



## **Projektname: The Wallstreet Wizzard – Demo Broker**

**Projektbeschreibung: Webprogrammierungsprojekt**

**Verfasser\*innen:**

Marko Robert Szönyi

Arvid Harang

Lennart Pfeiler

**Studiengang Wirtschaftsinformatik**

**Bearbeitungszeitraum: September 2024 – Oktober 2024**

**Matrikelnummern:**

2045018

3959770

6667860

**Kurs: WI23A**

**E-Mails:**

mar.szoenyi.23@lehre.mosbach.dhbw.de,

arv.harang.23@lehre.mosbach.dhbw.de,

len.pfeiler.23@lehre.mosbach.dhbw.de

**Studiengangsleiter: Prof. Dr. Christoph Sturm**

## **Zusammenfassung**

Das Portfolio über das Projekt "The Wall Street Wizzard" – ein Demo-Broker für den Aktienhandel – stellt die wichtigsten Ergebnisse dieses Projektes dar. Im ersten Kapitel wird erläutert, warum sich das Team für die Entwicklung einer Trading-Plattform entschieden hat und welche Rolle die aktuelle wirtschaftliche Lage und die steigende Bedeutung von digitalen Finanzlösungen dabei gespielt haben. Der zweite Abschnitt beschreibt das Endergebnis der App mithilfe eines Architekturdiagramms. Die Plattform besteht aus einer REST API, die mit einer Datenbank verbunden ist, sowie einer interaktiven Webanwendung für Nutzer, um in Echtzeit Aktien zu handeln. Im dritten Abschnitt werden die Herausforderungen und Erfolge während der Entwicklung beschrieben, einschließlich der technischen Umsetzung der Kernkomponenten: Backend, Frontend und die Integration von externen APIs für Echtzeit-Marktdaten. Die grundlegenden Funktionen des Users, wie die Verwaltung von Portfolios, das Platzieren von Handelsaufträgen (Buy/Sell) und die Anzeige von Marktanalysen, wurden erfolgreich umgesetzt. Zudem gibt jedes Teammitglied eine Einschätzung zu seinen Beiträgen und reflektiert die gewonnenen Erfahrungen und Lernergebnisse während des Projekts.

## **Abstract**

The portfolio on the project “The Wall Street Wizzard” - a demo broker for share trading - presents the most important results of this project. The first section explains why the team decided to develop a trading platform and what role the current economic situation and the increasing importance of digital financial solutions played in this. The second section describes the final result of the app using an architecture diagram. The platform consists of a REST API connected to a database and an interactive web application for users to trade stocks in real time. The third section describes the challenges and successes during development, including the technical implementation of the core components: Backend, Frontend and the integration of external APIs for real-time market data. The basic functions of the user, such as managing portfolios, placing trading orders (buy/sell) and displaying market analysis, were successfully implemented. In addition, each team member gives an assessment of their contributions and reflects on the experience gained and learning outcomes during the project.

# Inhaltsverzeichnis

1. Produktbeschreibung und Motivation .....	1
2. Beschreibung der Use Cases .....	2
3. API-Design.....	6
4. Herausforderungen und Erfolge während der Entwicklung .....	7
4.1. Backend .....	7
4.2. Java Script Code .....	9
4.3. Multipage App.....	11
4.4. Single Page App .....	14
4.5. GraphQL.....	17
4.6 Alexa .....	19
5. Wissenschaftliche Reflexion .....	20
5.1. Lennart Pfeiler .....	20
5.2. Marko-Robert Szönyi .....	21
5.3. Arvid Harang .....	22

## Abbildungsverzeichnis

Abbildung 1: Use-Case-Diagramm .....	2
Abbildung 2: Architekturdiagramm .....	4
Abbildung 3: Darstellung des finally Blocks am Beispiel der deletePortfolioStock-Methode .....	7
Abbildung 4: Ausschnitt aus dem DecreasePortfolioStockOrder-Endpunkt .....	8
Abbildung 5: Anzeige eines PortfolioStocks vor Laden des aktuellen Aktienpreises .....	10
Abbildung 6: Anzeige eines PortfolioStocks nach Laden des aktuellen Aktienpreises ..	10
Abbildung 7: Header und Navbar der html-Seiten .....	11
Abbildung 8: Beispielhafte Darstellung von zwei Slide-Animationen .....	12
Abbildung 9: Media Queries .....	13
Abbildung 10: Darstellung der richtigen Navbar in Angular .....	14
Abbildung 11: Nutzung des Router-outlet .....	15
Abbildung 12: Verschachtelte Routen .....	15
Abbildung 13: Alexa Input Types .....	18

# 1. Produktbeschreibung und Motivation

"The Wall Street Wizzard" ist eine Plattform, die wir ins Leben gerufen haben, um Nutzern die Möglichkeit zu geben, den Aktienmarkt in einer sicheren Umgebung zu simulieren. Mit virtuellem Geld kann man hier handeln und dabei seine Trading-Fähigkeiten verbessern, ohne echtes Kapital aufs Spiel zu setzen. Die Idee zu dieser App entstand, weil das Interesse an Finanzmärkten immer weiter wächst und wir besonders Anfängern eine unterhaltsame Möglichkeit bieten wollten, den Aktienhandel kennenzulernen.

Die wirtschaftlichen Turbulenzen der letzten Jahre - von der Covid-19-Pandemie bis hin zu globalen Finanzunsicherheiten - haben viele Menschen dazu gebracht, sich mehr mit ihren Finanzen auseinanderzusetzen. Mit unserer App möchten wir diesen Personen ein Werkzeug an die Hand geben, mit dem sie spielerisch lernen können, wie man Aktien profitabel handelt, Marktdaten analysiert und ein Portfolio führt.

Unser Ziel ist es, eine realistische Börsenumgebung zu schaffen, die neben den Mechanismen des Marktes auch die Emotionen, die mit Kursschwankungen einhergehen, vermittelt. Das ist besonders wichtig für alle, die neu in die Welt des Aktienhandels eintauchen und noch unsicher sind, wo sie anfangen sollen.

"The Wall Street Wizzard" soll nicht nur lehrreich sein, sondern auch Spaß machen. Nutzer können verschiedene Aktien beobachten, Kauf- und Verkaufsentscheidungen treffen und ihre Strategien anpassen. Durch die Verbindung mit echten Marktdaten wird das Lernen noch effektiver. Gleichzeitig können sie virtuelle Portfolios erstellen und verwalten, was ihnen im Idealfall Selbstvertrauen für zukünftige reale Investitionen gibt.

Ein weiterer Vorteil unserer App ist der soziale Aspekt. Wissbegierige Nutzer können durch die Inhalte und Funktionen unserer Anwendung voneinander lernen, Erfolge nachverfolgen und ihr Wissen erweitern. Für Einsteiger bietet dies eine hervorragende Möglichkeit, die Finanzwelt risikofrei und auf spielerische Weise zu erkunden.

Uns war es wichtig, eine App zu entwickeln, die sowohl Anfänger als auch Fortgeschrittene anspricht. Wir wollten etwas schaffen, das wir selbst gerne nutzen würden, um unsere Fähigkeiten im Aktienhandel zu testen und zu verbessern. Außerdem haben wir durch die Arbeit an diesem Projekt viel über Webentwicklung gelernt und festgestellt, dass wir uns in diesem Bereich auch beruflich weiterentwickeln möchten.

## 2. Beschreibung der Use Cases

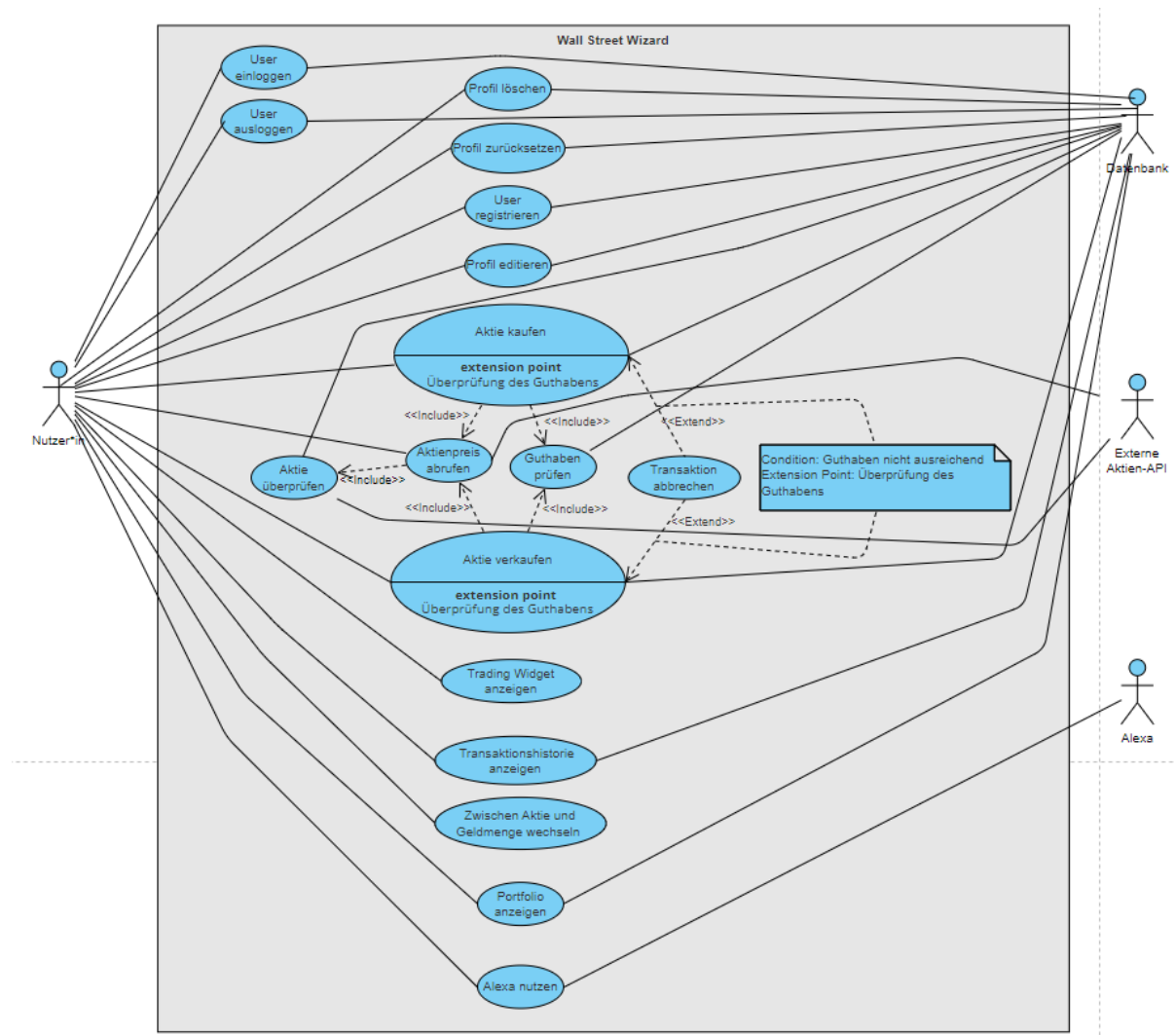


Abbildung 1: Use-Case-Diagramm

Das gezeigte Use-Case-Diagramm stellt die wichtigsten Funktionalitäten und Akteure der Anwendung „Wall Street Wizzard“ dar. Es wurde genutzt, um die Struktur und die Interaktionen der Anwendung zu definieren und zu entwickeln. Das Diagramm zeigt insgesamt 17 Use Cases, die in der Anwendung durch den Nutzer, die Datenbank, eine externe Aktien-API und Alexa genutzt werden können.

Ein zentraler Fokus der Entwicklung liegt auf den folgenden Funktionen:

- User einloggen/ausloggen: Diese grundlegenden Anwendungsfälle betreffen die Authentifizierung der Nutzer, damit sie auf die geschützten Funktionen der Anwendung zugreifen können.
- User registrieren: Dies erlaubt es einem neuen Benutzer ein Profil für die Anwendung zu erstellen.

- Profil editieren: Dies ermöglicht es einem Benutzer innerhalb der Anwendung seinen Vorname, Nachnamen und Email zu verändern. Zusätzlich besteht die Möglichkeit manuell Budget zum Portfolio hinzuzufügen.
- Aktien kaufen/verkaufen: Dies sind die wesentlichen Funktionen der Anwendung, da sie das Hauptziel der Plattform unterstützen. Der Nutzer kann Transaktionen zum Handeln von Aktien hinzufügen. Innerhalb beider Use Cases gibt es einen Extension-Point für die Überprüfung des Guthabens, was sicherstellt, dass Transaktionen nur durchgeführt werden, wenn der Kontostand des Nutzers ausreicht. Die Use Cases wurden gemäß des LIFO-Prinzips entwickelt und enthalten jeweils viele Hintergrundoperationen, um die Richtigkeit der Transaktionen zu gewährleisten.
- Guthaben prüfen und Aktienpreis abrufen: Diese Funktionen sind entscheidend für den Kauf- und Verkaufsprozess von Aktien. Sie sind als „include“-Beziehungen mit dem Aktienhandel verbunden, da diese Informationen erforderlich sind, bevor eine Transaktion durchgeführt werden kann.
- Aktie überprüfen: Dieser inkludierte Use Case „Aktienpreis abrufen“, überprüft, ob die abgefragte Aktie bereits in der Datenbank existiert. Falls nicht wird diese mit allen notwendigen Daten in die Stock-Tabelle hinzugefügt, damit ein User eine Transaktion mit der Aktie durchführen kann.
- Transaktion abbrechen: Dieser erweiterte Use Case wird aktiviert, wenn die Überprüfung des Guthabens ergibt, dass das Guthaben nicht ausreichend für die Transaktion ist. In diesem Fall wird die Transaktion abgebrochen. Dieser Use Case umfasst ebenfalls weitere Hintergrundüberprüfungen, welche das Abschließen einer Ordertransaktion verhindern.
- Trading Widget anzeigen: Dies ermöglicht es einem Benutzer ein TradingView Widget in der Oberfläche anzuzeigen. In diesem ist es ihm möglich verschiedene Informationen über den Markt zu erhalten und Indikatoren für die technische Analyse benutzerfreundlich zu erstellen.
- Transaktionshistorie anzeigen: Dies ermöglicht es dem Nutzer all seine Transaktionen der Vergangenheit anzuzeigen.
- Wechsel zwischen Aktie und Geldmenge: Dem Benutzer steht die Möglichkeit zu, eine Transaktion über Angabe eines Geldbetrags oder Angabe der Aktienanzahl durchzuführen.
- Portfolio anzeigen: Dieser Use Case zeigt dem Benutzer seine eigentlichen Portfoliodaten an. Dies umfasst die Anzeige des gesamten Portfoliowerts, die Positionshöhe einzelner Aktienpositionen sowie das verbleibende Budget. Dem Benutzer wird ebenfalls die prozentuale Änderung der Portfoliowerte im Vergleich zum Kaufzeitpunkt angezeigt, um den Erfolg der Käufe darzustellen.
- Alexa nutzen: Hier wird die Integration mit dem Sprachassistenten Alexa ermöglicht, um bestimmte Funktionen der Anwendung zu steuern.



Die Anwendung „Wall Street Wizzard“ fokussiert sich hauptsächlich auf den Aktienhandel und die damit verbundenen Datenabfragen. Die Abhängigkeiten und Bedingungslogik, wie die Guthabenprüfung, sind essenziell für die korrekte Abwicklung der Transaktionen. Weitere Use Cases wie die Integration von Alexa und die Transaktionshistorie runden das Nutzererlebnis ab, indem sie zusätzliche Informationen bereitstellen und den Zugang zur Anwendung vereinfachen.

Insgesamt konnten alle Use Cases bis auf die Alexa Integration in vollständiger Form implementiert werden. Die Umsetzung der Alexa umfasst nicht alle Anwendungsfunktionalitäten, da dies innerhalb des Zeitraums nicht umsetzbar war. Eine zukünftige Erweiterung der Alexa API um die Hauptfunktionen der Anwendung ist definitiv als sinnvoll zu betrachten, um potenziellen Kundenwünschen gerecht zu werden.

Der Aufbau der Anwendung ist in folgendem Architekturdiagramm dargestellt:

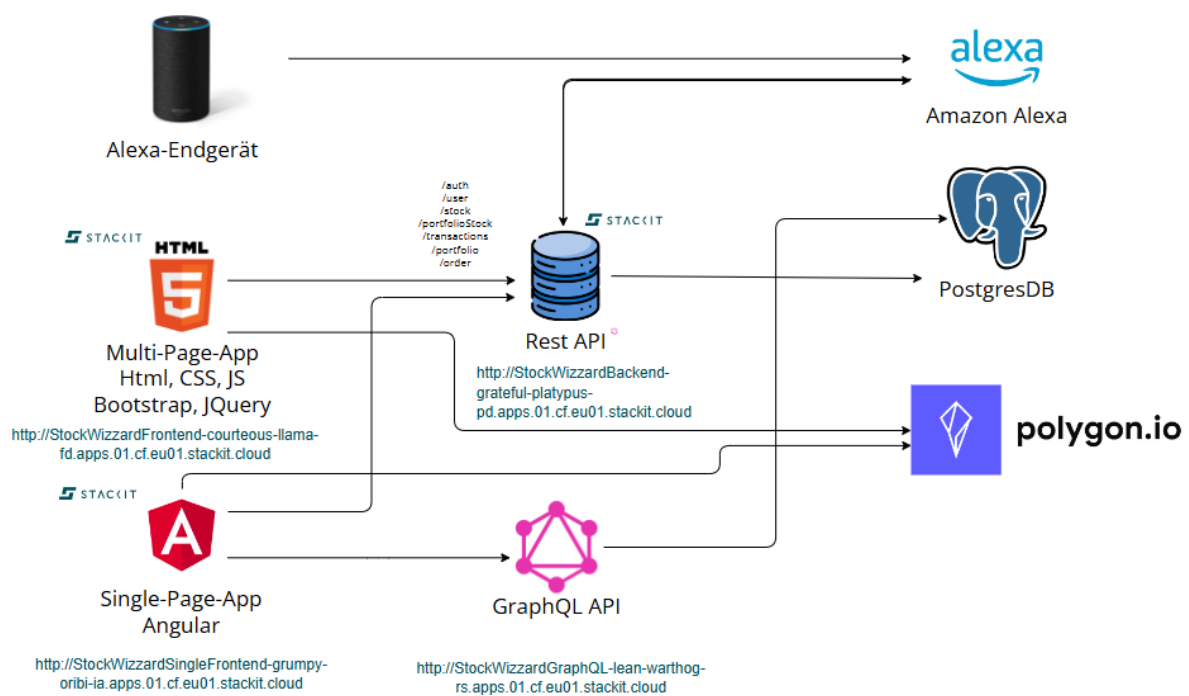


Abbildung 2: Architekturdiagramm

Abbildung 2 stellt die der The Wallstreet Wizzard App zugrundeliegende Architektur dar. Die zentrale Komponente ist eine REST API, die über die URL <http://StockWizzardBackend-grateful-platypus-pd.apps.01.cf.eu01.stackit.cloud> zugänglich ist. Diese API bietet Endpunkte wie `/auth`, `/user`, `/stock`, `/portfolioStock`, `/transactions`, `/portfolio` und `/order`, um eine Vielzahl von Funktionen im Zusammenhang mit Benutzerverwaltung, Aktienhandel und Portfolio-Management bereitzustellen.

Die REST API greift innerhalb der Endpunkte auf die PostgreSQL-Datenbank zu, die zur Speicherung aller erforderlichen Benutzer-, Transaktions- und Portfoliodaten verwendet

wird. Neben der selbst entwickelten API werden Markt- sowie Unternehmensdaten der Aktien über die externe Polygon.io API verwendet.

Für die Benutzeroberfläche stehen zwei Webanwendungen zur Verfügung, die beide mit der REST API verbunden sind. Erstere ist eine Multi-Page-App (MPA), die unter <http://StockWizzardFrontend-courteous-llama-fd.apps.01.cf.eu01.stackit.cloud> erreichbar ist. Diese Multi-Page-App bietet den Benutzern eine vollfunktionale Weboberfläche für die Interaktion mit dem Aktienportfolio und weiteren Funktionen der Plattform.

Die zweite Webanwendung ist eine Single-Page-App (SPA), entwickelt mit Angular, die unter <http://StockWizzardSingleFrontend-grumpy-oribia.apps.01.cf.eu01.stackit.cloud> zur Verfügung steht. Diese SPA ermöglicht eine schnelle und dynamische Benutzerinteraktion, da sie ohne vollständiges Aktualisieren der Seite mit der REST API kommuniziert. Diese Webanwendung verwendet zum Teil Rest-API Aufrufe, jedoch auch den entwickelten GraphQL-Endpunkt. Insgesamt betrachtet stellt die SPA zwar auch alle Funktionen zu Verfügung, jedoch ist diese im Frontend in wenigen Punkten unterschiedlich zur MPA. Hauptunterschied zu der entwickelten Multipage ist die fehlende Aktualisierung der Portfoliopage nachdem ein User diese aufgerufen hat. Aufgrund dessen wird empfohlen vorerst die MPA für das volle Nutzerlebnis zu verwenden.

Darüber hinaus ist die MPA in der Lage, über Amazon Alexa mit einem Alexa-fähigen Endgerät zu kommunizieren. Alexa ist an die REST API angebunden und kann Informationen über aktuell registrierte Benutzer der Seite abrufen und wiedergeben.

Diese Architektur kombiniert ein flexibles Backend mit mehreren Frontend-Optionen und einer Sprachsteuerungsintegration, um eine umfassende und benutzerfreundliche Plattform für den Aktienhandel bereitzustellen.

### 3. API-Design

Zu Beginn des Projekts wurde das API-First-Prinzip als Leitlinie für die Entwicklung festgelegt. Dies bedeutet, dass die API-Endpunkte vorab definiert und dokumentiert wurden, um eine klare Schnittstelle zwischen Frontend und Backend zu schaffen und eine frühe Abstimmung zu gewährleisten. Die initial festgelegten REST-Endpunkte ermöglichten eine gezielte Entwicklung und Tests auf beiden Seiten. Im Verlauf der Implementierung zeigte sich jedoch, dass zusätzliche Endpunkte erforderlich waren, um erweiterte Funktionalitäten abzudecken und den Anforderungen gerecht zu werden. Diese Erweiterungen wurden sorgfältig dokumentiert, um die Konsistenz der API-Dokumentation zu gewährleisten. Insgesamt wurden 18 REST-Endpunkt innerhalb der Multi-Page integriert. Für alle Endpunkte, die innerhalb der Anwendung nach dem Login aufgerufen werden, ist ein Authentifizierungstoken erforderlich, um die korrekte Authentifizierung und Autorisierung der Anwendung sicherzustellen.


Für die Single Page Anwendung wurde zusätzlich GraphQL eingebunden. Im Gegensatz zu der API-Struktur mit REST, wird bei GraphQL nur ein einziger Endpunkt benötigt und ermöglicht somit eine Performancesteigerung. In unserer GraphQL-Implementierung werden, bis auf eine Ausnahme, nur einzelne Argumente übergeben. Da dies oft keine komplexen Anforderungen und Verschachtelungen enthält, haben wir uns für dieses Vorgehen entschieden. Bei den Methoden mit Alexa, übergeben wir eine ganze Instanz, da diese verschachtelt ist. Somit ersparen wir uns einige Hilfsklassen, welche wir sonst für den Input der Mutations benötigt hätten. Stattdessen definieren wir die Werte für die einzelnen Instanzen mit den setter-Methoden.

Eine ausführliche Dokumentation der REST-Endpunkte und der GraphQL Queries und Mutations befindet sich unter den Anlagen, da der Umfang dieser Dokumentationen den Rahmen dieses Portfolios stark übersteigen würden.

## 4. Herausforderungen und Erfolge während der Entwicklung

### 4.1. Backend

Nachdem die ersten API-Endpunkte entworfen worden waren, konnte die Backend-Entwicklung starten. Die erstellten UML-Diagramme sowie ein im Modul "Fallstudie" entwickeltes ER-Modell bildeten dabei die Basis, wodurch die Implementierung der notwendigen Klassen und Interfaces effizient begonnen werden konnte. Im Laufe der Zeit konnten somit die definierten API Endpunkte implementiert werden, was das Hinzufügen weiterer Klassen zur Verwendung als RequestBody oder Rückgaben umfasste. Zu Beginn der Entwicklung wurden Property-Dateien als einfaches Mittel zur persistenten Datenspeicherung eingesetzt. Diese Dateien boten eine schnelle und unkomplizierte Möglichkeit, Benutzerdaten vorübergehend zu speichern, um Anwendungsfunktionalitäten zu testen. Mit der zunehmenden Komplexität der Anwendung und dem Vorteil, unkompliziert mit SQL auf gespeicherte Daten abzurufen, wurde jedoch auf die relationale PostgreSQL Datenbank umgestellt. Diese Umstellung erforderte zwar besondere Sorgfalt bei der Handhabung der Verbindungen, stellte sich jedoch sehr schnell als Zeitersparnis heraus, da durch die Verwendung von Subqueries in den Abfragen um einiges schneller Daten verschiedener Tabelle abgerufen werden konnten, als mit dem Iterieren durch verschiedene Properties Dateien. Da bei den Datenbankverbindungen oftmals Verbindungsfehler aufgetreten sind, spielte der darauf verwendete Einsatz von finally-Blöcken eine wichtige Rolle, um sicherzustellen, dass Datenbankverbindungen unabhängig vom Ausgang einer Operation immer korrekt geschlossen werden.



```
    } finally {
        try {
            if (rs != null) rs.close();
            if (stmt != null) stmt.close();
            if (connection != null) connection.close();
        } catch (SQLException e) {
            Logger.getLogger(name:"DeletePortfolioStockLogger").log(Level.SEVERE, msg:"Error when closing the resource. Error: {0}", e);
            e.printStackTrace();
        }
    }
```

Abbildung 3: Darstellung des finally Blocks am Beispiel der deletePortfolioStock-Methode

Eine weitere Herausforderung bestand darin, die verschiedenen Funktionalitäten der Anwendung in sinnvolle API-Aufrufe zu unterteilen, um unnötige API-Calls zu vermeiden. Dies war besonders wichtig, um die Performance zu optimieren und die Last auf den Server zu reduzieren. Es wurde sorgfältig abgewogen, welche Operationen zusammengefasst werden konnten und welche eine separate Verarbeitung erforderten.

Dies stellte sich vor allem bei der Aufstellung der Order-Funktion als Herausforderung dar, da das Erstellen einer Order viele Funktionalitäten erfüllt, welche als verschiedene http-Methoden angesehen werden können. Da das Erstellen einer Order verschiedene Unterfunktionalitäten erfüllt, darunter Transaktion erstellen, User Budget verändern und

Aktienpositionswert anpassen, fiel es uns schwer einen geeigneten Endpunkt auszuwählen. Während zu Beginn jeweils ein POST-Endpunkt für den Aktienkauf bzw. -verkauf vorgesehen war, ergab sich schnell, dass dies kein guter Weg zur Implementation gewesen wäre, da die jeweiligen Methoden zwei Funktionalitäten hätten erfüllen müssen, was dem Single-Responsibility Prinzip widersprochen hätte. Somit wurde der Aktienkauf in die zwei Endpunkte `createBuyOrder(POST)` sowie `increasePortfolioStockOrder(PUT)` und der Aktienverkauf in die Endpunkte `decreasePortfolioStockOrder(PUT)` und `deletePortfolioStockOrder(DELETE)` aufgeteilt.

Die Implementation dieser Funktion war im Nachhinein betrachtet am zeitintensivsten und schwersten. Dies lag zum einen daran, dass wir versucht haben, Transaktionen wie bei einem realen Broker durchzuführen, sodass die Nutzer der Anwendung eine hohe User Experience haben. Dazu zählt beispielsweise die Implementation des LIFO-Prinzips beim Verkauf einer Aktie, bei welcher die zuletzt gekauften Aktien zuerst verkauft werden. Dies half dabei, die Abwicklung von Transaktionen konsistent zu gestalten und den realen steuerlichen Anforderungen gerecht zu werden, die auf die Reihenfolge der Verkäufe abzielen.

```
transactionManager.addTransaction(transactionContent);
Portfolio userPortfolio = portfolioManager.getUserPortfolio(transactionContent.getEmail());
List<Transaction> transactionsInPortfolio = transactionManager.getAllTransactionsInPortfolioStock(transactionContent.getEmail());
Double remainingAmount = transactionContent.getTotalPrice();
Double totalBoughtValueReduction = 0.0;
for (Transaction transaction : transactionsInPortfolio) {
    if (remainingAmount <= 0) {
        break;
    }
    Double leftInTransaction = transaction.getLeftInPortfolio();
    Double transactionBoughtValue = transaction.getTotalPrice();
    Integer transactionId = transaction.getTransactionID();
    if (remainingAmount >= leftInTransaction) {
        remainingAmount -= leftInTransaction;
        totalBoughtValueReduction += transactionBoughtValue;
        transactionManager.editLeftInPortfolio(transactionId, newLeftInPortfolio:0.0);
    } else {
        Double proportion = remainingAmount / transactionBoughtValue;
        Double reductionInBoughtValue = transactionBoughtValue * proportion;
        totalBoughtValueReduction += reductionInBoughtValue;

        Double newLeftInTransaction = leftInTransaction - remainingAmount;
        remainingAmount = 0.0;
        transactionManager.editLeftInPortfolio(transactionId, newLeftInTransaction);
    }
}
Double newCurrentValue = portfolioStockValues.getCurrentValue() - transactionContent.getTotalPrice();
Double newBoughtValue = portfolioStockValues.getBoughtValue() - totalBoughtValueReduction;
portfolioStockManager.decreasePortfolioStock(newCurrentValue, newBoughtValue,
    transactionContent.getStockAmount(), userPortfolio.getPortfolioID(), transactionContent.getSymbol());
userManager.editUserBudget(currentUser.getEmail(), currentUser.getBudget(),
    transactionContent.getTotalPrice(), transactionContent.getTransactionType());
editPortfolioValue(currentUser.getEmail());
return ResponseEntity.ok(new StringAnswer(answer:"Transaction was successfully completed!"));
```

Abbildung 4: Ausschnitt aus dem `DecreasePortfolioStockOrder`-Endpunkt

Ein weiterer kritischer Aspekt war die präzise Handhabung von Preisdaten. Da die abgerufenen Aktienschlusskurse der externen API oft mehr als zwei Nachkommastellen umfassen, wurde konsequent eine Rundung auf zwei Dezimalstellen durchgeführt.

Diese Maßnahme stellt sicher, dass alle Berechnungen korrekt sind und die finanzielle Integrität der Daten gewahrt bleibt. Die genaue Behandlung der Nachkommastellen war besonders bei der Berechnung von Transaktionswerten sowie des Portfoliowerts und Budgets entscheidend, um Ungenauigkeiten und potenzielle Fehler bei der Darstellung der Ergebnisse zu vermeiden.

Neben den bereits genannten Schwierigkeiten gab es viele weitere Probleme bzw. Herausforderungen, welche jedoch nicht weiter im Detail erläutert, da diese entweder als Teilbereich der genannten Herausforderungen angesehen werden können oder keinen großen Umfang hatten.

## **4.2. Java Script Code**

Die größte Herausforderung bei der Implementierung der Anwendungsfunktionalität stellte die Integration einer externen API dar. Diese API wird verwendet, um tägliche Schlusskursdaten von Aktien sowie Firmeninformationen der Aktienunternehmen zu erhalten. Zu Beginn wurde eine API verwendet, die sich jedoch aufgrund der begrenzten Anzahl monatlicher Aufrufe in der kostenlosen Version als ungeeignet für die Anwendung erwies. Daher wurde die Entscheidung getroffen, die Polygon.io API zu nutzen, die bis zu fünf Aufrufe pro Minute ermöglicht. Obwohl dies eine signifikante Verbesserung gegenüber der vorherigen API darstellt, führte es dennoch zu erheblichen Komplikationen bei der Entwicklung der Portfolioseite.

Auf der Portfolioseite werden alle Aktien des Portfolios angezeigt, was es notwendig macht, für jede Aktie den aktuellen Preis abzurufen. Zusätzlich kann der Benutzer den Aktienpreis manuell durch Eingabe eines Aktiennamens abrufen. Da diese Anforderungen die Grenze von fünf Anfragen pro Minute schnell überschreiten, musste eine Lösung entwickelt werden, um diese Anfragen zu begrenzen und gleichzeitig die Benutzererfahrung nicht zu beeinträchtigen.

Der optimale Implementierungsweg, der zunächst verfolgt wurde, bestand darin, die aktuellen Aktienpreisdaten in der Datenbank zu speichern, sodass eine Anfrage des Preises nicht direkt an den API-Endpunkt gesendet werden muss. Dies erfordert jedoch ein serverseitig ausgeführtes Timer-Event, dass die Aktiendaten täglich aktualisiert. Da die Aktienpreise in der API nur einmal täglich aktualisiert werden, wäre das einmalige Ausführen des Timer-Events ausreichend. Mithilfe dieser Methode wäre ein Aktienpreisabruf nur notwendig, wenn eine Aktie noch nicht in der Datenbank vorhanden ist und hinzugefügt werden muss.

Nach Absprache wurde jedoch entschieden, diese Methode nicht zu verfolgen, da kurze Wartezeiten die Benutzererfahrung nicht großartig negativ beeinflussen. Aufgrund dessen musste der gesamte Code, der mit den externen Aktiendaten in Verbindung

steht, geändert werden, da die Abrufbegrenzung schnell zu langen Wartezeiten führen kann.

Die Anzeige der Aktien im Portfolio wurde in zwei Methoden aufgesplittet. Zunächst wird jede Aktie ohne die Höhe der Position und Prozentänderung angezeigt. Während der aktuelle Aktienpreis abgerufen wird, zeigt die Anwendung “Calculating value” an, was den Benutzer darüber informiert, dass das Laden der Aktie noch nicht abgeschlossen ist.

```
//Display a single portfolioStock
function displayPortfolioStock(symbol) {
  const stockListContainer = document.querySelector('.portfolio .stock-list');
  const stockDiv = document.createElement('div');
  stockDiv.id = symbol; // Füge eine ID für die spätere Aktualisierung hinzu
  stockDiv.innerHTML = `${symbol}: Calculating Portfolio data... <span class="change"></span>`;
  stockListContainer.appendChild(stockDiv);
}
```

Abbildung 5: Anzeige eines PortfolioStocks vor Laden des aktuellen Aktienpreises

Nach erfolgreichem Abruf werden die entsprechenden HTML-Elemente mit den richtigen Daten aktualisiert.

```
function updateStockDisplay(symbol, currentValue, boughtValue) {
  const stockElement = document.getElementById(symbol);
  const stockValue = roundToTwoDecimalPlaces(currentValue);

  const { percentageChange, changeClass } = calculatePercentage(boughtValue, currentValue);
  stockElement.innerHTML = `${symbol}: ${stockValue}$ <span class="change ${changeClass}">${percentageChange}</span>`;
}
```

Abbildung 6: Anzeige eines PortfolioStocks nach Laden des aktuellen Aktienpreises

Eine weitere Anpassung betraf das Aktualisieren der Portfolio-Aktien nach dem Kauf oder Verkauf einer Aktie. In der alten Implementierung wurden alle Portfolio-Aktien erneut aus der Datenbank abgerufen, was aufgrund der Abrufbegrenzung nicht praktikabel war. Daher wurde die Implementierung dahingehend geändert, dass nur das HTML-Element der betroffenen Aktie verändert, hinzugefügt oder gelöscht wird. Dies umfasst ebenfalls die Anzeige “Calculating stock value” und nach erfolgreichem Abruf die Aktualisierung mit den richtigen Daten.

Durch diese Anpassungen kann die Anwendung trotz der Einschränkungen der API eine zufriedenstellende Benutzererfahrung bieten, indem die Anzahl der API-Anfragen minimiert und die Ladezeiten optimiert werden.

Zusammenfassend lässt sich feststellen, dass der Lösungsansatz zur Bewältigung der begrenzten API-Anfragen als erfolgreich angesehen werden kann, da die Benutzererfahrung trotz der Wartezeiten nicht wesentlich beeinträchtigt wird. Dennoch hätte die Implementierung eines Timer-Events wahrscheinlich viele Probleme, die

während des Entwicklungsprozesses auftreten, vermeiden können und wäre möglicherweise der einfachere Lösungsweg gewesen. Für eine potenzielle Veröffentlichung der Anwendung ist es jedoch unerlässlich, die Aktien Daten über das Timer-Event regelmäßig zu aktualisieren und sie über die Datenbank abzurufen. Andernfalls könnte die Anzahl der API-Calls mit steigender Benutzerzahl die zulässige Grenze von fünf Anfragen pro Minute erheblich überschreiten. Aufgrund dessen wurde der bereits implementierte Weg mit dem Abruf der Daten aus der Datenbank nur auskommentiert, um diesen zukünftig direkt verwenden zu können. Außerdem bleibt die stockprice-Spalte der Stock-Tabelle bestehen, um zukünftig die Datenstruktur nicht anpassen zu müssen. Da die Spalte keine null-Values akzeptiert, wird beim Hinzufügen neuer Aktien in die Datenbank vorübergehend der Wert 0 eingefügt.

### 4.3. Multipage App

Das Frontend der Anwendung wurde parallel zum Backend entwickelt, was sich als effiziente Methode herausstellte, um eine reibungslose Abstimmung zwischen den beiden Bereichen sicherzustellen. Unser erster Schritt bestand darin, die Grundstruktur der Webseite mit HTML zu erstellen. Diese Struktur umfasste den Header, die Navigation sowie die verschiedenen Sektionen. Ein Beispiel für den HTML-Code für den Header und die Navigation aus der home.html-Datei sieht wie folgt aus:

```
<header id="header" class="d-flex justify-content-between align-items-center fixed-top p-3">
  <a href="#" class="logo">
    
  </a>

  <nav class="navbar d-flex">
    <a href="home.html" class="nav-link active" data-index="1">Home</a>
    <a href="tracker.html" class="nav-link" data-index="2">Stock tracker</a>
    <a href="portfolio.html" class="nav-link" data-index="3">Portfolio</a>
    <a href="profile.html" class="nav-link" data-index="4">Profile</a>
  </nav>

  <div class="login">
    <a href="vorHome.html" class="login-text" onclick="logout()">Logout</a>
    <a href="vorHome.html" onclick="logout()"><i class='bx bx-log-out-circle'></i></a>
  </div>
</header>
```

Abbildung 7: Header und Navbar der html-Seiten

Dieser Abschnitt bildet das Gerüst der Seite und stellt sicher, dass der Header und die Navigation auf allen Seiten vorhanden sind. Das Header-Element ist dabei fest positioniert, damit es beim Scrollen sichtbar bleibt, und enthält Links zu den wichtigsten Seiten.



Die visuelle Gestaltung der Webseite spielte eine zentrale Rolle im Entwicklungsprozess, um die Seite ansprechend und dynamisch zu gestalten. Dazu haben wir verschiedene CSS-Animationen verwendet, die das Benutzererlebnis verbessern, indem sie den Inhalten eine subtile Bewegung verleihen. Eine besondere Herausforderung war es, die Animationen so zu optimieren, dass sie auf allen Geräten möglichst flüssig laufen und keine Performanceprobleme verursachen.

Ein Beispiel hierfür ist die Animation, die wir auf der Startseite eingesetzt haben. Der folgende Code zeigt zwei CSS-Keyframes-Animationen namens `slideRight` und `slideLeft`, die verwendet werden, um Elemente sanft von der Seite hineinzuschieben. Diese Animationen sorgen dafür, dass die Elemente auf der Seite dynamisch erscheinen, statt statisch geladen zu werden:

```
@keyframes slideRight {
  0% {
    transform: translateX(-100px);
    opacity: 0;
  }
  100% {
    transform: translateX(0);
    opacity: 1;
  }
}

@keyframes slideLeft {
  0% {
    transform: translateX(100px);
    opacity: 0;
  }
  100% {
    transform: translateX(0px);
    opacity: 1;
  }
}
```

Abbildung 8: Beispielhafte Darstellung von zwei Slide-Animationen

Eine der größten Herausforderungen während der Entwicklung war es, die Responsivität der Website auf verschiedenen Geräten, insbesondere auf Mobilgeräten, sicherzustellen. Dies erforderte eine sorgfältige Planung, um sicherzustellen, dass sich die Elemente nahtlos an unterschiedliche Bildschirmgrößen und Ausrichtungen anpassen, ohne die Benutzerfreundlichkeit oder das Design zu beeinträchtigen. Da moderne Webanwendungen auf einer Vielzahl von Geräten genutzt werden können, war es entscheidend, eine Lösung zu implementieren, die das Layout dynamisch anpasst.

Um dieses Problem zu lösen, haben wir eine Kombination aus CSS-Media-Queries und dem responsiven Grid-System von Bootstrap verwendet. Bootstrap erwies sich dabei als äußerst hilfreich, da es eine flexible und gut strukturierte Grundlage für die Erstellung responsiver Layouts bot, ohne dass umfangreicher CSS-Code geschrieben werden musste. Das Framework erleichterte die Anordnung von Elementen und die Anpassung von Abständen, Polsterungen und Ausrichtungen für verschiedene Bildschirmgrößen, insbesondere für komplexe Bereiche wie das Portfolio und den Stock-Tracker.

Ein spezielles Problem trat beim Positionieren bestimmter grafischer Elemente auf, wie zum Beispiel dem `.rhombus2`-Element auf der Startseite, das auf kleineren Bildschirmen visuell überladen wirkte. Durch den Einsatz von Media-Queries konnten wir diese Elemente auf mobilen Geräten ausblenden oder deren Größe anpassen, um sicherzustellen, dass das Layout nicht überladen wirkt. Zusätzlich ermöglichten die Utility-Klassen von Bootstrap ein einfaches Umschalten der Sichtbarkeit und Layout-Anpassungen, ohne das Grunddesign zu stören.

Durch diese Herangehensweise konnten wir ein responsives Layout erreichen, das die Funktionalität und die visuelle Attraktivität der Anwendung beibehält und eine optimale Benutzererfahrung auf allen Geräten gewährleistet.

```
@media (max-width: 1420px) {  
  .rhombus2 {  
    display: none;  
    overflow: hidden;  
  }  
}  
  
@media (max-width: 1420px) {  
  .rhombus {  
    display: none;  
    overflow: hidden;  
  }  
}  
  
@media (max-width: 1420px) {  
  body {  
    overflow: scroll;  
    margin-top: 100px;  
  }  
}
```

Abbildung 9: Media Queries

Mit dieser Medienabfrage wird das Element `.rhombus2` auf kleineren Bildschirmen ausgeblendet, um das Layout nicht zu überladen und sicherzustellen, dass es gut aussieht, ohne unübersichtlich zu wirken. Dies ist ein Beispiel dafür, wie wir auf die Herausforderung der unterschiedlichen Bildschirmgrößen eingegangen sind.

Durch die Kombination aus HTML, CSS und JavaScript gelang es uns, eine funktionale und ästhetisch ansprechende Webseite zu erstellen. Die Herausforderungen lagen insbesondere in der Responsivität und der Integration dynamischer Funktionen wie der Echtzeitdaten im Stock Tracker. Dank der Verwendung von CSS-Animationen und -Medienabfragen konnten wir sicherstellen, dass die Benutzererfahrung auf verschiedenen Geräten und Bildschirmgrößen stets optimal ist.

## 4.4. Single Page App

Während der Entwicklung des Frontends mit Angular standen mehrere Herausforderungen im Mittelpunkt, insbesondere die dynamische Navigation, die Integration von TypeScript und das Styling der Anwendung. Eine der größten Hürden war die richtige Handhabung der Navigation, um sicherzustellen, dass die Benutzer abhängig von ihrem Zustand (eingeloggt oder nicht) die richtige Ansicht und Navigationsleiste sehen. Wir haben dafür `*ngIf` verwendet, um basierend auf der aktuellen Seite unterschiedliche Navigationsleisten anzeigen zu können. Der folgende Codeausschnitt zeigt, wie diese Navigation in Angular gelöst wurde:

```
<!-- Standard-Navigation vor dem Login -->
<nav *ngIf="!isVorHomePage()">
  <a routerLink="/content/vor-home">Vor Home</a>
  <a routerLink="/content/home">Home</a>
  <a routerLink="/content/portfolio">Portfolio</a>
  <a routerLink="/content/profile">Profile</a>
  <a routerLink="/content/tracker">Tracker</a>
</nav>

<!-- Navigation für Vor Home Seite -->
<ng-container *ngIf="isVorHomePage(); else defaultNavbar">
  <app-vorhome-navbar></app-vorhome-navbar> <!-- Zeigt diese Navbar nur auf Vor Home Seite -->
</ng-container>

<!-- Lila Navbar für andere Seiten, NICHT auf der Vor Home Seite -->
<ng-container *ngIf="!isVorHomePage()">
  <app-navbar></app-navbar> <!-- Zeigt diese Navbar nur für andere Seiten -->
</ng-container>

<!-- Standard Navbar für alle anderen Seiten -->
<ng-template #defaultNavbar>
  <app-navbar></app-navbar>
</ng-template>
```

Abbildung 10: Darstellung der richtigen Navbar in Angular

Dieser Code zeigt, wie mithilfe von Angular-Directives und dem `ng-container` je nach Zustand unterschiedliche Navigationsleisten geladen werden. Das war notwendig, um eine Benutzerführung zu bieten, die sich an den jeweiligen Status des Nutzers anpasst. Eine besondere Herausforderung war hier die Logik, die sicherstellt, dass die Benutzer korrekt weitergeleitet werden, wenn sie sich auf unterschiedlichen Seiten befinden. Die Funktion `isVorHomePage()` bestimmt, ob sich der Benutzer auf der „Vor Home“-Seite befindet, und zeigt dann die entsprechende Navigationsleiste an.

Ein weiteres bedeutendes Problem in der Angular-Entwicklung war die Verwaltung von Routen und die Anzeige von Komponenten basierend auf der URL. Durch das `Router-outlet`-Tag in Angular wird die Komponente geladen, die der aktuellen Route entspricht. Hier war es wichtig, dass die Routen korrekt in der `app.routes.ts`-Datei definiert sind, damit die verschiedenen Seiten wie das Portfolio, der Tracker und das Profil dynamisch

und ohne Fehler angezeigt werden. Der folgende Codeausschnitt zeigt die Verwendung von Angulars Router-outlet, das als Platzhalter für die jeweilige Komponente dient:

```
<router-outlet></router-outlet>
```

Abbildung 11: Nutzung des Router-outlet

Das Router-outlet ermöglicht es, basierend auf der aktuellen Route die richtige Komponente zu laden. Die Herausforderung bestand darin, sicherzustellen, dass die Benutzer nahtlos zwischen den verschiedenen Seiten der Anwendung navigieren können und dass die entsprechende Komponente korrekt geladen und dargestellt wird. Dies erforderte eine präzise Routenverwaltung in der TypeScript-Datei, um sicherzustellen, dass jede URL der richtigen Komponente zugeordnet ist.

Ein wesentlicher Teil der Herausforderungen bei der Entwicklung mit Angular war das Routing-System, das in der Datei `app.routes.ts` definiert wurde. Dieses System ermöglichte es uns, zwischen verschiedenen Seiten zu navigieren, ohne die Seite neu laden zu müssen, was zu einer nahtlosen Benutzererfahrung führte.

In diesem Routing-Setup haben wir mehrere Routen definiert, um sicherzustellen, dass die verschiedenen Komponenten basierend auf der URL geladen werden. Die Komplexität dieser Implementierung lag in der Verschachtelung von Routen und der Notwendigkeit, verschiedene Module wie Authentifizierung und Content-Management zu trennen.

Eine der Herausforderungen bestand darin, das korrekte Laden und Weiterleiten von Benutzern zu gewährleisten. So wird beispielsweise ein Benutzer, der die URL `/` aufruft, automatisch auf die Seite `vor-home` umgeleitet, da die erste Route definiert ist. Dies stellt sicher, dass die Anwendung von Anfang an die gewünschte Landingpage anzeigt.

Eine weitere Herausforderung war die Konfiguration der verschachtelten Routen.

```
{
  path: 'auth', component: AuthComponent,
  children: [
    { path: 'login', component: LoginComponent },
    { path: 'sign-up', component: SignupComponent }
  ]
},
```

Abbildung 12: Verschachtelte Routen

Der in Abbildung 12 abgebildete Code definiert die Routen für die Authentifizierung. Wenn ein Benutzer auf die URL `/auth/login` zugreift, wird die Login-Komponente innerhalb des `Auth-Layouts` geladen. Ähnlich wird bei `/auth/sign-up` die Registrierungsseite angezeigt. Diese Verschachtelung sorgt dafür, dass wir eine zentrale Struktur für authentifizierungsbezogene Seiten haben, während andere Routen weiterhin den Hauptinhalt anzeigen.

Eine der anspruchsvollsten Aufgaben bestand in der Verwaltung der Hauptrouten unter `/content`, die den größten Teil der Benutzeroberfläche abdecken. Hier haben wir eine Reihe von Routen definiert, wie `home`, `portfolio`, `profile`, `tracker`, und `vor-home`, die jeweils ihre eigenen Komponenten laden. Das verschachtelte Routing ermöglichte es uns, eine logische Trennung zwischen verschiedenen Bereichen der Anwendung zu schaffen, ohne unnötige Wiederholung der Logik.

Insgesamt spiegeln diese Codebeispiele einige der wichtigsten Herausforderungen wider, denen wir bei der Verwendung von Angular begegnet sind. Die dynamische Navigation, die Routenverwaltung und die Kontrolle von Benutzereingaben und Interaktionen erforderten eine sorgfältige Planung und Implementierung, um eine benutzerfreundliche und funktionale Anwendung zu gewährleisten.

Neben den Routen war die Einbindung von GraphQL ebenfalls eine große Herausforderung. Es war schwierig zu verstehen, wie alles korrekt konfiguriert und abgerufen werden musste. Die Komplexität der GraphQL-Architektur erforderte ein gründliches Einarbeiten in die verschiedenen Konzepte, wie Queries, Mutations und das Schema-Design. Diese Lernkurve machte es anfänglich herausfordernd, die richtigen Daten effizient abzurufen und sicherzustellen, dass alle Komponenten reibungslos miteinander kommunizieren. Da diese Aufgabe insgesamt sehr viel Zeit beanspruchte, wurde sich aus Zeitgründen dafür entschieden, nur einen Teil der notwendigen API-Aufrufe bzw. Query/Mutation Aufrufe in die SPA einzubinden. Die nichteingebunden Aufrufe, funktionieren jedoch trotzdem in vollem Umfang.

## 4.5. GraphQL

GraphQL ist eine leistungsstarke Abfragesprache und Laufzeitumgebung für APIs, die es ermöglicht, präzise und maßgeschneiderte Datenabfragen zu erstellen. Sie bietet eine effiziente und flexible Alternative zu traditionellen REST-APIs, bei der nur ein Endpunkt existiert, über welchen die geforderten Daten abgerufen werden können. Dies erlaubt es, verschiedene Datenquellen und Objekte in einer einzigen, flexiblen Anfrage zu kombinieren.

Allerdings bringt die Integration von GraphQL Herausforderungen mit sich, insbesondere wenn es in bestehende REST-Architekturen eingebunden werden soll. Da dieses Thema in den Vorlesungen nur oberflächlich behandelt wurde, waren wir uns bewusst, dass wir uns in viele Aspekte eigenständig einarbeiten mussten. Während der Umsetzung stießen wir auf Schwierigkeiten im Hinblick auf die Komplexität und das Zusammenspiel mehrerer einzelner Dateien und Module. Die Integration erforderte es, den GraphQL Code so zu implementieren, dass es die gleichen Ergebnisse wie die der REST-API zurückliefert. Diese Herausforderung verlangte ein tiefes Verständnis sowohl der bestehenden REST-Struktur als auch der neuen GraphQL-Konzepte.

Ein wichtiger Punkt beim Einsatz von GraphQL ist die Erstellung der Schema-Datei, die als „Vertrag“ zwischen Client und Server fungiert. Diese Schema-Datei definiert, welche Datenfelder verfügbar sind und wie sie abgerufen werden können, sowie mögliche Mutationen, um Daten zu ändern. Um GraphQL effizient zu nutzen, musste diese Datei sorgfältig strukturiert werden: Sie sollte vollständig genug sein, um alle notwendigen Abfragen und Mutationen zu ermöglichen, aber gleichzeitig nicht zu überladen sein, um unnötige Komplexität und Inflexibilität zu vermeiden. Die sorgfältige Gestaltung des Schemas erforderte Überlegungen zur Datenstruktur und den Bedürfnissen der Endnutzer, da ein schlecht konzipiertes Schema den gesamten Vorteil der Flexibilität von GraphQL zunichtemachen kann.

Die Erstellung des Schemas brachte auch spezifische Herausforderungen mit sich. Eine der größten Schwierigkeiten bestand darin, die richtige Balance zwischen Flexibilität und Kontrolle zu finden. Ein zu strikt definiertes Schema kann dazu führen, dass Entwickler nicht in der Lage sind, spezifische Abfragen durchzuführen, die sie benötigen. Auf der anderen Seite kann ein zu lockeres Schema zu Sicherheitsrisiken und einer schwer zu wartenden Codebasis führen. Diese Herausforderung ließ sich durch die bereits erstellte REST-API jedoch leicht bewältigen, da hier bereits auf die genannten Aspekte geachtet wurde. Zusätzlich stellte die Skalierbarkeit eine Herausforderung dar, da das Schema im Laufe der Zeit erweitert werden musste, um neuen Anforderungen gerecht zu werden, ohne dass bestehende Funktionalitäten gestört werden. Dies erforderte eine vorausschauende Planung und eine klare Dokumentation der Beziehungen zwischen den verschiedenen Datenfeldern und -typen. Unsere Dokumentation, half uns eine strukturierte und organisierte Weiterentwicklung zu gewährleisten.

Ein weiterer Aspekt war die Versionierung des Schemas, die im Gegensatz zu REST-APIs weniger direkt ist. Während in REST unterschiedliche API-Versionen parallel betrieben werden könnten, wird bei GraphQL das Schema kontinuierlich weiterentwickelt. Die Änderungen am Schema mussten dementsprechend abwärtskompatibel entwickelt werden, damit ältere Clients weiterhin funktionieren konnten. Dies benötigte eine durchdachte Vorgehensweise bei der Einführung neuer Features und das Entfernen veralteter Felder, um eine langfristige Stabilität der Anwendung sicherzustellen.

Darüber hinaus spielt bei der Nutzung von GraphQL die Frage der Sicherheit eine wichtige Rolle. Da die Abfragen sehr flexibel und dynamisch gestaltet werden können, besteht die Gefahr, dass schlecht konzipierte Abfragen zu einer Überlastung der Server führen. Beispielsweise kann eine tief verschachtelte Abfrage oder eine Abfrage, die eine große Menge an verschachtelten Daten anfordert, die Leistung des Servers stark beeinträchtigen. Dies machte es erforderlich, Mechanismen zur Abfrage-Validierung und -Drosselung zu implementieren, um sicherzustellen, dass das System stabil bleibt und keine unnötigen Belastungen durch unvorsichtige oder absichtlich schädliche Abfragen entstehen. In Alexa beispielsweise haben wir zusätzliche inputs erstellt, um sicher zu stellen, dass diese Risiken minimiert wurden. Des Weiteren haben wir auf die Balance zwischen Verschachtelung und unnötiger inputs geachtet, um sicher zu stellen, dass die Performance weiterhin hoch bleibt.

```
# Input Types

input AlexaROInput {
  response: ResponseInput
  session: SessionROInput
}

input ResponseInput {
  outputSpeech: OutputSpeechInput
  shouldEndSession: Boolean
}

input OutputSpeechInput {
  type: String
  text: String
}

input SessionROInput {
  attributes: [SessionAttributeInput]
}

input SessionAttributeInput {
  key: String
  value: String
}
```

Abbildung 13: Alexa Input Types

Neben der technischen Umsetzung stellt die Einführung von GraphQL in Teams, die mit klassischen REST-APIs vertraut sind, ebenfalls eine Herausforderung dar. Entwickler müssen umdenken und sich an neue Konzepte gewöhnen, was eine Lernkurve mit sich

bringt. Eine umfassende Schulung und klare Dokumentation helfen dabei, den Übergang zu erleichtern und sicherzustellen, dass alle Teammitglieder effizient mit der neuen Technologie arbeiten können. Da wir unsere Aufgaben aufgeteilt haben, haben wir in unseren Daily Scrum-Meeting sichergestellt, dass jeder einen Einblick in die Aufgaben der anderen erhält. Dies war wichtig, um sicherzustellen, dass alle Teammitglieder das gleiche Verständnis der Umsetzungen hatten. Durch dieses Vorgehen konnten wir GraphQL erfolgreich implementieren. Ein Teil der GraphQL-Queries und -Mutationen wird zudem erfolgreich in der Single Page Application (SPA) über die Konfiguration von Apollo Angular genutzt.

## 4.6 Alexa

Innerhalb der Anwendung wurde ebenfalls ein Alexa-Skill entwickelt, der die Anzahl der angemeldeten Benutzer ausgeben kann. Es war eine spannende Aufgabe, Alexa erfolgreich mit unserem Backend zu verbinden und den Skill zum Laufen zu bringen. Eine der größten Herausforderungen bestand in der Implementierung der Verbindung über HTTPS. Wir mussten dafür sorgen, dass unsere Backend URL korrekt eingerichtet ist, damit die Kommunikation zwischen Alexa und unserem Backend sicher und reibungslos funktioniert.

Im nächsten Schritt haben wir im JSON-Editor der Alexa Developer Console die nötige Struktur für den Skill erstellt. Dabei ging es darum, die richtigen Intents, Slots und Beispieläußerungen zu definieren, damit Alexa die Anfragen der Nutzer korrekt versteht. Im Backend haben wir daraufhin Alexa-Klassen genutzt, um die Anfragen zu verarbeiten und passende Antworten zu generieren. Wir haben die Logik implementiert, um die Anzahl der angemeldeten Benutzer aus unserer Datenbank abzurufen und diese Information verständlich an den Nutzer zurückzugeben. Dabei haben wir darauf geachtet, dass der Code sauber und übersichtlich bleibt, damit wir ihn zukünftig leicht anpassen können. Nachdem dies implementiert war, haben wir den Skill innerhalb der Alexa Developer Console getestet. Es ist erstaunlich zu sehen, wie Alexa auf unsere Befehle reagiert und die richtigen Informationen liefert.

Um weitere Alexa-Aufrufe integrieren zu können, versuchten wir ebenfalls, den Login unserer Anwendung in Alexa zu integrieren. Dies stellte sich jedoch als besondere Schwierigkeit dar. Alexa schaffte es nicht, die übergebenen Werte—in unserem Fall E-Mail und Passwort—zuverlässig an unser Backend zu übermitteln. Dies machte es schwierig, unsere Alexa-Anbindung zu erweitern, da man für weitere Funktionen nutzerspezifische Anforderungen benötigt.



## **5. Wissenschaftliche Reflexion**

### **5.1. Lennart Pfeiler**

In diesem Projekt übernahm ich sowohl die Rolle des Product Owners als auch des Entwicklers. Während diese Arbeitsweise zwar mit einem hohen Maß an Stress verbunden war, bot sie gleichzeitig die Möglichkeit abwechslungsreiche Aufgaben erledigen zu können. In meiner Rolle als Product Owner konnte ich erstmals Einblicke in die verschiedenen Aufgabenbereiche innerhalb eines Scrum Teams gewinnen, insbesondere in die Planung und Priorisierung von Anforderungen. Dies ermöglichte es mir, meine Kompetenzen im Bereich der Anforderungsmanagement erheblich zu erweitern. Trotz den traditionellen Aufgabenzuweisungen eines POs, war es mir während den administrativen Aufgaben besonders wichtig, dass die organisatorischen Abläufe sowie Vorgehensweisen in enger Abstimmung mit Arvid und Robert erfolgten. Dadurch konnten wir sicherstellen, dass trotz der verschiedenen Rollen innerhalb des SCRUM Teams wichtige Entscheidungen gemeinsam diskutiert und getroffen werden konnten. Diese kontinuierliche Kommunikation erfolgte in den regelmäßigen Scrum-Meetings, in denen wir die geplanten Aufgaben überprüften und gegebenenfalls Anpassungen vornahmen. Diese klare Strukturierung und Abstimmung der Aufgaben trug trotz unerwarteten Herausforderungen maßgeblich dazu bei, das Projektziel zu erreichen.

Neben den organisatorischen Tätigkeiten lag meine Hauptaufgabe jedoch als Entwickler in der Programmierung, insbesondere in der Entwicklung des Backends, der Implementierung des JavaScript-Codes für das Frontend sowie der API und GraphQL-Anbindungen an die SPA. Die Umsetzung des JAVA-Code sehe ich hierbei als meine Stärke, da ich aufgrund meiner vorherigen Programmierausbildung viele Vorerfahrung im Bereich der objektorientierten Programmierung mit C# sammeln konnte. Bei der Umsetzung selbst habe ich einen stets parallelen Entwicklungsprozess aus Backend und Frontend gewählt. Diese gleichzeitige Bearbeitung der verschiedenen Aufgaben erwies sich dabei als vorteilhaft, da ich so entwickelte API-Endpunkte direkt in der Benutzeroberfläche, vorausgesetzt diese war zu diesem Zeitpunkt bereits fertig, einbinden und testen konnte. Diese Vorgehensweise beschleunigte den Entwicklungsprozess und erhöhte die Qualität der Implementierung, da potenzielle Fehler schnell identifiziert und behoben werden konnten. Allerdings zeigte sich hierbei auch eine Herausforderung, dass es durch die gleichzeitige Arbeit an verschiedenen Aspekten gelegentlich zu Verwirrungen kam. Dies führte dazu, dass ich meine Aufgaben neu strukturieren musste, um den Überblick zu behalten. Letztlich habe ich aus dieser Erfahrung gelernt, dass eine strukturierte Herangehensweise bei komplexen Aufgaben, wie zum Beispiel der Implementierung des LIFO-Prinzips beim Kauf und Verkauf von Aktien, unerlässlich ist. In Zukunft werde ich bei der Umsetzung umfangreicher Methoden zunächst einen detaillierten Ablaufplan erstellen, der als Orientierung während der Implementierung dient. Diese Maßnahme hätte mir in diesem Projekt

sicherlich geholfen, bestimmte Verzögerungen zu vermeiden und die Qualität meiner Arbeit weiter zu verbessern.

Rückblickend bin ich zufrieden auf das Ergebnis, das wir als Team erreicht haben. Trotz zahlreicher Herausforderungen gelang es uns, einen voll funktionsfähigen Demobroker zu entwickeln, der die wesentlichen Anforderungen erfüllt. Meiner Meinung nach war die kontinuierliche Kommunikation sowie gegenseitige Hilfe entscheidend für den Ausgang dieses Projekts. Aus diesen gemachten Erfahrungen ziehe ich wichtige Erkenntnisse, die ich in künftigen Projekte einfließen lassen möchte. Beispielsweise würde ich in einem ähnlichen Projekt darauf achten, zu Projektbeginn detailliertere Zeitplanungen durchzuführen, an welche sich alle Teammitglieder halten müssen, um potenzielle Probleme frühzeitig zu erkennen zu können. Zudem würde ich versuchen meine investierte Zeit am Projekt intensiver zu planen, auch hinsichtlich der anstehenden Klausurenphase. Insgesamt habe ich sehr viel Zeit in das Projekt gesteckt, was vor allem gegen Ende zu hohem Stress geführt hat. Aufgrund dessen würde ich Studierenden des nächsten Jahrgangs empfehlen, sich von Beginn an eine klare Struktur für das Projekt zu schaffen und regelmäßig Meilensteine zu setzen. Bei der Umsetzung selbst ist eine offene Kommunikation im Team und eine konsequente Umsetzung der Vorgehens-Prinzipien (in unserem Fall SCRUM) ebenfalls von entscheidender Bedeutung.

## **5.2. Marko-Robert Szönyi**

Während dieses Projekts habe ich die Verantwortung für die Entwicklung des Frontends übernommen und eng mit dem Backend zusammengearbeitet. Die enge Abstimmung in unserem Scrum-Team ermöglichte es uns, die Entwicklung effizient voranzutreiben und sicherzustellen, dass Frontend und Backend nahtlos zusammenpassen.

Eine der größten Herausforderungen bestand darin, das richtige Design mit HTML und CSS umzusetzen. Besonders die Arbeit mit Media Queries und Bootstrap erwies sich als knifflig. Es war nicht immer einfach, die Seite auf verschiedenen Geräten und Bildschirmgrößen optimal darzustellen. Ich habe viel Zeit damit verbracht, die passenden Breakpoints zu finden und sicherzustellen, dass alle Elemente korrekt skaliert werden. Oft kam es vor, dass sich Komponenten überlappten oder nicht richtig angeordnet waren, was zusätzlichen Aufwand für Debugging und Anpassungen erforderte. Durch hartnäckiges Testen und Anpassen der CSS-Regeln konnte ich letztendlich eine responsive Benutzeroberfläche schaffen, die auf allen Geräten gut aussieht und funktioniert.

Ein weiteres komplexes Element war die Integration von Alexa in unsere Website. Die Einbindung stellte sich als schwieriger heraus als erwartet, da es wenige Ressourcen und Dokumentationen gab, die auf unsere spezifische Situation passten. Es erforderte viel Recherche und Experimentieren, um eine Lösung zu finden, die sowohl technisch

machbar als auch benutzerfreundlich ist. Letztendlich gelang es mir, Alexa erfolgreich zu integrieren.

Bei der Entwicklung der Single Page Application (SPA) habe ich mithilfe von TypeScript die Routen und App-Komponenten in Angular aufgebaut. Das Management der Navigation war dabei besonders herausfordernd, da wir unterschiedliche Ansichten für eingeloggte und nicht eingeloggte Nutzer bereitstellen wollten. Durch die Verwendung von Angular Directives und einer sorgfältigen Planung der Routing-Struktur konnte ich eine intuitive Navigation entwickeln, die es den Nutzern ermöglicht, mühelos zwischen den verschiedenen Seiten zu wechseln.

Rückblickend habe ich aus diesem Projekt viel gelernt, insbesondere wie wichtig es ist, frühzeitig klare Vorstellungen vom Design und den Funktionen der Website zu haben. Ständige kleine Änderungen können den Zeitplan erheblich beeinflussen und zusätzlichen Stress verursachen. Für zukünftige Projekte würde ich daher empfehlen, das Design und die Erwartungen so früh wie möglich detailliert zu besprechen. Das hilft nicht nur, das Projekt effizienter umzusetzen, sondern reduziert auch den Bedarf an nachträglichen Anpassungen.

Die Arbeit im Scrum-Team hat mir erneut gezeigt, wie wertvoll regelmäßige Kommunikation und Zusammenarbeit sind. Durch tägliche Abstimmungen konnten wir Probleme schnell identifizieren und gemeinsam Lösungen finden. Ich bin stolz auf das, was wir als Team erreicht haben, und freue mich darauf, die gewonnenen Erfahrungen in zukünftigen Projekten anzuwenden.

### **5.3. Arvid Harang**

Während des Projekts hatte ich die doppelte Verantwortung, sowohl als Scrum Master als auch als Entwickler zu agieren. Diese Doppelrolle brachte einige Herausforderungen, aber auch wertvolle Erfahrungen mit sich, die ich im Folgenden reflektieren möchte.

Als Scrum Master war es meine Aufgabe, das Team zu unterstützen und den Entwicklungsprozess zu organisieren. Ich habe eng mit Lennart zusammengearbeitet, um ihm beim Pflegen der Backlogs zu unterstützen. Dabei ging es nicht nur darum, die User Stories klar und verständlich zu formulieren, sondern auch darum, sicherzustellen, dass sie priorisiert und bereit für die Sprints waren. Eine meiner wichtigsten Aufgaben als Scrum Master bestand darin, das Team bei der Priorisierung und Zuweisung der User Stories zu den jeweiligen Sprints zu unterstützen und sicherzustellen, dass die Umsetzung im Einklang mit den Scrum-Prinzipien erfolgte. Dies erforderte eine sorgfältige Abstimmung mit dem Team, um die Kapazitäten und Prioritäten zu berücksichtigen. Darüber hinaus habe ich die Sprint Meetings moderiert, darunter die Sprint Planning Meetings, die täglichen Stand-Ups und die Sprint Retrospektiven. Hierbei war es wichtig, eine offene Kommunikation zu fördern und sicherzustellen, dass alle

Teammitglieder die Möglichkeit hatten, ihre Meinung zu äußern. Als Scrum Master habe ich versucht, Hindernisse zu beseitigen und das Team zu ermutigen, sich auf die Ziele jedes Sprints zu konzentrieren. Insgesamt hat mir diese Rolle geholfen mein Organisationstalent weiterzuentwickeln.

Als Entwickler lag mein Schwerpunkt zunächst auf dem Design des Frontends. Zusammen mit den anderen Teammitgliedern haben wir das Layout mit HTML und CSS entworfen und dabei versucht, eine benutzerfreundliche und ansprechende Oberfläche zu gestalten. Im Laufe des Projekts habe ich auch im Backend unterstützt. Dabei konnte ich mein Wissen erweitern, indem ich bei der Implementierung einer Methode und deren Einbindung im MappingController half. Eine der herausforderndsten, aber auch bereicherndsten Aufgaben war die vollständige Implementierung von GraphQL. Dies beinhaltete nicht nur das Erstellen von Abfragen und Mutationen, sondern auch die Dokumentation der Schnittstellen, um sicherzustellen, dass andere Entwickler und Nutzer die Funktionalität leicht nachvollziehen konnten. Die größte Herausforderung hierbei war die Analyse unserer bestehenden Rest-API und deren Implementierung in GraphQL. Da sich die Anforderungen während des Entwicklungsprozesses oft änderten, musste ich hier flexibel reagieren und die Abfragen und Mutationen ändern. Der Prozess hat mich gelehrt, wie wichtig es ist, strukturierte und verständliche Dokumentationen zu erstellen, besonders bei der Arbeit mit komplexen Technologien wie GraphQL.

Gegen Ende des Projekts übernahm ich die Integration der Alexa-API. Hier habe ich mich mit der Herausforderung konfrontiert gesehen, neue Technologien in das bestehende System zu integrieren. Leider konnte ich diese Aufgabe aufgrund von Zeitmangel nicht abschließen. Obwohl es frustrierend war, die Arbeit nicht vollständig abschließen zu können, habe ich aus dieser Erfahrung gelernt, wie wichtig eine realistische Zeitplanung und Priorisierung ist, insbesondere bei der Integration neuer Technologien in ein laufendes Projekt.

Insgesamt war das Projekt eine wertvolle Erfahrung für mich. Als Scrum Master konnte ich meine Kommunikations- und Organisationsfähigkeiten weiterentwickeln, während ich als Entwickler meine technischen Fähigkeiten ausbauen und neue Technologien erlernen konnte. Die Doppelrolle stellte eine besondere Herausforderung dar, da ich oft zwischen der Koordination des Teams und der Umsetzung technischer Aufgaben wechseln musste. Dennoch bin ich stolz auf das, was ich erreicht habe, und nehme viele wertvolle Lektionen mit, insbesondere in Bezug auf Zeitmanagement und die Integration neuer Technologien.