kommunikation zwischen React.js und Node.js aus Sicherheitsgründen nicht erlaubt ist. Durch die Einbindung von CORS konnten Cross Origin Requests auf den Imsserver.js erlaubt werden.

5 Reflexion

5.1 Was hat gut funktioniert?

Im Folgenden werden einige Punkte beleuchtet, die aus Sicht des Projektteams gut funktioniert haben:

GitHub

Die Verwendung von GitHub verlief jederzeit problemlos; durch die Versionsverwaltung waren auch parallele Arbeiten am Projekt jederzeit möglich.

Teamkommunikation und Scrum

Hier lässt sich sagen, dass die Teamkommunikation als solches und die Anwendung von Scrum, mit besonderem Hinblick auf Iterationen, sehr gut funktioniert hat. Die Kommunikation war während des gesamten Projekts jederzeit gewährleistet und es kam nur zu wenig Missverständnissen.

Schnelles Vorankommen

Trotz der vielen Herausforderungen und gleichzeitig großen Ambitionen, hat das Projekt gute Fortschritte erzielt. Die angestrebten Ziele wurden immer erreicht.

Transition React.js

An dieser Stelle ist hervorzuheben, dass die Entscheidung, auf React.js als Clientlösung umzusteigen, sehr gut funktioniert hat.

Austausch mit Dozenten

Aufkommende Fragen wurden immer schnell und ausführlich beantwortet.

Lösungsfindung

Nach einer gewissen Einarbeitungszeit konnte ein Gespür für die Möglichkeiten und Funktionsweise aller Komponenten/ Docker entwickelt werden, die zu zufriedenstellenden Lösungen (Möglichkeiten React js, Lösung mit CORS, Lösung des Rejectionproblems) geführt haben. Gegen Projektende liefen immer mehr Dinge reibungslos ab.

5.2 Was hätte besser gemacht werden können

lmsserver.js

Es gibt großes Verbesserungspotential bei dem Aufbau und Design der Schnittstellen/Schnittstellenaufrufen. Hierbei wäre bei der Struktur angebracht gewesen, die Requests,

bei denen lediglich Daten als Antwort empfangen werden (ohne das der Aufruf Ressourcen beim Server ändert), mit der GET-Methode zu gestalten, da diese Aufrufe idempotent sind. Dies würde unter anderem für die folgenden Schnittstellen gelten:

„/fetch_data“ imsserver.js

„/fetch_depotinhalt“ imsserver.js

Auch wäre es besser gewesen, Datenbankoperationen in eigenständige Funktionen auszulagern. Dies hätte für mehr Übersichtlichkeit gesorgt. Anzumerken ist hier vor allem auch die zusätzliche Nutzung von asynchronen Funktionen. Das Programm wäre zudem so erweitert worden, dass eine Validierung der Anmeldedaten beim imsserver.js stattfindet.

Datadelivery.js

Die Logik hätte durch den Einsatz einer for-each Schleife, die automatisch jeden Timestampkey des JSON´s der Alphavantage-Schnittstelle durchläuft, sehr stark vereinfacht werden können.

Komponenten React.js

Die Struktur und der Aufbau von dem IMS-React.js-Client ist durch den Lernprozess geprägt und daher an einigen Stellen unstrukturiert. Dies hätte durch eine bessere Planung vermieden werden können. Eine bessere Namensgebung, Ordnerstruktur und eine Hilfsklasse für Schnittstellenaufrufe hätten zu einer übersichtlicheren Struktur geführt.

Generell

Durch ein vermehrtes Arbeiten mit asynchronen Funktionen hätte eine Schachtelung von Datenbankabfragen-/operationen ineinander vermieden werden können und der Code wäre übersichtlicher. Es wurde sehr früh implementiert, dass Abfragen auf bzw. die Erstellung der Schematas hartkodiert in den jeweiligen Node.js Dateien liegen. Eine elegantere Lösung wäre hier die Verwendung einer Db.init-Datei gewesen. Darüber hinaus wäre es sinnvoll gewesen, den imsserver.js und seine 4 essentiellen Schnittstellen in weitere kleinere Microservices aufzuspalten, um stärker von den Vorteilen einer Microservicearchitektur zu profititieren. Außerdem war ursprünglich angedacht mittels der Käufe und Verkäufe, den Einkaufswert einer Aktie pro User zu berechnen. Damit hätte die tatsächliche Wertveränderung dargestellt werden könne. Dies wäre in weiteren Iterationen realisiert worden.

Literaturverzeichnis

Alphavantage (2020a): Customer Support | Alpha Vantage, https://www.alphavantage.co/support/#api-key, Abruf: 23.04.2020

Alphavantage (2020b): API Documentation | Alpha Vantage, https://www.alphavantage.co/documentation/, Abruf: 23.04.2020

Beliaev, i. (2014): MariaDB vs MySQL - Vorteile und Nachteile im Überblick, https://blog.php-dev.info/2014/04/mariadb-vs-mysql/, Abruf: 23.04.2020

Buchanan, S./Rangama, J./Bellavance, N. (2020): INTRODUCING AZURE KUBERNETES SERVICE, A practical guide to container orchestration, [S.l.]: APRESS

Influxdata (2020): Time Series Database (TSDB) Explained | InfluxDB | InfluxData, https://www.influxdata.com/time-series-database/, Abruf: 23.04.2020

Infopulse (2020): ReactJS einfach erklärt: Warum ist es gut für die Entwicklung, https://www.infopulse.com/de/blog/reactjs-einfach-erklaert-warum-ist-es-gut-fuer-die-entwicklung/, Abruf: 23.04.2020

Joos, T. (2019): Die 10 wichtigsten Vor- und Nachteile von Docker-Container, https://www.ip-insider.de/die-10-wichtigsten-vor-und-nachteile-von-docker-containern-a-844230/, Abruf: 23.04.2020

Momot, L. (2016): Microservices- Vor und Nachteile, https://www.sdx-ag.de/2016/11/microservices-vor-und-nachteile/, Abruf: 23.04.2020

Node.js (o.J.): About | Node.js, https://nodejs.org/en/about/, Abruf: 23.04.2020

Node-Influx (2017): Typedef | node-influx API Document, https://node-influx.github.io/typedef/index.html#static-typedef-IPoint, Abruf: 23.04.2020

o. A. (2020a): Cross-Origin Resource Sharing (CORS), https://developer.mozilla.org/de/docs/Web/HTTP/CORS, Abruf: 23.04.2020

o.A. (2020b): Was ist mariadb?, https://www.webgo.de/hilfe/content/77/103/de/was-ist-mariadb.html, Abruf: 23.04.2020

React.js (2020): React – A JavaScript library for building user interfaces, https://reactjs.org/, Abruf: 23.04.2020

Richardson, C. (o.J.): What are microservices?, https://microservices.io/, Abruf: 23.04.2020

Vantage, A. (2020): Alpha Vantage - Free APIs for Realtime and Historical Stock, Forex (FX), Cryptocurrency Data, Technical Analysis, Charting, and More!, https://www.alphavantage.co/#about, Abruf: 23.04.2020

Erklärung

Wir erklären hiermit, dass die Seminararbeit mit dem Thema: Ausarbeitung zum Projekt Investment Management System (IMS) selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt haben.

Wir versichern zudem, dass die eingereichte elektronische Fassung mit der gedruckten Fassung übereinstimmt.

Ludwigsburg, 29.04.2020	Maxim Bauer
(Ort, Datum)	(Unterschrift)
Ludwigsburg, 29.04.2020	Lena Simon
(Ort, Datum)	(Unterschrift)
Ludwigsburg, 29.04.2020	Lukas Spieß
(Ort, Datum)	(Unterschrift)