Anlagendashboard Verwaltungsapplikation

Dokumentation Teil 1

Projektname:	Anlagendashboard Verwaltungsapplikation
Projektmitglieder:	Jonas Vetsch
	Simon Leutert
Datum:	16.01.2026
Firma:	Die Schweizerische Post

1 Abstract (Kurzbeschreibung)

Das Projekt «Anlagendashboard Verwaltungsapplikation» umfasst die Entwicklung einer Java-basierten Anwendung zur Aus- und Eingabe von grundlegenden Daten des Anlagendashboards. Anlegerinnen und Anleger, die Konten bei verschiedenen Brokern führen, erhalten dank dem Anlagendashboard eine zentrale Übersicht über ihren Gesamtwert und die Performance ihrer Investitionen.

Die Applikation ist keine vollwertige Web-API, will aber ebenfalls JSON-basierte Ein- und Ausgaben ermöglichen. Damit dient dieses Projekt auch als Vorbereitung im Hinblick auf ein mögliches Folgeprojekt im Bereich Backend.

Wir verwenden eine Schichtenarchitektur (Controller, Service, DAO, DTO) und nutzen eine MySQL-Datenbank, die über JDBC angebunden ist. Kernfunktionen (UC-01) beinhalten die automatisierte Berechnung des Portfoliowerts sowie der absoluten und relativen Performance auf Basis aktueller Kursdaten. Die Ausgabe erfolgt strukturiert im JSON-Format, während die Steuerung über eine JavaFX-Benutzeroberfläche realisiert wird.

Inhaltsverzeichnis

1 Abstract (Kurzbeschreibung) 1
2 Anforderungsanalyse 4
2.1 Zielgruppe 4
2.2 Pflichtenheft 4
2.2.1 Use-Case - Diagramm 4
2.2.2 UC-01 5
2.2.3 UC-02 6
2.3 Vom User-Interface zum Datenmodell 7
2.4 Applikationsarchitektur 10
2.4.1 Das Schichtenmodell entsteht 10
2.4.2 Das definitive Konzept für das Schichtenmodell 11
2.4.3 Das Schichtenmodell des Anlagendashboards 12
2.5 Methoden / Vertrag (API-Design) 13
2.5.1 Das «Hello World» des Anlagendashboards 13
2.5.2 Portfoliokennzahlen abrufen 13
2.5.3 Eine Anlage kaufen 13
3 Projektplanung 14
3.1 Meilensteine 14

Abbildungsverzeichnis

Abbildung 1 Use Case Diagramm 4
Abbildung 2 Handskizzen des User Interfaces des Anlagendashboards, mehrere Views 7
Abbildung 3 Ideensammlung für erweiterte Funktionen (Visionen für Features für zukünfige Projekte) 8
Abbildung 4 Handskizze als Begleitung während dem Prozess der Priorisierung als Muss- und Kannziele 9
Abbildung 5 Handskizze: Erster Entwurf des Schichtenmodells für das Anlagendashboard 10
Abbildung 6 Skizze des definitiven Konzepts für das Schichtenmodell des Anlagendashboards.11

2 Anforderungsanalyse

2.1 Zielgruppe

Die Zielgruppe der Datenbank «Anlagendashboard» sind private Anlegerinnen und Anleger, die Wertpapiere, ETFs, Rohstoffe oder andere Anlagen bei mehreren Brokern (z. B. PostFinance, BEKB, Neobroker) halten und einen zentralen Überblick über ihr Gesamtportfolio wünschen. Diese Nutzer sind bereit, ihre Käufe und Verkäufe manuell zu erfassen und verfolgen in der Regel eine langfristige Anlagestrategie (Buy-and-Hold) mit wenigen, gezielten Transaktionen pro Jahr. Sie möchten ihre verschiedenen Depots nicht mehr einzeln prüfen müssen, sondern ihre Anlagen konsolidiert auswerten können. Im Vordergrund steht für sie, den aktuellen Wert des eigenen Portfolios, die bisherige Performance sowie die Verteilung nach Anlagetypen oder Brokern nachvollziehbar dargestellt zu bekommen. Typischerweise gehören dazu auch Situationen wie Steuererklärungen oder periodische Überprüfungen der eigenen Anlagestrategie, bei denen eine saubere, historische Nachvollziehbarkeit der Transaktionen und der Wertentwicklung unterstützt werden soll.

2.2 Pflichtenheft

2.2.1 Use-Case - Diagramm

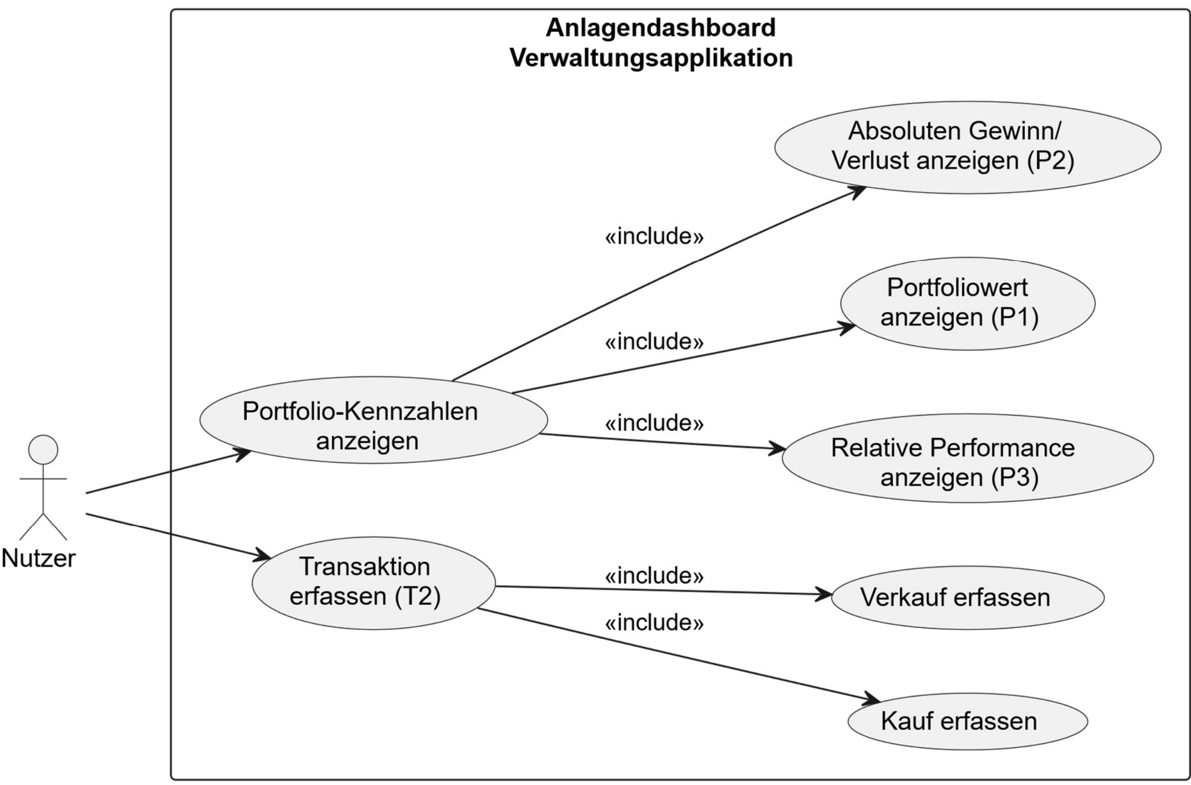


Abbildung 1 Use Case Diagramm

2.2.2 UC-01

Was	Beschreibung
Beschreibung	Der Nutzer fordert die Berechnung zentraler Portfolio-Kennzahlen an. Das System ermittelt auf Basis aller offenen Positionen den aktuellen Gesamtwert des Portfolios, den absoluten Kursgewinn/-verlust sowie die relative Performance in Prozent und gibt diese Daten strukturiert als JSON aus.
Beteiligte Akteure	**Primärakteur:** Nutzer • **Sekundärakteur:** System (Anlagendashboard-Verwaltungsapplikation)
Verwendete Anwendungsfälle (includes)	P1 – Portfoliowert anzeigen • P2 – Absoluten Kursgewinn/-verlust anzeigen • P3 – Relative Performance anzeigen
Auslöser	Der Nutzer möchte den aktuellen Wert und die Performance seines Portfolios einsehen.
Vorbedingungen	Der Nutzer existiert in der Tabelle NUTZER; es existieren Einträge in NUTZER_ANLAGE; zu den referenzierten Anlagen existieren Kursdaten in KURS.
Standardablauf	1. Der Nutzer gibt im GUI eine Nutzer-ID ein. 2. Der Nutzer löst den Anwendungsfall über einen Button aus. 3. Das System prüft syntaktisch die Nutzer-ID (Datentyp, leer/nicht leer). 4. Das System validiert die Existenz der Nutzer-ID in der Tabelle NUTZER. 5. Das System lädt alle Datensätze aus NUTZER_ANLAGE mit ZeitstempelVerkauf IS NULL für den Nutzer. 6. Das System ermittelt für jede referenzierte ANLAGE_ID den aktuellsten Kurs aus KURS. 7. Das System berechnet für jede offene Position den Positionswert (Anzahl × aktueller Kurs). 8. Das System summiert alle Positionswerte zum aktuellen Gesamtportfoliowert (P1). 9. Das System ermittelt für jede offene Position den Kaufkurs zum Kaufzeitpunkt und berechnet den investierten Gesamtbetrag. 10. Das System berechnet den absoluten Kursgewinn/-verlust (Gesamtwert – investierter Betrag) (P2). 11. Das System berechnet die relative Performance in Prozent (absolut ÷ investiert × 100) (P3). 12. Das System kapselt alle berechneten Werte in einem DTO. 13. Das System serialisiert das DTO in einen JSON-String. 14. Das System gibt den JSON-String über die Konsole aus.

2.2.3 UC-02

Was	Beschreibung
Beschreibung	Der Nutzer erfasst eine neue Transaktion in Form eines Kaufs oder Verkaufs. Das System validiert die Eingaben fachlich und technisch und speichert die Transaktion konsistent in der bestehenden Datenbank.
Beteiligte Akteure	**Primärakteur:** Nutzer **Sekundärakteur:** System (Anlagendashboard-Verwaltungsapplikation)
Verwendete Anwendungsfälle (includes)	Kauf erfassen / Verkauf erfassen
Auslöser	Der Nutzer möchte sein Portfolio durch das Erfassen eines Kaufs oder Verkaufs aktualisieren.
Vorbedingungen	Der Nutzer existiert in NUTZER; die Anlage existiert in ANLAGE; bei einem Verkauf ist ein ausreichender Bestand in NUTZER_ANLAGE vorhanden.
Standardablauf	1. Der Nutzer öffnet im GUI die Eingabemaske für Transaktionen. 2. Der Nutzer wählt den Transaktionstyp (Kauf oder Verkauf). 3. Der Nutzer gibt Nutzer-ID, Anlage-ID, Anzahl und Transaktionsdatum ein. 4. Der Nutzer bestätigt die Eingabe über einen Button. 5. Das System prüft die Eingaben auf syntaktische Korrektheit (Datentypen, Pflichtfelder). 6. Das System validiert die Existenz der Nutzer-ID in NUTZER. 7. Das System validiert die Existenz der Anlage-ID in ANLAGE. 8. Falls Transaktionstyp = Verkauf, prüft das System den aktuellen offenen Bestand der Anlage für den Nutzer. 9. Das System bricht den Vorgang ab, falls der Bestand für einen Verkauf nicht ausreicht. 10. Das System erstellt einen neuen Datensatz in NUTZER_ANLAGE (bei Kauf) bzw. aktualisiert den bestehenden Datensatz gemäss Verkaufslogik. 11. Das System schreibt die Transaktion mittels DAO und JDBC in die Datenbank. 12. Das System erstellt ein Bestätigungs-DTO mit Transaktionstyp, Anlage, Anzahl und Zeitstempel. 13. Das System serialisiert das DTO in einen JSON-String. 14. Das System gibt die Bestätigung als JSON über die Konsole aus.

2.3 Vom User-Interface zum Datenmodell

Für die konzeptionelle Ausarbeitung der Verträge («API-Endpunkte», wobei wir erst die Vorbereitungen für eine API entwickeln und noch nicht die API selbst), die unsere Applikation einhalten muss, haben wir zuerst ein User-Interface auf Papier skizziert. Die nachfolgende Abbildung zeigt unsere Handskizze mit mehreren Views des Anlagendashboards. Oben rechts ist die Dashboard-View zu sehen (Hauptseite), unten links Ideen für detaillierte Portfolioanalysen mit Kuchendiagrammen. Unten rechts ist eine Detailansicht für eine Anlage (z.B. VWRL.SIX). Sie zeigt den Kursverlauf, allgemeine Informationen als Fliesstext und eine Übersichtstabelle über von mir gehaltene Anteile dieser Anlage.

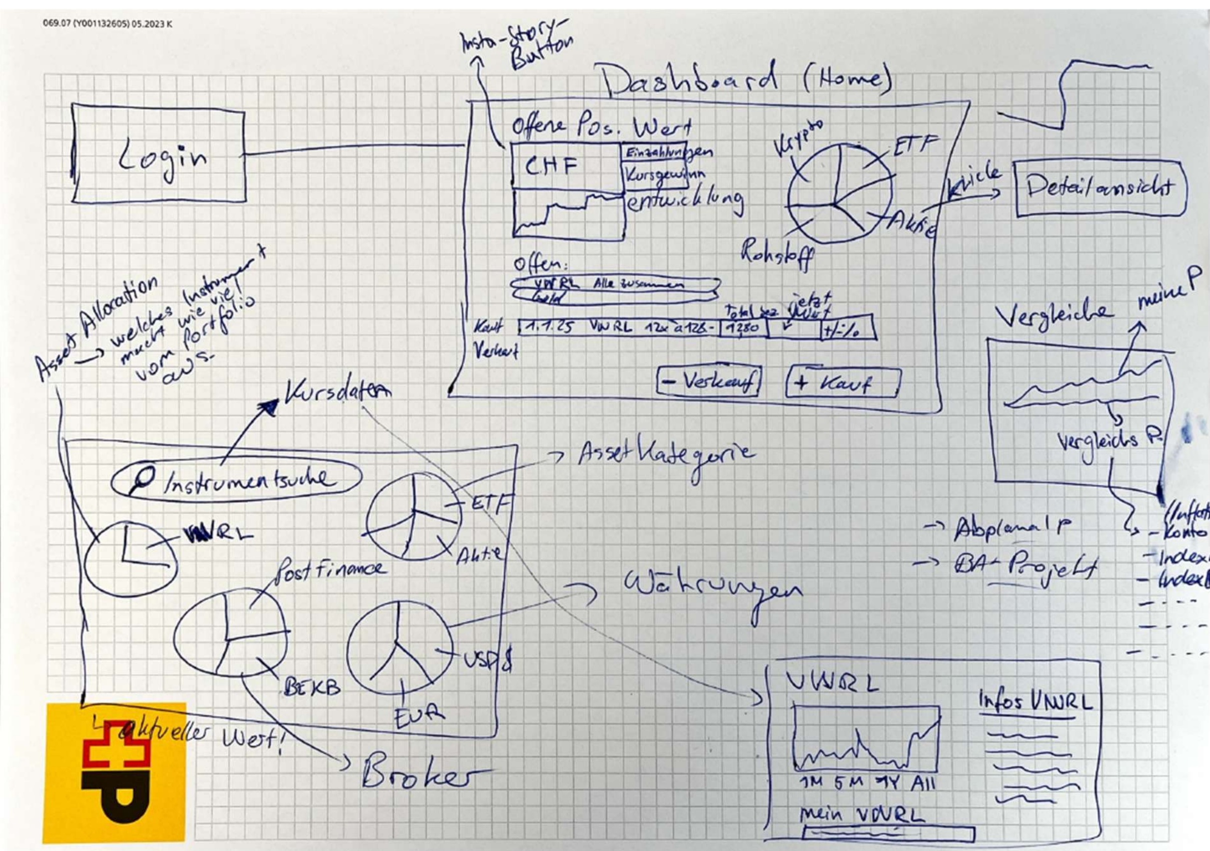


Abbildung 2 Handskizzen des User Interfaces des Anlagendashboards, mehrere Views

Danach haben wir noch eine Stufe weitergedacht: Welches wären erweiterte Funktionen, die unser Anlagendashboard in Zukunft haben könnte? Wir finden, dass spezialisierte Performanceindikatoren wie der MWR und der TWR für die Nutzer wertvoll wären. Zudem könnten auch ein Cashflow Diagramm, ein Yearly Wrapped (Jahresrückblick zum Portfolio am Ende eines Jahres) und eine Simulationsengine (z.B. Zinseszinsrechner und Vergleich mit der normalen Inflation) einen grossen Mehrwert für unsere Nutzer bieten.

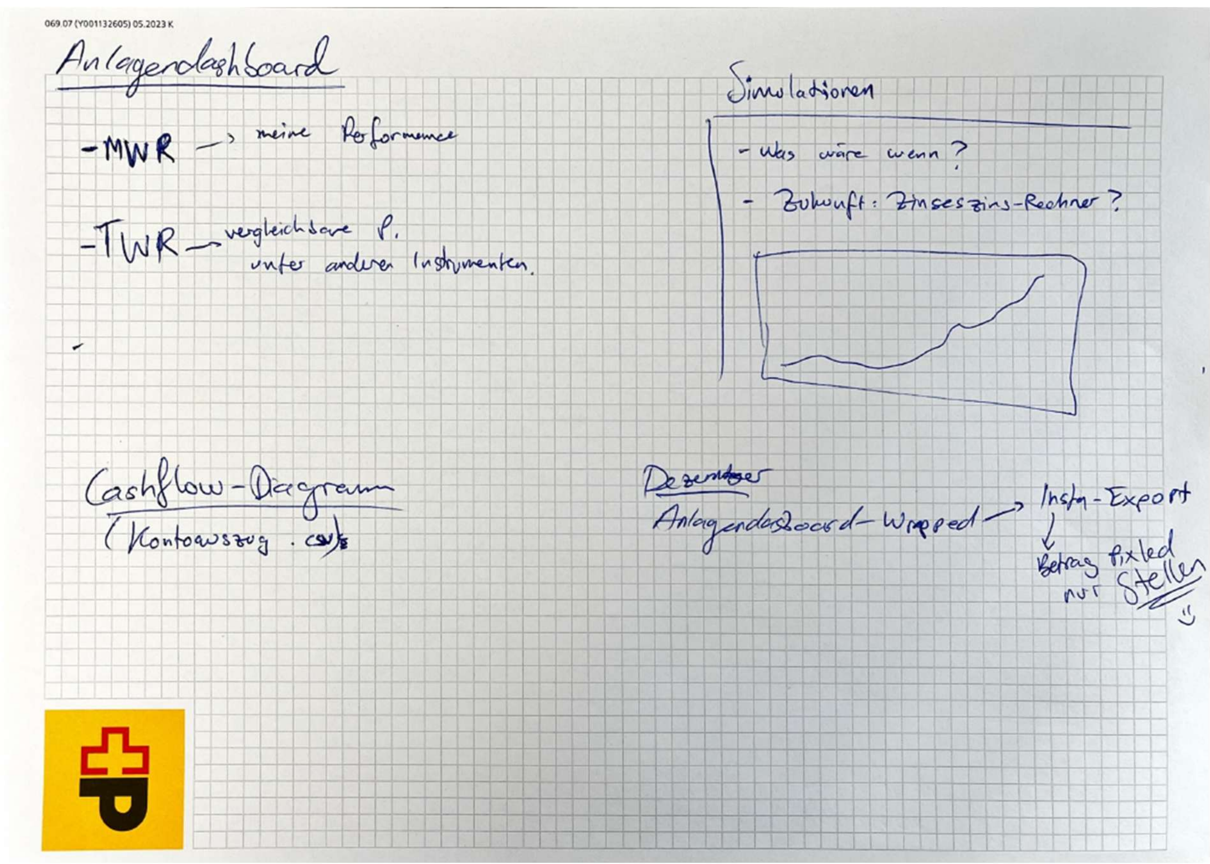


Abbildung 3 Ideensammlung für erweiterte Funktionen (Visionen für Features für zukünfige Projekte)

Aus diesem Brainstorming haben wir die wichtigsten Kernfunktionen des Anlagendashboards identifiziert und priorisiert. Damit konnten wir festlegen, welches die zentralen Daten und Interaktionen für die Nutzer unseres Anlagendashboards sein werden.

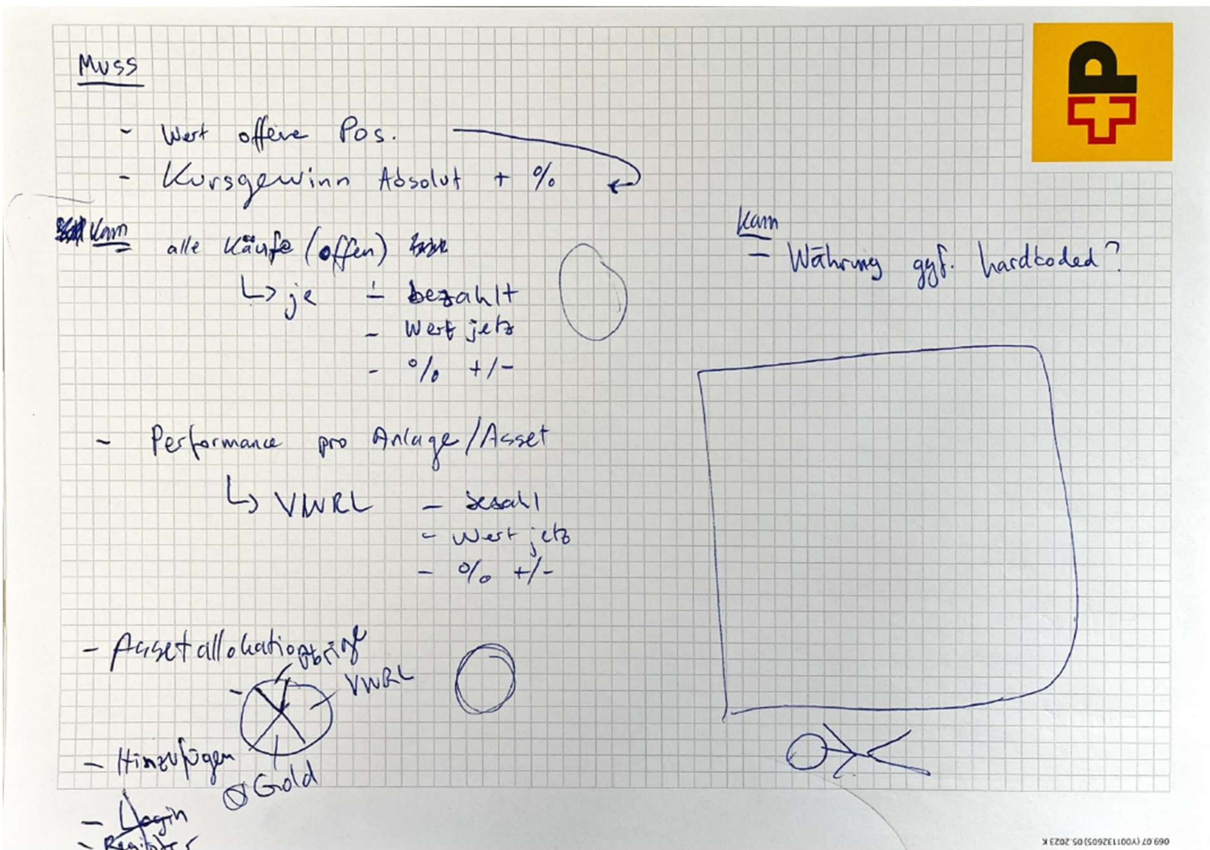


Abbildung 4 Handskizze als Begleitung während dem Prozess der Priorisierung als Muss- und Kannziele

Daraus sind die Muss- und Kann-Kriterien für unser Projekt entstanden. Anhand dieser konnten wir dann die Methoden und Rückgabeobjekte (DTOs) definieren.

2.4 Applikationsarchitektur

Als Architektur unserer Applikation möchten wir ein klassisches Schichtenmodell implementieren. Für uns ist die Erweiterbarkeit in diesem Projekt sehr wichtig, so wie auch das vorausgehende Datenbank-Miniprojekt «Anlagendashboard» der Fall war. Das Schichtenmodell hat den entscheidenden Vorteil, dass mit zunehmender Komplexität und Reife die Einzelteile der Applikation dennoch modular bleiben. In späteren Projekten könnten wir so die Datenbanktechnologie tauschen oder das Nutzerinterface tauschen und der Rest der Applikation müsste kaum angepasst werden. Beispielsweise könnten wir in Zukunft eine NoSQL-Datenbank im Hintergrund betreiben (statt MySQL aktuell) und das Frontend als Web-Applikation bereitstellen und der Kern der hier vorgelegten Applikation würde weiterhin funktionieren.

2.4.1 Das Schichtenmodell entsteht

Die nachfolgende Abbildung zeigt unseren ersten Ansatz bei der Konzeption unseres Schichtenmodells. Da es für uns beide noch sehr neu ist, mussten wir selbst zuerst die Theorie und die Einzelheiten des Schichtenmodells verstehen. Zudem ist es eine Projektvorgabe, ein JavaFX-GUI zu implementieren. Dieses haben wir ebenfalls in die erste Skizze mitaufgenommen.

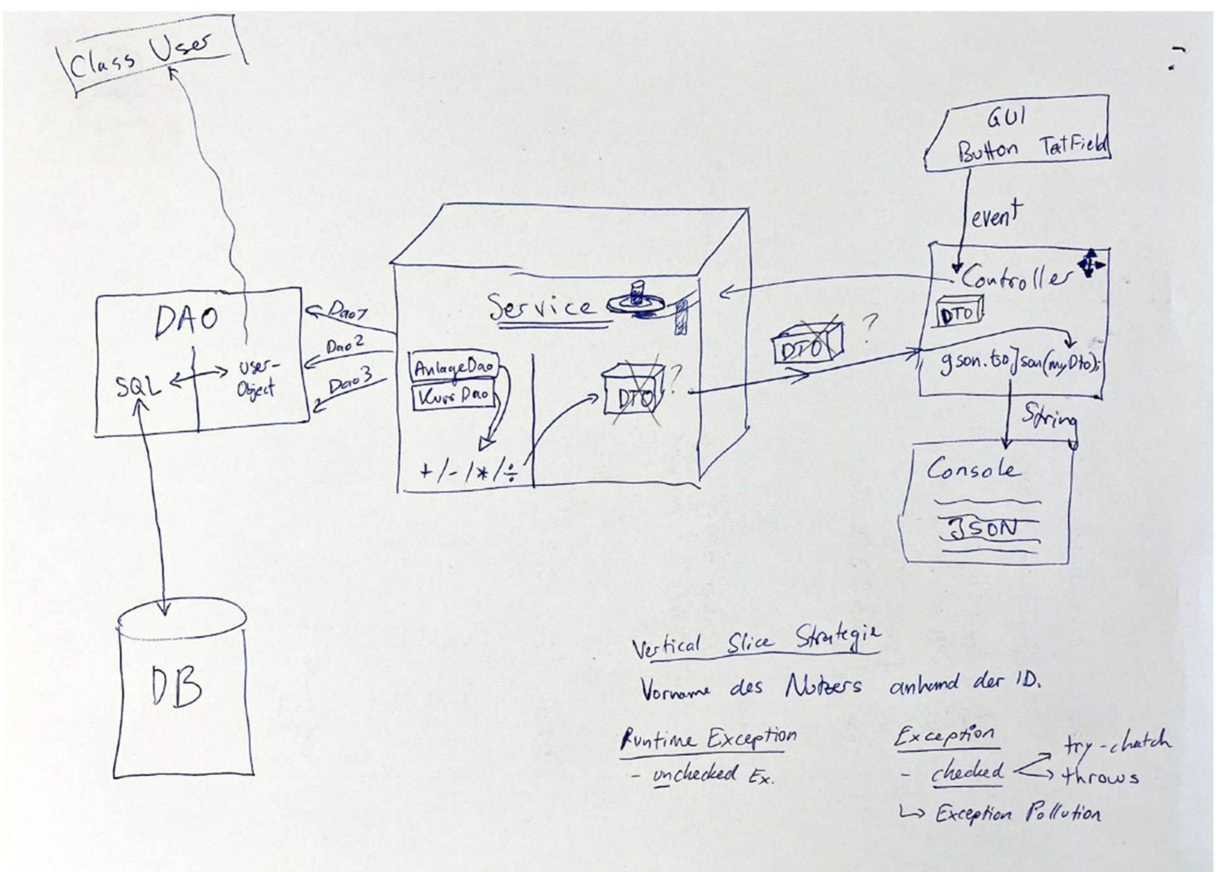


Abbildung 5 Handskizze: Erster Entwurf des Schichtenmodells für das Anlagendashboard

2.4.2 Das definitive Konzept für das Schichtenmodell

Nach umfassender Recherchearbeit und einem Test mit dem Casino-Projekt aus der BA wurde uns schliesslich das Schichtenmodell klar und wir konnten das finale Schichtenmodell zeichnen. Damit waren wir bereit für die Entwicklungsarbeit.

Abbildung 6 Skizze des definitiven Konzepts für das Schichtenmodell des Anlagendashboards

2.4.3 Das Schichtenmodell des Anlagendashboards

Unsere Applikation möchten wir in die folgenden Schichten gliedern.

1. Datenbank: MySQL Datenbank, bereits bestehend aus dem Datenbankprojekt, welches wir in der Basisausbildung umgesetzt haben.
2. Datenbankverbindung mit JDBC (Java Database Connectivity) als ConnectionFactory: Mit der JDBC-Bibliothek bauen wir die Verbindung zur Datenbank auf, rufen Daten ab und speichern Daten in die Datenbank.
3. Data Access Objects (DAO): Die Datenbankverbindung und den Datenzugriff führen wir in sogenannten DAOs aus. Pro Klasse im Model (z.B. Nutzer) erstellen wir ein DAO, welches sich um die Abfrage und Speicherung von Nutzerdaten kümmert. Dabei definieren wir aus Gründen der Modularität zuerst ein Interface, gegen welches wir danach Programmieren.
4. Serviceschicht: Die Serviceschicht ruft die Methoden der DAO-Interfaces auf. Die Serviceschicht kann auch Daten aus mehreren DAOs sammeln, sollte eine Anfrage an unsere Applikation dies erfordern. Es ist danach die Aufgabe der Serviceschicht, die geladenen Daten miteinander zu verrechnen (falls nötig) und danach sinnvoll zu verpacken, damit dem Aufrufer unseres Services auch nur die notwendigen Daten ausgegeben werden.
5. Data Transfer Objects (DTO): Neben den Klassen des Models, die die Tabellen der Datenbank nachbilden, erstellen wir für jede Ausgabe unseres Services ein DTO, damit wir die auszugebenden Daten in ein Objekt speichern können.
6. Controller: Der Controller nimmt die Anfragen an unsere Applikation entgegen und ruft die entsprechenden Methoden im Service auf. Die vom Service zurückkommenden Daten (DTO-Objekte) wandelt er in einen JSON-String um und gibt sie dann auf der Konsole aus. Beim Einfügen von Daten in die Datenbank mittels Anfrage an unsere Applikation schickt der Aufrufer einen JSON-String mit der Anfrage mit. In diesem Fall wandelt der Controller diesen zuerst in ein DTO, damit er dieses dem Service übergeben kann.

2.5 Methoden / Vertrag (API-Design)

2.5.1 Das «Hello World» des Anlagendashboards (Muss-Kriterium)

getUserById(userId);

```
// OutputDTO: "Hello World" des Anlagendashboards
public class UserDto {
    public int userId;
    public String firstName;
    public String lastName;
    public String email;
}
```

2.5.2 Portfoliokennzahlen abrufen (Muss-Kriterium)

getPortfolioSummary(userId);

```
// OutputDTO: Portfoliokennzahlen P1, P2, P3
public class PortfolioSummaryDto {
    public int userId;
    public double totalValue;        // aktueller Gesamtwert
    public double absoluteProfit;   // Absoluter Gewinn/Verlust
    public double relativeProfit;   // Relative Performance (in %)
    public String currency;         // default immer "CHF"
}
```

2.5.3 Eine Anlage kaufen (Soll-Kriterium)

registerPurchase(PurchaseDto purchaseDto);

```
// InputDTO: Kauf erfassen T2 (Kann-Kriterium)
public class PurchaseDto {
    public int userId;
    public int anlageId;
    public int brokerId;
    public double amount;
    public LocalDate timestampBought;
}
```

3 Projektplanung

3.1 Meilensteine

Meilenstein M1 – Anforderungen & Architektur festgelegt

Terminiert auf: 16.1.26 17:00

Kriterium	Beschreibung
Bezeich-nung	M1 – Anforderungen & Architektur festgelegt
Ziel	Die fachlichen und technischen Grundlagen für die Umsetzung der Muss-Kriterien sind definiert und dokumentiert.
Inhalt / Arbeiten	• Muss-Kriterien gemäss Projektantrag (Portfoliowert, absoluter Gewinn/Verlust, relative Performance, Schichtenmodell, Ein-/Ausgabe) schriftlich festhalten. • UC-01 «Portfolio-Kennzahlen berechnen» mit Akteuren, Vorbedingungen, Standardablauf und Ergebnis beschreiben. • UC-02 «Transaktion erfassen» (Kauf/Verkauf) fachlich beschreiben (Akteure, Vorbedingungen, Standardablauf), damit der Erweiterungspfad klar ist, auch wenn UC-02 nicht zu den Muss-Kriterien gehört. • Schichtenmodell definieren (Controller, Service, DAO, DTO, Datenbank) und Verantwortlichkeiten der Schichten festlegen. • Einfaches Architektur- bzw. Klassendiagramm erstellen, das den Datenfluss für eine Beispielanfrage zeigt (z. B. findUserById).
Erreicht, wenn	• Alle fünf Muss-Kriterien sind im Dokument explizit erwähnt und erläutert. • UC-01 ist so beschrieben, dass daraus klar hervorgeht, welche Daten benötigt werden und wie sie verarbeitet werden. • UC-02 ist als zusätzlicher Anwendungsfall beschrieben, sodass klar ist, wie später Transaktionen fachlich ablaufen sollen. • Das Schichtenmodell ist textlich beschrieben und in mindestens einem Diagramm visualisiert. • Ein-/Ausgabe-Vorgaben (Eingabe über ein einfaches GUI, Ausgabe als JSON über die Konsole) sind definiert.
Bezug zu Muss-Kriterien	Legt die Grundlage für die Umsetzung der fachlichen Muss-Kriterien (Portfolioberechnungen) sowie der technischen Vorgaben (Schichtenmodell, Ein-/Ausgabe) und beschreibt zusätzlich den Erweiterungspfad über UC-02 «Transaktion erfassen».

Meilenstein M2 – JDBC-Datenzugriff umgesetzt

Terminiert auf 21.1.26 15:00

Kriterium	Beschreibung
Bezeichnung	M2 – JDBC-Datenzugriff umgesetzt
Ziel	Die Applikation kann stabil auf die bestehende Datenbank «Anlagendashboard» zugreifen.
Inhalt / Arbeiten	• JDBC-Treiber einbinden und Verbindungsparameter konfigurieren (URL, Benutzer, Passwort). • Zentrale Klasse für den Verbindungsaufbau erstellen (z. B. DatabaseConnection). • Erste DAO-Klasse implementieren (z. B. NutzerDao), die Daten aus der Tabelle NUTZER lesen kann. • Testmethode erstellen, die eine einfache SELECT-Abfrage ausführt und Resultate in der Konsole ausgibt. • Fehlerbehandlung bei Verbindungsproblemen einbauen (z. B. try/catch mit sinnvoller Meldung).
Erreicht, wenn	• Eine Connection zur bestehenden Datenbank kann erfolgreich aufgebaut werden. • Mindestens eine DAO-Methode liest reale Datensätze aus der DB und gibt sie aus. • Bei falschen Verbindungsdaten oder nicht erreichbarer DB wird der Fehler abgefangen und die Applikation stürzt nicht ab. • JDBC-Logik befindet sich ausschliesslich in der DAO-/DB-Schicht, nicht im Controller.
Bezug zu Muss-Kriterien	Technische Voraussetzung für die Berechnung der Portfolio-Kennzahlen (Muss 1–3); konkrete Umsetzung der Vorgabe „Datenzugriff mittels JDBC“ und Bestandteil der klaren Applikationsarchitektur (Muss 4).

Meilenstein M3 – Ein-/Ausgabe & Schichtenfluss nachgewiesen

Terminiert auf 22.1.26 16:00

Kriterium	Beschreibung
Bezeichnung	M3 – Ein-/Ausgabe & Schichtenfluss nachgewiesen
Ziel	Die Vorgaben zur Ein-/Ausgabe (GUI-Eingabe, JSON-Konsole) und das Schichtenmodell werden in einem einfachen End-to-End-Ablauf umgesetzt.
Inhalt / Arbeiten	• Einfaches GUI erstellen (z. B. JavaFX oder Swing) mit Eingabefeld für Nutzer-ID und Button zum Auslösen einer Anfrage. • Controller implementieren, der GUI-Ereignisse entgegennimmt und den Service aufruft. • Service implementieren, der die Eingabe (Nutzer-ID) validiert und die passende DAO-Methode nutzt. • DTO definieren, das die zurückgegebenen Daten kapselt (z. B. Name, E-Mail). • JSON-Serialisierung im Controller implementieren und JSON-String in der Konsole ausgeben. • Fehlerfall bei ungültiger oder nicht existierender Nutzer-ID berücksichtigen.
Erreicht, wenn	• Eine Anfrage für eine Nutzer-ID durchläuft alle Schichten: GUI → Controller → Service → DAO → DB → DTO → JSON → Konsole. • Bei gültiger Nutzer-ID wird ein korrekt aufgebautes JSON mit den erwarteten Feldern ausgegeben. • Bei ungültiger oder nicht existierender Nutzer-ID wird eine verständliche Fehlermeldung ausgegeben, ohne dass die Applikation abstürzt. • Das GUI bleibt bewusst einfach und erfüllt die Vorgabe eines rudimentären Eingabefelds mit Absendebutton.
Bezug zu Muss-Kriterien	Direkte Umsetzung der Muss-Vorgaben zur Ein-/Ausgabe (GUI, JSON-Konsole) und Nachweis, dass das definierte Schichtenmodell (Muss 4 und 5) in der Praxis funktioniert.

Meilenstein M4 – UC-01: Portfolio-Kennzahlen implementiert

Terminiert auf 23.1.26 16:00

Kriterium	Beschreibung
Bezeichnung	M4 – UC-01: Portfolio-Kennzahlen implementiert
Ziel	Die drei fachlichen Kernfunktionen gemäss Muss-Kriterien (Portfoliowert, absoluter Gewinn/Verlust, relative Performance) sind implementiert und lauffähig.
Inhalt / Arbeiten	• Service-Methode zur Berechnung der Portfolio-Kennzahlen für eine Nutzer-ID implementieren. • Alle offenen Positionen des Nutzers aus NUTZER_ANLAGE laden (ZeitstempelVerkauf IS NULL). • Für jede Anlage den aktuellsten Kurs aus der Tabelle KURS ermitteln. • Positionswerte berechnen (Anzahl × aktueller Kurs) und zum aktuellen Gesamtportfoliowert summieren. • Investierten Gesamtbetrag anhand der Kaufdaten berechnen (Summe der Kaufbeträge für offene Positionen). • Absoluten Gewinn/Verlust als Differenz zwischen aktuellem Portfoliowert und investiertem Betrag berechnen. • Relative Performance in Prozent berechnen und sinnvoll mit dem Sonderfall „investierter Betrag = 0“ umgehen. • Kennzahlen in einem DTO (z. B. PortfolioSummaryDto) bündeln und als JSON ausgeben.
Erreicht, wenn	• Die Applikation liefert für eine gültige Nutzer-ID einen aktuellen Gesamtportfoliowert (Muss 1). • Der absolute Kursgewinn/-verlust in CHF wird korrekt ausgegeben (Muss 2). • Die relative Portfolio-Performance in Prozent wird korrekt berechnet und ausgegeben (Muss 3). • Die Ergebnisse lassen sich mit Testdaten per Hand nachrechnen und stimmen mit der JSON-Ausgabe überein.
Bezug zu Muss-Kriterien	Direkte Umsetzung der drei fachlichen Muss-Kriterien zur Berechnung der zentralen Portfolio-Kennzahlen (aktuellem Wert, absoluter Gewinn/Verlust, relativer Performance).

Meilenstein M5 – Validierung & Stabilität der Muss-Funktionen

Terminiert auf 28.1.26 16:00

Kriterium	Beschreibung
Bezeichnung	M5 – Validierung & Stabilität der Muss-Funktionen
Ziel	Die Muss-Funktionen, insbesondere UC-01, reagieren robust auf fehlerhafte Eingaben und typische Randfälle.
Inhalt / Arbeiten	• Eingabevalidierung für Nutzer-ID (Pflichtfeld, Datentyp, sinnvolle Wertebereiche) implementieren. • Fall „Nutzer existiert nicht in NUTZER" behandeln (z. B. Fehlermeldung im JSON oder in der Konsole). • Fall „Nutzer hat keine offenen Positionen" behandeln (z. B. Kennzahlen = 0 und Hinweis in der Ausgabe). • Fall „fehlende Kursdaten in KURS" definieren und implementieren (z. B. Abbruch mit Fehlermeldung oder Ignorieren der betroffenen Position). • Exceptions aus JDBC-/DAO-Schicht im Service abfangen und in verständliche Meldungen übersetzen.
Erreicht, wenn	• Falsche oder leere Nutzer-IDs führen zu einer kontrollierten Reaktion (Fehlermeldung statt Programmabsturz). • Ein Nutzer ohne offene Positionen wird sinnvoll behandelt (kein Absturz, konsistente Kennzahlen). • Fehlende oder unvollständige Kursdaten führen zu einem definierten Verhalten, das dokumentiert ist. • Die Applikation bleibt auch in den wichtigsten Fehlerfällen stabil und bedienbar.
Bezug zu Muss-Kriterien	Stützt alle Muss-Kriterien ab, indem die implementierten Funktionen robust und realistisch einsetzbar werden und nicht nur im Happy Path funktionieren.

Meilenstein M6 – Testfälle & Nachvollziehbarkeit

Terminiert auf 29.1.26 12:00

Kriterium	Beschreibung
Bezeichnung	M6 – Testfälle & Nachvollziehbarkeit
Ziel	Die korrekte Umsetzung der Muss-Kriterien ist durch definierte Testfälle überprüfbar und für Dritte nachvollziehbar.
Inhalt / Arbeiten	• Mindestens drei Testfälle für UC-01 definieren (z. B. Nutzer mit mehreren Positionen, Nutzer mit einer Position, Nutzer ohne offene Positionen). • Für jeden Testfall die relevanten Daten (NUTZER, NUTZER_ANLAGE, KURS) und die erwarteten Kennzahlen (Gesamtwert, absoluter Gewinn/Verlust, relative Performance) festhalten. • Applikation für jeden Testfall ausführen und die JSON-Ausgaben dokumentieren (z. B. Screenshot oder Kopie). • Erwartete und tatsächliche Werte vergleichen und Abweichungen analysieren.
Erreicht, wenn	• Für alle definierten Testfälle liegen Input, erwartetes Resultat und tatsächliche JSON-Ausgabe vor. • Die berechneten Werte stimmen mit der Handrechnung überein oder Abweichungen sind nachvollziehbar begründet. • In der Dokumentation ist klar ersichtlich, dass die Muss-Kriterien 1–3 fachlich korrekt erfüllt sind.
Bezug zu Muss-Kriterien	Liefert den Nachweis, dass die Berechnungen der Portfolio-Kennzahlen (Muss 1–3) korrekt sind und im Zusammenspiel mit Architektur und Ein-/Ausgabe wie gefordert funktionieren.

Meilenstein M7 – Projektabschluss & Abgabe

Terminiert auf 29.1.26 16:00

Kriterium	Beschreibung
Bezeichnung	M7 – Projektabschluss & Abgabe
Ziel	Das Projekt ist fachlich, technisch und formal abgabebereit, alle Muss-Kriterien sind umgesetzt und dokumentiert.
Inhalt / Arbeiten	• Überprüfen, dass alle Meilensteine M1–M6 erreicht sind. • Projektstruktur aufräumen (nicht verwendete Klassen/Dateien entfernen, sinnvolle Paketstruktur). • ZIP-Archiv mit vollständigem Projekt erstellen (Quellcode, ggf. Konfigurationsdateien, falls nötig SQL-Skripte). • Installations- und Startanleitung verfassen (Voraussetzungen, DB-Verbindung konfigurieren, Applikation starten, UC-01 ausführen). • Dokumentation abschliessen und sicherstellen, dass klar ersichtlich ist, wo welche Muss-Kriterien umgesetzt sind.
Erreicht, wenn	• Ein Dritter kann mithilfe der Anleitung die Applikation starten und die Portfolio-Kennzahlen für einen Testnutzer abrufen. • Die Dokumentation verweist nachvollziehbar auf die Umsetzung der fünf Muss-Kriterien. • Die formalen Vorgaben zur Abgabe (Format, Umfang, Frist) sind erfüllt.
Bezug zu Muss-Kriterien	Führt alle vorherigen Meilensteine zusammen und stellt sicher, dass die Muss-Kriterien 1–5 vollständig, nachvollziehbar und fristgerecht umgesetzt sind.

Anlagendashboard Verwaltungsapplikation

Dokumentation Teil 2

Projektname:	Anlagendashboard Verwaltungsapplikation
Projektmitglieder:	Jonas Vetsch
	Simon Leutert
Datum:	29.01.2026
Firma:	Die Schweizerische Post

1 Abstract (Kurzbeschreibung)

Das Projekt «Anlagendashboard Verwaltungsapplikation» umfasst die Entwicklung einer Java-basierten Anwendung zur Aus- und Eingabe von grundlegenden Daten des Anlagendashboards. Anlegerinnen und Anleger, die Konten bei verschiedenen Brokern führen, erhalten dank dem Anlagendashboard eine zentrale Übersicht über ihren Gesamtwert und die Performance ihrer Investitionen.

Die Applikation ist keine vollwertige Web-API, will aber ebenfalls JSON-basierte Ein- und Ausgaben ermöglichen. Damit dient dieses Projekt auch als Vorbereitung im Hinblick auf ein mögliches Folgeprojekt im Bereich Backend.

Wir verwenden eine Schichtenarchitektur (Controller, Service, DAO, DTO) und nutzen eine MySQL-Datenbank, die über JDBC angebunden ist. Kernfunktionen (UC-01) beinhalten die automatisierte Berechnung des Portfoliowerts sowie der absoluten und relativen Performance auf Basis aktueller Kursdaten. Die Ausgabe erfolgt strukturiert im JSON-Format, während die Steuerung über eine JavaFX-Benutzeroberfläche realisiert wird.

2 Inhaltsverzeichnis

1 Abstract (Kurzbeschreibung) 1
2 Inhaltsverzeichnis 2
3 Abbildungsverzeichnis 3
4 Technische Dokumentation 5
4.1 Klassendiagramm 5
4.2 Sequenzdiagramm 6
5 Testauswertung 9
6 Installationsanleitung 10
6.1 Systemvoraussetzungen 10
6.2 Repository klonen und Projekt öffnen 10
6.3 Datenbank-Setup 10
6.4 Konfiguration der Applikation 10
6.5 Abhängigkeiten laden und Start 11
7 Benutzerhandbuch 11
7.1 Was kann das Anlagendashboard aktuell? 11
7.2 Anlagendashboard starten 11
7.3 Anlagendashboard bedienen 11
7.4 Die Zahlen verstehen 14
7.5 Mögliche Fehler 15
8 Fazit 16
8.1 Fazit Simon 16
8.2 Fazit Jonas 17

3 Abbildungsverzeichnis

Abbildung 1 UML Klassendiagramm Anlagendashboard 4
Abbildung 2 UML Sequenzdiagramm PortfolioMetricsService 6
Abbildung 3 Portfoliokennzahlen des Nutzer mit der ID 5 als JSON-String........ 7
Abbildung 4 Grafisches Interface des Anlagendashboards........11
Abbildung 5 Nutzer ID 10 wurde eingegeben........12
Abbildung 6 Mögliche Ausgabe für den Endpoint "Find User by ID" 12
Abbildung 7 Mögliche Ausgabe für den Endpoint "Get Portfo Metrics by User ID" 13
Abbildung 8 Beispielhafte Ausgabe vom Endpoint "Get Portfo Metrics by User ID" 14

Main
- void main(String[] args)

ruft auf

App
- void start(Stage stage)
- void main(String[] args)

ruft auf

ruft auf

UserController
- UserService userService
- getUserByIdEndpoint()

PortfolioMetricsController
- PortfolioMetricsService service
- getPortfolioMetricsByUserIdEndpoint()

UserService
- UserDao userDao
- getUserById()

PortfolioMetricsService
- PriceDao priceDao
- UserAssetDao userAssetDao
- getPortfolioMetricsByUserId()

nutzt

nutzt

UserDao
- findById()

UserDto
- int userId
- String firstName
- String lastName

UserAssetDao
- getOpenPositionsByUserId()

PriceDao
- getPriceBeforeTimestampByAssetID()
- getPriceAfterTimestampByAssetID()

PortfolioMetricsDto
- int userId
- double totalPortfolioValue
- double investedTotalValue
- double absolutePerformance
- double relativePerformance

nutzt

implementiert

nutzt

implementiert

nutzt

implementiert

UserModel
- int userId
- String firstName
- String lastName
- String email
- String passwordHash

UserJdbcDao
- ConnectionFactory connectionFactory
- findById()

UserAssetModel
- int id
- int userId
- int assetId
- double amount
- double priceAtPurchase

UserAssetJdbcDao
- ConnectionFactory connectionFactory
- getOpenPositionsByUserId()

PriceModel
- int assetId
- double price
- LocalDateTime timestamp

PriceJdbcDao
- ConnectionFactory connectionFactory
- getPriceBeforeTimestampByAssetID()
- getPriceAfterTimestampByAssetID()

DbProperties
- Properties props
- Properties getProperties()

nutzt

ConnectionFactory
- ConnectionFactory instance
- ConnectionFactory getInstance()
- Connection getConnection()

Abbildung 1 UML Klassendiagramm Anlagendashboard

4 Technische Dokumentation

4.1 Klassendiagramm

Das auf der vorherigen Seite abgebildete Klassendiagramm beschreibt die Architektur unserer Java-Anwendung. Die Struktur folgt einem modularen Aufbau nach dem bewährten Schichtenmodell. Unsere Anwendung haben wir in folgende Schichten unterteilt.

4.1.1 Programmstart und GUI

Der Einstiegspunkt der Anwendung ist die Klasse Main, welche die App-Klasse aufruft. Diese rendert die grafische Benutzeroberfläche (basierend auf JavaFX) und fungiert als zentraler Koordinator, der die beiden Controller (UserController, PortfolioMetricsController) instanziiert und verwaltet.

4.1.2 User-Modul

Dieses Modul verwaltet die Benutzerdaten. Das UserModel enthält die vollständigen Daten eines Users, während das UserDto als Datentransferobjekt für die Ausgabe dient und nur die notwendigen Daten enthält.

Der UserController greift auf den UserService zu, welcher wiederum ein UserDao-Interface nutzt. Die konkrete Implementierung dessen, das UserJdbcDao, kapselt den Zugriff auf die Datenbank.

4.1.3 Portfolio- und Asset-Modul

Dies ist das Kernstück für die Berechnung von Performance-Kennzahlen. Über das UserAssetDao (implementiert durch UserAssetJdbcDao) erhalten wie die gehaltenen Vermögenswerte eines Benutzers (UserAssetModel) aus der Datenbank.

Das PriceDao hält Methoden, um historische und aktuelle Kurse (PriceModel) für bestimmte Assets abzufragen.

Der PortfolioMetricsService enthält die Logik. Er nutzt die Asset- und die Preis-Daten, um Berechnungen durchzuführen. Die Ergebnisse werden im PortfolioMetricsDto gespeichert und über den PortfolioMetricsController an das GUI geliefert.

4.1.4 Datenbank-Infrastruktur (Hilfsklassen)

Die Datenhaltung wird durch Hilfsklassen unterstützt: Die ConnectionFactory ist ein Singleton, das sicherstellt, dass nur eine aktive Datenbankverbindung existiert. DbProperties lädt die notwendigen Konfigurationsdaten (URL, Benutzer, Passwort) für den Datenbankzugriff aus einer Konfigurationsdatei, welche jeder Betreiber unseres Programms selbst erstellen muss, da diese Datei von Git ausgeschlossen ist.

4.1.5 Beziehungen

Wir nutzen Data Access Objects, um die Geschäftslogik von der Datenbankanbindung zu trennen. Zudem haben wir eine klare Schichtentrennung implementiert, die die Oberfläche (Controller) von der Logik (Service), und die Logik von der Datenquelle (DAO und Model) trennt.

4.2 Sequenzdiagramm

Sequence
PortfolioMetricsService.getPortfolioMetricsByUserId()

PortfolioMetricsService
userAssetDao
priceDao
FinanceMathUtil

1 getPortfolioMetricsByUserId(...)
2 getOpenPositionsByUserId(...)
3 List<UserAssetModel> openPositions

getPositionsTotalValue(openPositions)
loop [UserAssetModel position : openPositions]
4 getPriceBeforeTimestampByAssetId(...)
5 PriceModel priceBefore
6 getPriceAfterTimestampByAssetId(...)
7 PriceModel priceAfter

getPositionsTotalInvested(openPositions)
loop [UserAssetModel position : openPositions]
8 getPriceBeforeTimestampByAssetId(...)
9 PriceModel priceBefore
10 getPriceAfterTimestampByAssetId(...)
11 PriceModel priceAfter

12 calculateAbsolutePerformance(...)
13 absolutePerformance
14 calculateRelativePerformance(...)
15 relativePerformance
16 PortfolioMetricsDto

PortfolioMetricsService
userAssetDao
priceDao
FinanceMathUtil

Abbildung 2 UML Sequenzdiagramm PortfolioMetricsService

Das abgebildete Sequenzdiagramm illustriert den Ablauf der Methode getPortfolioMetricsByUserId innerhalb des PortfolioMetricsService. Dies ist die wichtigste Methode im PortfolioMetricsService und dient dazu, die wichtigsten Leistungskennzahlen für das Portfolio eines spezifischen Benutzers zu berechnen und als PortfolioMetricsDto zurückzugeben. Das PortfolioMetricsDto wird von unserer API dann wie folgend ausgegeben.

Nutzer mit der ID 5 hat am 28.1. um 9.42 Uhr einen Portfoliogesamtwert von 1 510 274 CHF, wobei er 290 582 CHF investiert hat. Damit hat er 1 219 592 CHF Gewinn gemacht, was einer relativen Performance von 419.56% entspricht.


```
==== RESPONSE of GET /portfoliometrics/{id} ====
{
  "userId": 5,
  "timestamp": "2026-01-28T09:42:22.027571777",
  "totalPortfolioValue": 1510274.82521,
  "investedTotalValue": 290682.60102500004,
  "absolutePerformance": 1219592.224185,
  "relativePerformance": 4.195614804203948,
  "currency": "CHF"
}
```


Abbildung 3 Portfoliokennzahlen des Nutzer mit der ID 5 als JSON-String

Der Prozess, um diese Portfoliokennzahlen zu berechnen, lässt sich in folgende Phasen unterteilen.

4.2.1 Datenbeschaffung

Zunächst ruft der Service über das userAssetDao (User Asset Data Access Object) alle offenen Positionen (openPositions) des Benutzers ab. Eine Position gilt als offen, wenn sie noch nicht verkauft wurde. In der Datenbank muss also das Attribut SoldAt den Wert null haben.

4.2.2 Berechnung der Portfoliowerte

Anschliessend müssen die zwei wichtigsten Werte berechnet werden: Wie viel sind die offenen Positionen dieses Nutzers aktuell wert? Und wie viel hat der Nutzer insgesamt für den Kauf dieser Positionen bezahlt?

4.2.2.1 Aktueller Gesamtwert (getPositionsTotalValue)

In einer Schleife wird für jede offene Position der aktuelle Marktwert ermittelt. Hierzu fragt der Service beim priceDao sowohl den Preis unmittelbar vor (getPriceBeforeTimestampByAssetId) als auch unmittelbar nach (getPriceAfterTimestampByAssetId) dem aktuellen Zeitpunkt (LocalDate.now()) ab. Im Code wird dies durch die Hilfsmethode getPriceByTimeAndAssetId abstrahiert, die den zeitlich am nächsten liegenden Preis auswählt. Für den aktuellen Gesamtwert des Portfolios wird dabei dann automatisch der neuste Kurseintrag in unserer Kurstabelle für die jeweilige Anlage gelesen.

4.2.2.2 Investiertes Kapital (getPositionsTotalInvested)

Ähnlich wie beim Gesamtwert wird erneut über alle Positionen iteriert. In diesem Fall wird jedoch der historische Preis zum exakten Zeitpunkt des Kaufs (PurchasedAt) ermittelt, um die ursprünglichen Anschaffungskosten zu berechnen. Hier ist die Methode getPriceByTimeAndAssetId besonders wichtig, denn sie stellt sicher, dass die Datenbank genau den Kurs liefert, der dem gefragten Zeitpunkt am nächsten liegt.

4.2.3 Performance-Analyse und Abschluss

Nachdem der aktuelle Wert des Portfolios und die Investitionssumme feststehen, nutzt der Service die Utility-Klasse FinanceMathUtil, um die für das PortfolioMetricsDto noch fehlenden Performance-Kennzahlen zu berechnen. Die absolute Performance ist der Gewinn oder Verlust in der Währung CHF. Die relative Performance ist der Gewinn oder Verlust in Prozent.

5 Testauswertung

Alle definierten Test-Cases (ST-01 bis ST-10) wurden gemäss Testkonzept vollständig durchgeführt. Sämtliche Tests verliefen erfolgreich, das beobachtete Verhalten entsprach in allen Fällen den erwarteten Resultaten.

ID	Ergebnis	Testauswertung (Ist-Resultat)
ST-01	Erfolgreich	Bei Eingabe einer gültigen, existierenden User-ID wurde der Controller korrekt aufgerufen. In der Konsole erschien der Response-Header ==== RESPONSE of GET /users/{id} ==== gefolgt von genau einem UserDto-JSON mit korrekten Attributen. Es wurde kein ErrorDto und kein Stacktrace ausgegeben.
ST-02	Erfolgreich	Eine numerische, aber nicht existierende User-ID wurde korrekt erkannt. Der UserController gab ein ErrorDto-JSON mit einer verständlichen Fehlermeldung und Timestamp aus. Es wurde kein UserDto-JSON und keine Stacktrace-Ausgabe erzeugt.
ST-03	Erfolgreich	Für einen User mit offenen Positionen wurden die Portfolio-Kennzahlen korrekt berechnet. In der Konsole wurde ein vollständiges PortfolioMetricsDto-JSON mit gültigen numerischen Werten ausgegeben. Es trat kein Fehlerfall auf.
ST-04	Erfolgreich	Ein existierender User ohne offene Positionen wurde korrekt erkannt. Statt eines PortfolioMetricsDto wurde ein ErrorDto-JSON mit einer inhaltlich passenden Fehlermeldung ausgegeben. Es wurden keine 0-Werte berechnet und kein Stacktrace angezeigt.
ST-05	Erfolgreich	Eine nicht-numerische User-ID führte zu einer NumberFormatException im GUI-Code, die korrekt abgefangen wurde. Es erschien eine klare Fehlermeldung im Terminal. Controller-Methoden wurden nicht aufgerufen und es wurde kein JSON ausgegeben.
ST-06	Erfolgreich	Ein leeres User-ID-Feld führte ebenfalls zu einer NumberFormatException, die im GUI-Code korrekt behandelt wurde. Es wurde eine verständliche Fehlermeldung ausgegeben, ohne Controller-Aufruf, JSON-Ausgabe oder Stacktrace.
ST-07	Erfolgreich	Eine negative User-ID wurde im UserService erkannt und führte zu einer IllegalArgumentException. Der UserController fing diese korrekt ab und gab ein ErrorDto-JSON mit dem Präfix „Invalid request:" sowie einem Timestamp aus. Es wurde kein UserDto ausgegeben.
ST-08	Erfolgreich	Eine negative User-ID in den PortfolioMetrics wurde im PortfolioMetricsService korrekt validiert. Der Controller gab ein ErrorDto-JSON mit der Exception-Message und Timestamp aus. Es wurde kein PortfolioMetricsDto erzeugt und kein Stacktrace angezeigt.
ST-09	Erfolgreich	Ein gezielt provozierter Datenbankfehler bei der User-Abfrage führte zu einer RuntimeException im DAO. Diese wurde im UserController korrekt abgefangen und als ErrorDto-JSON mit passender Fehlermeldung und Timestamp ausgegeben. Es erschien kein Stacktrace.
ST-10	Erfolgreich	Ein DB-/DAO-Fehler bei den PortfolioMetrics wurde erfolgreich simuliert. Die RuntimeException aus dem DAO wurde im PortfolioMetricsController im catch-Block abgefangen. Es wurde ein ErrorDto-JSON mit message und timestamp ausgegeben, ohne PortfolioMetricsDto und ohne Stacktrace.

6 Installationsanleitung

Diese Installationsanleitung führt dich durch die notwendigen Schritte, um das „Anlagendashboard" auf einem neuen System erfolgreich in Betrieb zu nehmen.

6.1 Systemvoraussetzungen

Unsere Applikation liegt als Quellcode vor. Wir empfehlen, die IntelliJ (JetBrains) Entwicklungsumgebung zu nutzen. Wir empfehlen ein Java Development Kit (JDK) mit mindestens Version JDK 17 (LTS), wobei wir konkret mit Eclipse Temurin 21.0.9 gearbeitet haben.

6.2 Repository klonen und Projekt öffnen

Lade zunächst den Quellcode direkt aus dem Repository herunter. Klone dazu das Projekt unter der URL https://github.com/BA-2025-2026/JAVA_SimonJonas_Anlagendashboard. Öffne den geladenen Ordner anschliessend in der IntelliJ IDEA.

6.3 Datenbank-Setup

Die Applikation benötigt eine MySQL-Datenbank. Wir haben MySQL 8.0.44 verwendet und empfehlen, dieselbe Version zu verwenden. Die Datenbank richtest du wie folgt ein.

6.3.1 Datenbank erstellen

Führe zunächst die folgenden SQL-Befehle aus, um die Grundstruktur anzulegen:

```
drop database if exists java_simonjonas_anlagendashboard_db;

create database java_simonjonas_anlagendashboard_db;

use java_simonjonas_anlagendashboard_db;
```

6.3.2 Daten importieren (Dump)

Führe im Anschluss den SQL-Dump aus, um die Tabellen und Daten zu initialisieren. Die Datei befindet sich bereits im geklonten Repository unter dem Pfad

src/main/java/net/ictcampus/semodul/anlagendashboard/database/setup/260121_dump.sql.

6.3.3 Alternativer Datenimport

Sollte der Import des Dumps fehlschlagen, findest du im selben Verzeichnis die Dateien `CreateDatabase.sql` und `ImportFromCSV.sql`. Letztere Datei benötigt die Daten als .CSV-Dateien. Diese liegen im Ordner src/main/java/net/ictcampus/semodul/anlagendashboard/setup/csv bereit.

6.4 Konfiguration der Applikation

Damit die Java-Applikation auf die soeben erstellte Datenbank zugreifen kann, muss eine Konfigurationsdatei erstellt werden.

Lege im Verzeichnis src/main/resources eine Datei mit dem Namen config.properties und folgendem Inhalt an, wobei die lila eingefärbten Werte anzupassen sind.

```
db.url=jdbc:mysql://localhost:3306/java_simonjonas_anlagendashboard_db

db.user=[Datenbank-Benutzername]

db.password=[Datenbank-Passwort]
```

6.5 Abhängigkeiten laden und Start

Nachdem die Konfiguration abgeschlossen ist, müssen die Projektabhängigkeiten geladen werden. Da es sich um ein Maven-Projekt handelt, kannst du mit Maven die benötigten Dependencies (wie JavaFX oder den MySQL-Connector) automatisch bauen lassen.

Sobald der Build-Vorgang abgeschlossen ist, kannst du die Applikation über die Klasse Main.java starten. Das JavaFX-GUI sollte sich daraufhin öffnen und einsatzbereit sein. Die Ausgaben der API erfolgen über die Konsole.

7 Benutzerhandbuch

Mit dem Anlagendashboard behältst du den Überblick über deine Investments, auch wenn du bei mehreren Brokern Anlagen hältst. Das Anlagendashboard zeigt die die Performance deiner Anlagen – brokerübergreifend.

7.1 Was kann das Anlagendashboard aktuell?

Das Anlagendashboard ist wie eine API aufgebaut. Dabei kannst du die Grundinformationen zu einem bestimmten Benutzer anhand seiner ID abrufen. Und du kannst den Gesamtwert deiner Anlagen berechnen und dir Performancekennzahlen anzuzeigen lassen.

Das Programm antwortet immer in JSON-Strings, wie eine echte API es auch tun würde. Die JSON-Strings werden auf der Konsole ausgegeben. Das war eine bewusste Designentscheidung des Entwicklerteams. Denn: Dank der Ausgabe auf der Konsole behältst du den Überblick und die Historie. Du kannst nach vielen getätigten Anfragen wieder nach oben scrollen und dir deine vorherigen Anfragen ansehen.

7.2 Anlagendashboard starten

Zur Installation der Applikation befolge zuerst die Installationsanleitung im vorherigen Kapitel.

Um das Programm zu starten, führst du die Datei Main.java aus. Es öffnet sich ein Fenster mit einer einfachen grafischen Oberfläche. Diese verwendest du, um mit dem Anlagendashboard zu interagieren.

7.3 Anlagendashboard bedienen

Sobald das Fenster offen ist, siehst du ein Eingabefeld und zwei Buttons.

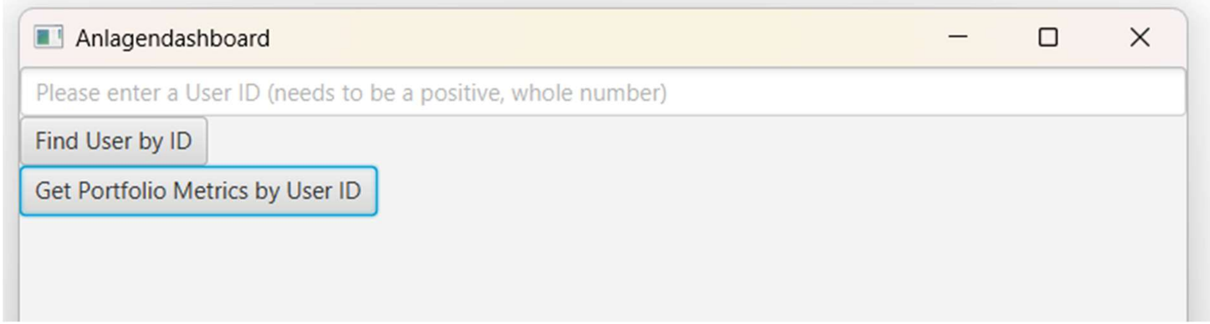


Abbildung 4 Grafisches Interface des Anlagendashboards

7.3.1 Grundinformationen eines Nutzers abfragen

Um Informationen zu einem Nutzer abzufragen, tippst du in das Textfeld die ID des Nutzers ein (z. B. 10). Die ID muss eine positive Ganzzahl sein. Danach klickst du auf den Button «Find User by ID».

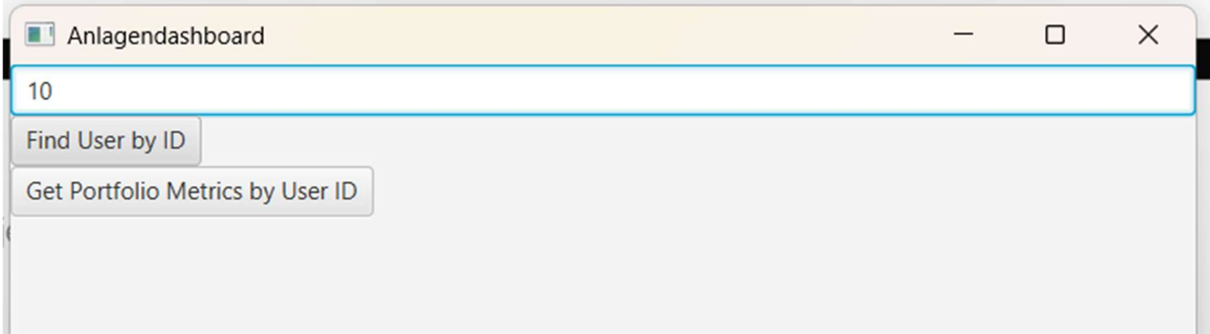


Abbildung 5 Nutzer ID 10 wurde eingegeben

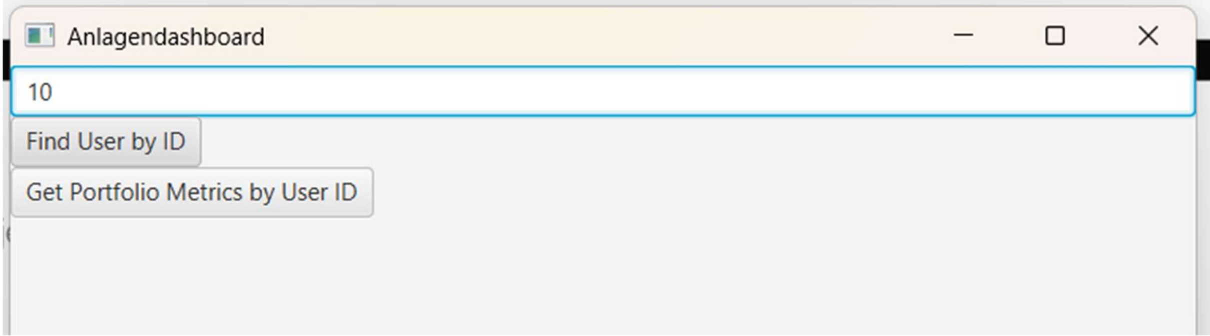
Das Programm gibt dir die Grundinformationen des Nutzers auf der Konsole aus.

```
==== RESPONSE of GET /users/{id} ====
{
  "userId": 10,
  "firstName": "Selina",
  "lastName": "Arnold",
  "email": "selina.arnold@example.com"
}
```

Abbildung 6 Mögliche Ausgabe für den Endpoint "Find User by ID"

7.3.2 Portfoliokennzahlen abfragen

Nun kommen wir zum spannenden Teil. Um Portfoliokennzahlen abzufragen, gibst du wiederum im Textfeld die ID des Nutzers ein, für den du die Anfrage stellen möchtest. Beachte, dass die ID eine positive Ganzzahl sein muss. Danach klickst du auf «Get Portfolio Metrics by User ID».



Das Programm sucht nun alle offenen Anlagen dieses Nutzers, rechnet alle aktuellen Kurse zusammen und vergleicht den aktuellen Gesamtpreis der Anlagen mit dem Preis, den der Nutzer für diese Anlagen beim Einkauf bezahlt hat. Dabei gibt das Programm dir die Nutzer-ID, den Gesamtwert des Portfolios, den Einkaufswert, die absolute Performance (Betrag in CHF) und die relative Performance (Dezimalzahl, die der Prozentzahl der Performance entspricht) aus.

```
==== RESPONSE of GET /portfoliometrics/{id} ====
{
  "userId": 10,
  "timestamp": "2026-01-29T09:32:07.217479400",
  "totalPortfolioValue": 60268.1128,
  "investedTotalValue": 51192.0275,
  "absolutePerformance": 9076.085300000006,
  "relativePerformance": 0.17729489811670396,
  "currency": "CHF"
}
```

Abbildung 7 Mögliche Ausgabe für den Endpoint "Get Portfo Metrics by User ID"

7.4 Die Zahlen verstehen

Beispielausgabe

```
==== RESPONSE of GET /portfoliometrics/{id} ====
{
  "userId": 10,
  "timestamp": "2026-01-29T09:32:07.217479400",
  "totalPortfolioValue": 60268.1128,
  "investedTotalValue": 51192.0275,
  "absolutePerformance": 9076.085300000006,
  "relativePerformance": 0.17729489811670396,
  "currency": "CHF"
}
```

Abbildung 8 Beispielhafte Ausgabe vom Endpoint "Get Portfo Metrics by User ID"

Name des Werts	Erklärung	Gemäss Beispiel
userId	ID (eindeutige Identifikationsnummer) des Nutzers, für den die nachfolgenden Kennzahlen berechnet wurden.	Nutzer 10
timestamp	Zeitpunkt der Berechnung (Datum und Uhrzeit)	29.1.2026 um 9:32 Uhr
totalPortfolioValue	Gesamtwert des Portfolios (alle offenen Positionen dieses Nutzers) unter Berücksichtigung der aktuellsten Kursdaten, die der Datenbank vorliegen. Der Betrag ist in der unten angegeben Währung.	60268.11 CHF
investedTotalValue	Betrag, der dieser Nutzer investiert hat beim Einkauf dieser Positionen, in der unten angegeben Währung.	51192.02 CHF
absolutePerformance	Absoluter Gewinn (oder Verlust) in der unten angegeben Währung. Eine negative Zahl bedeutet einen Verlust, eine positive Zahl bedeutet einen Gewinn.	+ 9076.08 CHF
relativePerformance	Die relative Veränderung des Portfolios vom Einkaufswert zum aktuellen Wert in Prozent (als Dezimalzahl dargestellt). Eine negative Zahl bedeutet einen Verlust, eine positive Zahl bedeutet einen Gewinn.	+ 17.72%
currency	Die Währung in der die angegebenen Kennzahlen sind. Bitte beachte, dass das Anlagendashboard aktuell nur CHF als Währung unterstützt und noch keine Währungsumrechnung zu Tageskursen macht.	CHF

7.5 Mögliche Fehler

Das Programm gibt Fehlermeldungen auf Englisch aus, sollte etwas nicht funktioniert haben. Gründe für Fehler sind unter anderen:

- Ungültige Eingabe bei der ID (z.B. eine negative Ganzzahl, eine Kommazahl oder ein Wort eingegeben)
- Nutzer in der Datenbank nicht vorhanden
- Nutzer vorhanden, aber besitzt keine offenen Positionen
- Fehlende Datenbank oder fehlende config.properties-Datei (siehe Installationsanleitung).

Zudem gibt es folgende internen Fehler. Es ist allerdings sehr unwahrscheinlich, dass ein Nutzer des Anlagendashboards diese bei normaler Nutzung des Programms auslöst.

- Zu einer Anlage bestehen keine Kursdaten
- Interner Datenbankfehler
- Interne Berechnungsfehler (z.B. wenn der Einkaufswert der Anlagen 0 ist)

8 Fazit

8.1 Fazit Simon

8.1.1 Was lief gut, was lief weniger gut?

Sehr gut lief die Organisation zu Beginn des Projekts. Wir haben uns bewusst Zeit für die konzeptionelle Phase genommen und früh über Architektur, Aufbau der Applikation und Zielsetzung nachgedacht. Diese Phase war nicht nur gründlich, sondern auch realistisch. Die Ziele, die wir uns gesetzt haben, waren machbar und klar priorisiert, was sich später in der Umsetzung deutlich ausgezahlt hat. Wir konnten im geplanten Zeitrahmen arbeiten und die gesetzten Muss-Ziele ohne Zeitdruck umsetzen.

Wirklich schlecht gelaufen ist rückblickend nichts. Ein Punkt, der sich während des Projekts allerdings zäh angefühlt hat, war das Schreiben der Testfälle. Das war weniger ein inhaltliches Problem als vielmehr ein subjektives Gefühl, dass dieser Teil sehr viel Zeit in Anspruch genommen hat und sich nicht ganz so flüssig angefühlt hat wie andere Arbeiten. Am Endergebnis hat das jedoch nichts geändert: Die Tests funktionieren, sind vollständig dokumentiert und erfüllen ihren Zweck.

Positiv hervorzuheben ist auch die Zeitplanung insgesamt. Bereits am Donnerstagmorgen waren Code und Dokumentation praktisch abgeschlossen. Das hat spürbar Stress reduziert und zu einem insgesamt entspannten Projektverlauf beigetragen.

8.1.2 Zufriedenheit mit dem Endergebnis

Insgesamt bin ich mit dem Endergebnis sehr zufrieden. Wir haben unsere Ziele erreicht, diese sogar früher als geplant, und konnten dabei viel lernen. Bewusst haben wir den funktionalen Umfang eher klein gehalten. Das war eine direkte Konsequenz aus früheren Projekten, bei denen wir uns oft zu viele oder zu hohe Ziele gesetzt haben und gegen Ende unter Zeitdruck geraten sind.

Diese Entscheidung hatte zur Folge, dass das Projekt von aussen betrachtet unter Umständen keinen grossen Wow-Effekt auslöst. Die Benutzeroberfläche und der Funktionsumfang sind eher schlicht. Für jemanden, der nur das Endprodukt sieht, wirkt die Applikation möglicherweise wenig spektakulär.

Gleichzeitig liegt die Stärke dieses Projekts klar unter der Oberfläche. Der mehrschichtige Aufbau mit Controller-, Service-, DAO-, DTO- und JDBC-Ebene, das saubere Weiterreichen und Umformen der Daten sowie die klare Trennung der Verantwortlichkeiten sind technisch anspruchsvoll und praxisnah. Auch wenn es funktional nicht viele Features gibt, ist der Code qualitativ hochwertig, modular aufgebaut und gut wartbar. Genau dieses Verständnis für Struktur, Wartbarkeit und sauberes Design sehe ich als eines der grössten Ergebnisse des Projekts.

8.1.3 Was habe ich gelernt?

Fachlich konnte ich mich nochmals deutlich in Java vertiefen, insbesondere in Syntax, Fehlerbehandlung und den Umgang mit Exceptions im Unterschied zu Runtime Exceptions. Auch die Datenanbindung über JDBC und das strukturierte Weiterreichen von Daten durch verschiedene Schichten konnte ich nochmals festigen.

Sehr wertvoll war auch der konzeptionelle Teil. Wir haben konsequent mit Trello gearbeitet, was ich zwar bereits aus dem Pfadi-Kontext kannte, bisher aber kaum für eigene Projekte genutzt

habe. Als Planungs- und Management-Tool hat sich Trello klar bewährt und ist etwas, das ich für zukünftige Projekte definitiv weiterverwenden möchte.

Eine der wichtigsten Erkenntnisse war, wie stark sich eine saubere Planungsphase auszahlt. Sich bewusst Zeit zu nehmen, noch keinen Code zu schreiben und den Fokus vollständig auf Architektur, Abläufe und Zuständigkeiten zu legen, hat das Projekt später deutlich vereinfacht und stabiler gemacht.

Zudem konnte ich meine Fähigkeiten im Umgang mit Github erweitern und es war spannden zu sehen, wie sich damit zu zweit am selben Code arbeiten lässt. Gleichzeitig hat mir die Arbeit über Github auch gezeigt, dass gute Absprachen nötig sind, damit man sich nicht gegenseitig das Leben schwer mach mit Mergekonflikten und doppelt geschriebenem Code.

8.1.4 Ist alles vorhanden oder fehlt noch etwas?

Alle Muss-Ziele wurden vollständig erreicht. Der Code ist fertig, die Dokumentation abgeschlossen, Testkonzept und Testauswertung sind vorhanden und vollständig. Das Projekt kann in diesem Zustand als abgeschlossen betrachtet werden.

Da wir zeitlich gut liegen, besteht noch die Möglichkeit, einzelne Soll-Ziele umzusetzen, beispielsweise das Erfassen von Käufen und Verkäufen. Diese Erweiterungen sind jedoch optional und nicht notwendig, um das Projekt als erfolgreich zu bewerten.

8.1.5 Persönliches Fazit

Für mich bestätigt dieses Projekt sehr klar, wie sinnvoll realistische Zielsetzungen sind. Lieber etwas weniger Umfang, dafür sauber umgesetzt, gut dokumentiert und ohne Stress. Genau so entstehen die besten Lerneffekte. Nicht die möglichst komplexe Applikation bringt am meisten, sondern das Verständnis für Aufbau, Struktur und Wartbarkeit einer Software. Und genau dieses Verständnis konnte ich aus diesem Projekt mitnehmen.

8.2 Fazit Jonas

Grundlage dieses Projekts ist eine Datenbank, die Simon und ich bereits in einem vorherigen Projekt zusammen erstellt hatten. Wir hatten den Traum, eine dazu passende API zu bauen, die wir in Zukunft vielleicht noch um ein Frontend (z.B. Web-App) erweitern könnten.

Der Start des Projekts war besonders anspruchsvoll. Wir mussten uns überlegen, welche Daten ein zukünftiges Frontend brauchen würde. Dazu haben wir einige Skizzen angefertigt und die wichtigste Kernfunktionalität identifiziert: Die Darstellung von Portfoliokennzahlen.

8.2.1 Wie funktioniert eine API hinter den Kulissen?

Dieses Projekt war keine leichte Aufgabe, denn wir hatten beide noch nie eine API gebaut. Zudem standen wir vor der Herausforderung, dass die Datenbank die Einkäufe eines Nutzers mit Zeitstempel aber ohne Einkaufspreis speichert. Den Einkaufspreis müssen wir also durch den zeitlich am nächsten liegenden Kurswert für diese Anlage aus unserer Datenbank approximieren. Dafür mussten wir eine entsprechend komplexe Geschäftslogik entwickeln.

Auch wenn der Projektstart besonders anspruchsvoll war, war es für mich einer der spannendsten Teile des Projekts. Wir mussten uns selbständig und autodidaktisch die Architektur einer API erarbeiten. Dadurch haben wir gelernt, was das Schichtenmodell einer klassischen API ist, wie dieses aufgebaut ist und wozu DAOs und DTOs genutzt werden. Service und Controller kannten wir aus einer JavaFX-Übung bereits, doch in einer API funktionieren sie wiederum etwas anders.

Wir haben bewusst auf Frameworks wie Spring Boot verzichtet, da wir die API möglichst manuell und händisch bauen wollten. Dies mit dem Ziel, ein möglichst hoher Lerneffekt und möglichst tiefes Verständnis für die Abläufe zu erzielen. So parsen wir JSON manuell über die GSON-Library von Google und machen die Datenbankabfragen manuell über JDBC, wodurch wir die SQL-Statements selbst schreiben durften.

Die ersten zwei Projekttage waren von intensiver Lernarbeit geprägt. Besonders spannend war hierbei die Teamarbeit: Wir haben uns regelmässig ausgetauscht und so gegenseitig unser Verständnis für die API-Architektur verbessert. Zusammen lernen geht schneller, als wenn man lange Verständnisprozesse allein durchmacht.

Erst im Verlauf des zweiten Projekttags haben wir begonnen, Code zu schreiben. Wir haben das bewusst zuerst ausgelassen, um uns in die Konzeptarbeit zu vertiefen. Wir wollten sicherstellen, dass wir eine sehr genaue Idee und einen sehr genauen Plan haben, von was wir bauen werden, bevor wir die erste Zeile Code schreiben. Dies kommt aus Erfahrungen, die wir in früheren Projekten gemacht haben: Wer zu früh mit Coden beginnt, codet doppelt!

8.2.2 Zusammenarbeit mit Git

Ein zweiter spannender Teil des Projekts war die Zusammenarbeit mit GitHub. Wir hatten beide zuvor schon mit Git gearbeitet, aber immer allein. Nun zu zweit (und gleichzeitig!) an einem Repo zu arbeiten war für uns beide neu. Wer soll zuerst den Code pushen? Wie funktioniert das mit dem Mergen? Müssen wir nun sicherstellen, dass wir niemals an demselben File arbeiten? Und was, wenn wir es trotzdem einmal tun müssen?

Alle diese Fragen konnten wir, auch durch gemachte Fehler, beantworten und beherrschen nun die Zusammenarbeit (nach nur 5 Projekttagen) schon erstaunlich gut! Ich habe sogar gelernt, wie man interaktive Rebases macht.

8.2.3 Das Ergebnis

Mit dem Arbeitsergebnis bin ich sehr zufrieden. Die Applikation erfüllt alle Anforderungen, die wir uns gesetzt haben. Dies spricht einerseits für unsere Planung und unseren Durchhaltewillen, andererseits aber auch für unsere Konzeptarbeit. Wir haben es geschafft, einen realistischen Projektumfang festzulegen. Es ist keine leichte Aufgabe, sich einzugrenzen, wenn man von Ideen nur so sprudelt und dutzende Features umsetzen möchte.

Die API funktioniert genau so, wie wir es uns vorgestellt hatten und liefert genau die Daten, die wir im API-Vertrag festgelegt haben. Besonders stolz bin ich auf das Errorhandling: Wir fangen eine Vielzahl von Fehlern mit jeweils eigens dafür geschriebenen Fehlermeldungen ab. Dadurch weiss man bei einem Fehler immer sehr genau, was fehlgeschlagen hat. Die Fehler werden alle als JSON durch die API ausgegeben und bringen das Programm nicht zum Abstürzen. Die spannendste Herausforderung hier war das Fehlen der Konfigurationsdatei (wir hard-coden nämlich keine Datenbankzugangsdaten!) korrekt zu fangen: Die Konfigurationsdatei wurde zunächst direkt beim Start gelesen, da die ConnectionFactory als Singleton implementiert ist. Dadurch konnten wir den Fehler zunächst nicht korrekt im Controller fangen, da dieser noch gar nicht initialisiert war zum Zeitpunkt des Fehlers. Doch auch dafür haben wir eine elegante Lösung gefunden: Wir lesen und erstellen die Datenbankkonfiguration erst beim ersten Aufruf durch einen Controller, wobei wir dennoch sicherstellen, dass nur eine Verbindung gemacht wird.

8.2.4 Ausbilck

Wenn ich in die Zukunft blicke, bin ich gespannt, was wir aus unserem Anlagendashboard noch so für spannende Projekte bauen können. Eine Web-App als Frontend wäre bestimmt sehr spannend. Zudem könnten wir auch noch weitere Endpunkte bauen. Wir haben schon sehr viele Ideen für dutzende weitere spannende Auswertungen, die ein Nutzer auf seinem Anlagendashboard sehen sollte!

9 Ausblick

Wegen des engen Zeitrahmens in diesem Projekt haben wir uns bereits bei der Planung entschieden, einige Kompromisse einzugehen.

Die API hat im aktuellen Stand, wie wir sie gebaut haben, noch keine http-Fähigkeit. Zudem verwenden wir noch kein Framework (wie Spring Boot), was zu mehr Boilerplate-Code geführt hat. Wir haben uns bewusst dafür entschieden, da wir dadurch ein tieferes Verständnis für den Datenfluss innerhalb einer solchen Applikation erwerben konnten. In einem weiterführenden Projekt würde sich anbieten, http-Fähigkeit einzubauen (damit die API auch produktiv eingesetzt werden könnte), Authentifizierung und Rechte einzuführen, sowie das Projekt auf ein Framework wie Springboot umzuziehen, um weniger Boilerplate-Code schreiben zu müssen.

Für dieses Projekt haben wir uns bewusst entschieden, den Datentyp **`double`** zu verwenden für Geldbeträge. In der Praxis wird allerdings **`BigDecimal`** verwendet, um Rundungsfehlern vorzubeugen. Die Arbeit mit **`BigDecimal`** ist allerdings deutlich aufwendiger als mit **`double`**. Zum Beispiel kann bei der Vaidierung von Zahlen nicht **`if (amountB > amountC)`** geschrieben werden. Deshalb haben wir für dieses Projekt **`double`** eingesetzt. In einem weiterführenden Projekt wird es sinnvoll sein, auf **`BigDecimal`** umzustellen.

Im Rahmen dieses Projekts haben wir die unterschiedlichen Währungen bewusst ausser Acht gelassen. Es wäre im Projektrahmen nicht umsetzbar gewesen, auch noch Währungsumrechnung mit Echtzeitkursdaten zu implementieren. Deshalb reduzierten wir im Rahmen dieses Projekts die Portfoliometriken auf CHF. In einem weiterführenden Projekt wäre es eine spannende Herausforderung, Währungsumrechnung in die Applikation einzubauen.

Testkonzept

Projektname:	Anlagendashboard Verwaltungsapplikation
Projektmitglieder:	Jonas Vetsch
	Simon Leutert
Datum:	28.01.2026
Firma:	Die Schweizerische Post

Inhaltsverzeichnis

Inhaltsverzeichnis 2
Testumgebung 3
Betriebssystem 3
Java-Laufzeitumgebung 3
Entwicklungs- und Test-Tools 3
Datenbankanbindung 3
JUnit Tests 4
PortfolioMetricsService 4
FinanceMathUtil 4
User Testfälle 5
ST-01 – GUI: Gültige User-ID 5
ST-02 – GUI: Nicht existierende User-ID 6
ST-03 – GUI: Gültige User-ID mit offenen Positionen 7
ST-04 – GUI: User mit keinen offenen Positionen 8
ST-05 – GUI: Ungültige (nicht-numerische) User-ID 9
ST-06 – GUI: Leeres User-ID-Feld 10
ST-07 – Controller: Negative User-ID in getUserByIdEndpoint 11
ST-08 – Controller: Negative User-ID in getPortfolioMetricsByUserIdEndpoint 12
ST-09 – Controller: DB-Fehler bei User-Abfrage 13
ST-10 – Controller: DB-/DAO-Fehler bei PortfolioMetrics 14

Testumgebung

Die Entwicklung sowie die Durchführung der Tests der Applikation erfolgten auf einem lokalen Windows-System. Die nachfolgend beschriebene Umgebung wurde sowohl für manuelle GUI-Tests als auch für technische Tests der Applikationslogik verwendet.

Betriebssystem

Getestet wurde auf Microsoft Windows 11 Pro in der 64-Bit-Ausführung.
Verwendete Version: 10.0.26200 (Build 26200).
Das System basiert auf einer x64-Architektur und wurde über Windows PowerShell für Befehls- und Versionsabfragen genutzt.

Java-Laufzeitumgebung

Die Applikation wurde mit OpenJDK 21 aus der Temurin-Distribution entwickelt, kompiliert und ausgeführt.
Eingesetzte Versionen:

- Java Runtime Environment: OpenJDK 21.0.9 LTS
- Java Virtual Machine: OpenJDK 64-Bit Server VM (Temurin 21.0.9+10, LTS)
- Java Compiler: javac 21.0.9

Diese Java-Version wird sowohl für den Build-Prozess als auch zur Laufzeit der Applikation verwendet.

Entwicklungs- und Test-Tools

Für Entwicklung, Ausführung und Tests wurden folgende Werkzeuge eingesetzt:

- IDE: IntelliJ IDEA Ultimate 2025.3.1
  Build: IU-253.29346.138 (Build-Datum: 18.12.2025)
  Die IDE wurde für Projektverwaltung, Kompilierung sowie den Start der JavaFX-Applikation über Run-Konfigurationen verwendet.
  Automatisierte Tests (z. B. JUnit) wurden über den integrierten Test-Runner von IntelliJ IDEA ausgeführt.
- Versionsverwaltung: Git, Version 2.52.0.windows.1
- Build-Tools: Es werden keine externen Build-Tools wie Maven oder Gradle eingesetzt. Der Build erfolgt vollständig über die IDE.

Datenbankanbindung

Die Applikation verwendet eine lokal installierte relationale MySQL-Datenbank. Der MySQL-Server läuft direkt auf dem Testsystem als Windows-Dienst.

- Datenbanktyp: MySQL Community Server
- Version: 8.0.44
- Plattform: Win64 (x86_64)
- Betriebsart: lokal installiert
- Windows-Dienstname: MySQL80

Der Zugriff auf die Datenbank erfolgt ausschliesslich über die Java-Applikation mittels JDBC. Der Verbindungsaufbau wird zentral über die im Projekt implementierte ConnectionFactory realisiert.

Die notwendigen Verbindungsparameter (JDBC-URL, Benutzername und Passwort) sind in der Konfigurationsdatei config.properties hinterlegt.
Alle Lese- und Schreibzugriffe auf die Datenbank erfolgen über dedizierte DAO-Klassen (unter anderem UserJdbcDao, UserAssetJdbcDao und PriceJdbcDao), welche die Verbindung aus der ConnectionFactory beziehen. Ein separater MySQL-Client ist auf dem Testsystem nicht im Systempfad verfügbar; sämtliche Datenbankoperationen werden direkt aus der Java-Applikation heraus ausgeführt.

JUnit Tests

Für die Qualitätssicherung wurden zwei zentrale Klassen mit JUnit getestet: PortfolioMetricsService und FinanceMathUtil. Der zugehörige Testcode wird vollständig mit der Abgabe mitgeliefert.

PortfolioMetricsService

Bei dieser Klasse lag der Fokus darauf, die korrekte Berechnung der Portfolio-Kennzahlen sicherzustellen. Getestet wurden insbesondere:

- der Umgang mit ungültigen Eingaben (z. B. negative User-IDs),
- der Fall eines Benutzers ohne offene Positionen,
- sowie ein realistisches Szenario mit mehreren offenen Positionen.

Datenbankzugriffe wurden dabei bewusst durch Fake-DAO-Implementierungen ersetzt, um die Tests unabhängig von einer realen Datenbank und reproduzierbar zu halten. So wird gezielt nur die Business-Logik getestet.

FinanceMathUtil

Diese Utility-Klasse wurde isoliert getestet, da sie zentrale finanzmathematische Berechnungen enthält. Der Schwerpunkt lag auf:

- der korrekten Berechnung von absoluter und relativer Performance,
- Grenzfällen wie Startwert = 0,
- negativen Eingabewerten,
- sowie Szenarien mit Gewinn, Verlust und Null-Performance.

Zusätzlich wurde geprüft, ob bei ungültigen Parametern konsequent IllegalArgumentException geworfen wird.

Durch diese Tests wird sichergestellt, dass sowohl die Kernlogik der Portfolio-Berechnung als auch die zugrunde liegenden mathematischen Funktionen korrekt und robust implementiert sind.

User Testfälle

ST-01 – GUI: Gültige User-ID

Abschnitte	Inhalte
ID	ST-01
Anforderungen	Testart: Black-Box (GUI + Terminal), positiver Test. Die Anwendung verarbeitet eine gültige numerische User-ID, ruft UserController.getUserByIdEndpoint(int userId) auf und gibt die Userdaten als JSON im Terminal aus.
Vorbedingungen	1. Anwendung ist gebaut und startfähig (Start z. B. über Main → App). 2. JavaFX-Hauptfenster **„Anlagendashboard"** ist sichtbar. 3. Textfeld mit Prompt **„User ID"** ist leer. 4. Button **„Find User by ID"** ist sichtbar und aktiv. 5. Run-Konsole in IntelliJ ist sichtbar, um System.out.println(...) zu sehen. 6. In der Tabelle user existiert ein User mit bekannter ID, z. B. 1, und bekannten Werten (Vorname, Nachname, Email).
Ablauf	1. Anwendung starten. 2. Warten, bis das Fenster „Anlagendashboard" angezeigt wird. 3. Im Textfeld **„User ID"** die existierende ID eingeben, z. B. 1. 4. Button **„Find User by ID"** anklicken. 5. Terminalausgabe beobachten.
Erwartetes Resultat	1. In der Konsole erscheint zuerst die Zeile: ==== RESPONSE of GET /users/{id} ====. 2. Direkt danach wird genau **ein** JSON-Objekt mit den Userdaten ausgegeben, gemäss UserDto: { "userId": 1, "firstName": "...", "lastName": "...", "email": "..." } (Feldnamen gemäss Code). 3. Die Werte stimmen mit der entsprechenden Zeile in der Tabelle user überein. 4. Es wird **kein** ErrorDto-JSON ausgegeben. 5. Es erscheint **keine** Stacktrace-Ausgabe. 6. Das Textfeld „User ID" ist nach dem Klick wieder leer (es wurde userIdField.clear() aufgerufen).

ST-02 – GUI: Nicht existierende User-ID

Abschnitte	Inhalte
ID	ST-02
Anforderungen	Testart: Black-Box (GUI + Terminal), Negativtest. Die Anwendung erkennt eine numerische, aber in der Datenbank nicht vorhandene User-ID und gibt über UserController ein ErrorDto als JSON aus.
Vorbedingungen	1. Anwendung gestartet, „Anlagendashboard“ sichtbar. 2. Textfeld „User ID“ ist leer. 3. Button **„Find User by ID“** ist sichtbar und aktiv. 4. Run-Konsole sichtbar. 5. In der Tabelle user existiert **kein** Eintrag mit der Test-ID, z. B. 9999.
Ablauf	1. Im Textfeld **„User ID“** die nicht existierende ID eingeben, z. B. 9999. 2. Button **„Find User by ID“** anklicken. 3. Terminalausgabe beobachten.
Erwartetes Resultat	1. In der Konsole erscheint: ==== RESPONSE of GET /users/{id} ====. 2. Danach wird ein ErrorDto-JSON ausgegeben, z. B.: { "message": "Could not find a user with id 9999", "timestamp": "..." }. 3. Das Feld message enthält die eingegebene ID und formuliert klar, dass für diese ID kein User gefunden wurde. 4. timestamp ist nicht leer und enthält einen plausiblen ISO-8601-Zeitstempel. 5. Es wird **kein** UserDto-JSON ausgegeben. 6. Es erscheint **keine** Stacktrace-Ausgabe. 7. Das Textfeld „User ID“ ist nach dem Klick wieder geleert.

ST-03 – GUI: Gültige User-ID mit offenen Positionen

Abschnitte	Inhalte
ID	ST-03
Anforderungen	Testart: Black-Box (GUI + Terminal), positiver Test. Die Anwendung berechnet für einen User mit mindestens einer offenen Position Portfolio-Kennzahlen und gibt sie als PortfolioMetricsDto-JSON im Terminal aus (PortfolioMetricsController.getPortfolioMetricsByUserIdEndpoint(int userId)).
Vorbedingungen	1. Anwendung gestartet, Fenster „Anlagendashboard" sichtbar. 2. Textfeld „User ID" ist leer. 3. Button **„Get Portfolio Metrics by User ID"** ist sichtbar und aktiv. 4. Run-Konsole sichtbar. 5. In der Tabelle user_asset existiert für den gewählten User (z. B. User_ID = 1) **mindestens eine** offene Position (SoldAt IS NULL). 6. Für die betroffenen Asset_ID existieren in der Tabelle price passende Preise, sodass ein aktueller Wert berechnet werden kann.
Ablauf	1. Im Textfeld „User ID" die ID eines Users mit offenen Positionen eingeben, z. B. 1. 2. Button **„Get Portfolio Metrics by User ID"** anklicken. 3. Terminalausgabe beobachten.
Erwartetes Resultat	1. In der Konsole erscheint: ==== RESPONSE of GET /portfoliometrics/{id} ====. 2. Danach wird ein JSON-Objekt mit der Struktur von PortfolioMetricsDto ausgegeben: • userId (int) • timestamp (String, ISO-8601) • totalPortfolioValue (double) • investedTotalValue (double) • absolutePerformance (double) • relativePerformance (double) • currency (immer "CHF" gemäss Code). 3. userId entspricht der eingegebenen ID. 4. timestamp ist nicht leer und wirkt plausibel. 5. Alle numerischen Felder enthalten gültige Zahlenwerte (kein "NaN", keine leeren Strings). 6. Es wird **kein** ErrorDto ausgegeben. 7. Das Textfeld „User ID" ist nach dem Klick wieder geleert.

ST-04 – GUI: User mit keinen offenen Positionen

Abschnitte	Inhalte
ID	ST-04
Anforderungen	Testart: Black-Box (GUI + Terminal), Negativ-/Grenzfall. Ein existierender User hat **keine offenen Positionen**. Die Anwendung gibt **kein** PortfolioMetricsDto zurück, sondern ein ErrorDto mit einer klaren Fehlermeldung („User hat keine offenen Positionen" o. ä.).
Vorbedingungen	1. Anwendung gestartet, „Anlagendashboard" sichtbar. 2. Textfeld „User ID" ist leer. 3. Button **„Get Portfolio Metrics by User ID"** sichtbar und aktiv. 4. Run-Konsole sichtbar. 5. In der Tabelle user existiert ein User mit Test-ID, z. B. 7. 6. Für diesen User gibt es **keine offenen Positionen** in user_asset: entweder keine Zeilen mit User_ID = 7 oder alle Zeilen mit SoldAt IS NOT NULL.
Ablauf	1. Im Textfeld „User ID" die Test-ID des Users ohne offene Positionen eingeben, z. B. 7. 2. Button **„Get Portfolio Metrics by User ID"** anklicken. 3. Terminalausgabe beobachten.
Erwartetes Resultat	1. In der Konsole erscheint: ==== RESPONSE of GET /portfoliometrics/{id} ====. 2. Es wird **kein** PortfolioMetricsDto mit 0-Werten ausgegeben. 3. Stattdessen wird ein ErrorDto-JSON ausgegeben. Die message formuliert klar, dass der User mit dieser ID keine offenen Positionen hat (genauer Wortlaut abhängig von deiner Implementierung, aber inhaltlich eindeutig). 4. timestamp ist nicht leer und plausibel. 5. Es erscheint **keine** Stacktrace-Ausgabe. 6. Das Textfeld „User ID" ist nach dem Klick wieder geleert.

ST-05 – GUI: Ungültige (nicht-numerische) User-ID

Abschnitte	Inhalte
ID	ST-05
Anforderungen	Testart: Black-Box (GUI + Terminal), Negativtest. In der GUI wird eine nicht-numerische User-ID eingegeben. Beim Klick auf „Find User by ID“ wird im App-Code eine NumberFormatException abgefangen und eine saubere Fehlermeldung im Terminal ausgegeben; Controller werden **nicht** aufgerufen.
Vorbedingungen	1. Anwendung gestartet, „Anlagendashboard“ sichtbar. 2. Textfeld „User ID“ ist leer. 3. Button **„Find User by ID“** ist sichtbar und aktiv. 4. Run-Konsole sichtbar.
Ablauf	1. Im Textfeld „User ID“ einen eindeutig nicht-numerischen String eingeben, z. B. abc. 2. Button **„Find User by ID“** anklicken. 3. Terminalausgabe beobachten.
Erwartetes Resultat	1. Die GUI versucht, Integer.parseInt("abc") auszuführen und löst eine NumberFormatException aus. 2. Diese Exception wird im catch (NumberFormatException er) im App-Code abgefangen. 3. In der Konsole erscheint **genau eine** Zeile: User ID needs to be a number (integer): For input string: "abc" 4. Es erscheinen **keine** Zeilen mit ==== RESPONSE of GET /users/{id} ==== oder ==== RESPONSE of GET /portfoliometrics/{id} ====. 5. Es wird **kein** JSON ausgegeben (weder UserDto noch ErrorDto). 6. Es erscheint **keine** Stacktrace-Ausgabe. 7. Das Textfeld „User ID“ bleibt unverändert (der Text abc steht weiterhin im Feld).

ST-06 – GUI: Leeres User-ID-Feld

Abschnitte	Inhalte
ID	ST-06
Anforderungen	Testart: Black-Box (GUI + Terminal), Negativtest. Der Benutzer klickt auf einen Button, ohne eine User-ID einzugeben. Das führt zu einer NumberFormatException, die im App-Code abgefangen wird; es gibt eine Fehlermeldung, aber kein Controller-Aufruf.
Vorbedingungen	1. Anwendung gestartet, „Anlagendashboard“ sichtbar. 2. Textfeld „User ID“ ist leer (kein Text). 3. Button **„Get Portfolio Metrics by User ID“** ist sichtbar und aktiv. 4. Run-Konsole sichtbar.
Ablauf	1. Sicherstellen, dass das Textfeld „User ID“ leer ist. 2. Button **„Get Portfolio Metrics by User ID“** anklicken. 3. Terminalausgabe beobachten.
Erwartetes Resultat	1. Die GUI führt Integer.parseInt(" ") aus und löst eine NumberFormatException aus. 2. Diese wird im catch (NumberFormatException er) abgefangen. 3. In der Konsole erscheint: User ID needs to be a number (integer): For input string: " ". 4. Es erscheinen **keine** ==== RESPONSE of GET /users/{id} ==== oder ==== RESPONSE of GET /portfoliometrics/{id} ====. 5. Es wird kein JSON ausgegeben. 6. Keine Stacktrace-Ausgabe. 7. Das Textfeld bleibt leer.

ST-07 – Controller: Negative User-ID in getUserByIdEndpoint

Abschnitte	Inhalte
ID	ST-07
Anforderungen	Testart: White-Box (Controller + Service, ohne GUI), Negativtest. Es wird geprüft, dass eine **negative** User-ID im UserService eine IllegalArgumentException auslöst und UserController.getUserByIdEndpoint(int userId) dies korrekt in ein ErrorDto-JSON mit Präfix „Invalid request:“ übersetzt.
Vorbedingungen	1. Es existiert eine kleine technische Testklasse (oder entsprechender Testcode), die UserDao, UserService und UserController analog zur App initialisiert. 2. DB-Verbindung ist korrekt konfiguriert (es darf zu keinem separaten DB-Fehler kommen).
Ablauf	1. In der Testklasse UserController initialisieren. 2. Methode getUserByIdEndpoint(-1) direkt aufrufen. 3. Konsolenausgabe beobachten.
Erwartetes Resultat	1. In der Konsole erscheint: ==== RESPONSE of GET /users/{id} ====. 2. UserService.getUserById(-1) erkennt im Code (if (id > 0) … else throw new IllegalArgumentException("Invalid user ID: " + id);) die negative ID und wirft eine IllegalArgumentException mit Message Invalid user ID: -1. 3. UserController fängt diese im catch (IllegalArgumentException e)-Block ab und erzeugt ein ErrorDto mit Message: "Invalid request: " + e.getMessage() → also effektiv: Invalid request: Invalid user ID: -1. 4. Über JsonUtil.toJson(errorDto) wird ein JSON mit dieser Message und einem timestamp ausgegeben. 5. Es wird **kein** UserDto-JSON ausgegeben. 6. Es erscheint **keine** Exception-Stacktrace auf der Konsole (nur das Error-JSON).

ST-08 – Controller: Negative User-ID in getPortfolioMetricsByUserIdEndpoint

Abschnitte	Inhalte
ID	ST-08
Anforderungen	Testart: White-Box (Controller + Service, ohne GUI), Negativtest. Es wird geprüft, dass PortfolioMetricsService.getPortfolioMetricsByUserId(int userId) eine negative User-ID ablehnt und PortfolioMetricsController diese Situation korrekt in ein Error-JSON übersetzt.
Vorbedingungen	1. Testcode initialisiert PriceDao, UserAssetDao, PortfolioMetricsService und PortfolioMetricsController analog zur App (z. B. mit den echten JDBC-DAOs). 2. DB-Verbindung ist funktionsfähig (kein separater DB-Fehler).
Ablauf	1. In der Testklasse PortfolioMetricsController initialisieren. 2. Methode getPortfolioMetricsByUserIdEndpoint(-5) direkt aufrufen. 3. Konsolenausgabe beobachten.
Erwartetes Resultat	1. In der Konsole erscheint: ==== RESPONSE of GET /portfoliometrics/{id} ====. 2. PortfolioMetricsService.getPortfolioMetricsByUserId(-5) prüft if (userId < 0) throw new IllegalArgumentException("Invalid user ID: " + userId); und wirft die Exception mit Message Invalid user ID: -5. 3. PortfolioMetricsController fängt diese im catch (IllegalArgumentException e)-Block ab und erzeugt ein ErrorDto mit **genau** der Message "Invalid user ID: " + userId → also Invalid user ID: -5 (im Unterschied zum UserController wird die Exception-Message hier **nicht** nochmals mit „Invalid request:" erweitert). 4. In der Konsole wird das entsprechende Error-JSON (mit message und timestamp) ausgegeben. 5. Es wird **kein** PortfolioMetricsDto-JSON ausgegeben. 6. Es erscheint keine Stacktrace-Ausgabe.

ST-09 – Controller: DB-Fehler bei User-Abfrage

Abschnitte	Inhalte
ID	ST-09
Anforderungen	Testart: White-Box (End-to-End mit Controller, ohne GUI), Negativtest. Es wird gezielt ein **Datenbankfehler** provoziert, um den catch (RuntimeException e)-Zweig in UserController.getUserByIdEndpoint zu testen.
Vorbedingungen	1. Für diesen Test wird die DB-Konfiguration (z. B. ConnectionFactory / config.properties) so manipuliert, dass ConnectionFactory.getInstance().getConnection() scheitert (z. B. falsche DB-URL oder Port), während der Rest unverändert bleibt. 2. Testcode initialisiert UserJdbcDao, UserService und UserController mit dieser Konfiguration. 3. Es wird eine formal gültige, positive User-ID verwendet, z. B. 1.
Ablauf	1. Testklasse ausführen und userController.getUserByIdEndpoint(1); direkt aufrufen. 2. Konsolenausgabe beobachten.
Erwartetes Resultat	1. In der Konsole erscheint: ==== RESPONSE of GET /users/{id} ====. 2. UserJdbcDao.findById(1) versucht, eine DB-Verbindung zu öffnen, löst dabei eine SQLException aus und wirft als Folge eine RuntimeException mit Message: "Error from database while trying to get user data for user with ID " + id + ". " + e.getMessage(). 3. UserController fängt diese RuntimeException im catch (RuntimeException e)-Block ab, erzeugt ein ErrorDto mit message = e.getMessage() und gibt über JsonUtil.toJson(errorDto) ein Error-JSON aus. 4. Dieses JSON enthält den oben beschriebenen Text im message-Feld sowie einen timestamp. 5. Es wird **kein** UserDto-JSON ausgegeben. 6. Die eigentliche Stacktrace der Exception wird nicht in der Konsole ausgegeben (nur die JSON-Fehlermeldung).

ST-10 – Controller: DB-/DAO-Fehler bei PortfolioMetrics

Abschnitte	Inhalte
ID	ST-10
Anforderungen	Testart: White-Box (Controller + Service + DAOs, ohne GUI), Negativtest. Es wird ein Fehler in den Portfolio-Datenzugriffen provoziert (z. B. DB nicht erreichbar oder gezielt Exception aus UserAssetDao/PriceDao), um den catch (RuntimeException e)-Zweig im PortfolioMetricsController zu testen.
Vorbedingungen	Variante A – DB-Fehler: 1. DB-Konfiguration so anpassen, dass DB-Zugriffe (z. B. in UserAssetJdbcDao.getOpenPositionsByUserId oder PriceJdbcDao.getPriceBeforeTimestampByAssetID) fehlschlagen und eine RuntimeException geworfen wird („Error while loading open positions for user with ID …" bzw. „Error while querying price …"). Variante B – künstlicher DAO-Fehler: 1. Statt der echten JDBC-Implementierungen werden Test-DAOs verwendet, die an passender Stelle direkt eine RuntimeException mit einer klaren Fehlerbotschaft werfen. In beiden Varianten: 2. PortfolioMetricsService und PortfolioMetricsController sind mit diesen DAOs korrekt initialisiert. 3. Eine formal gültige, positive User-ID wird verwendet (z. B. 2).
Ablauf	1. Testklasse starten und portfolioMetricsController.getPortfolioMetricsByUserIdEndpoint(2); direkt aufrufen. 2. Konsolenausgabe beobachten.
Erwartetes Resultat	1. In der Konsole erscheint: ==== RESPONSE of GET /portfoliometrics/{id} ====. 2. Beim Zugriff auf offene Positionen oder Preise tritt eine RuntimeException auf (entweder aus den JDBC-DAOs mit einer der im Code definierten Fehlermeldungen oder aus einem Test-DAO). 3. PortfolioMetricsController fängt diese im catch (RuntimeException e)-Block ab. 4. Es wird ein ErrorDto erzeugt mit message = e.getMessage() und timestamp = ..., das über JsonUtil.toJson(errorDto) als JSON auf die Konsole geschrieben wird. 5. Es wird **kein** PortfolioMetricsDto-JSON ausgegeben. 6. Es erscheint kein Exception-Stacktrace auf der Konsole, nur das Error-JSON.

Testprotokoll

Projektname: Anlagendashboard Verwaltungsapplikation

Testperson: Simon Leutert, Marius Morf (Externer Reviewer)

Datum: 29.01.2026

Testfälle

Die definierten Testfälle entsprechen dem Dokument **„Testkonzept"** (Version vom 29.01.2026) und wurden anhand der dort beschriebenen Abläufe, Vorbedingungen und erwarteten Resultate vollständig durchgeführt.

Auswertung

Die folgende Tabelle fasst die Resultate der durchgeführten System- und Controller-Tests zusammen. Für jeden Testfall wird festgehalten, ob dieser erfolgreich war und welches beobachtete Verhalten dokumentiert wurde.

ID	Ergebnis	Testauswertung (Ist-Resultat)
ST-01	Erfolgreich	Bei Eingabe einer gültigen, existierenden User-ID wurde der Controller korrekt aufgerufen. In der Konsole erschien der Response-Header ==== RESPONSE of GET /users/{id} ==== gefolgt von genau einem UserDto-JSON mit korrekten Attributen. Es wurde kein ErrorDto und kein Stacktrace ausgegeben.
ST-02	Erfolgreich	Eine numerische, aber nicht existierende User-ID wurde korrekt erkannt. Der UserController gab ein ErrorDto-JSON mit einer verständlichen Fehlermeldung und Timestamp aus. Es wurde kein UserDto-JSON und keine Stacktrace-Ausgabe erzeugt.
ST-03	Erfolgreich	Für einen User mit offenen Positionen wurden die Portfolio-Kennzahlen korrekt berechnet. In der Konsole wurde ein vollständiges PortfolioMetricsDto-JSON mit gültigen numerischen Werten ausgegeben. Es trat kein Fehlerfall auf.
ST-04	Erfolgreich	Ein existierender User ohne offene Positionen wurde korrekt erkannt. Statt eines PortfolioMetricsDto wurde ein ErrorDto-JSON mit einer inhaltlich passenden Fehlermeldung ausgegeben. Es wurden keine 0-Werte berechnet und kein Stacktrace angezeigt.
ST-05	Erfolgreich	Eine nicht-numerische User-ID führte zu einer NumberFormatException im GUI-Code, die korrekt abgefangen wurde. Es erschien eine klare Fehlermeldung im Terminal. Controller-Methoden wurden nicht aufgerufen und es wurde kein JSON ausgegeben.
ST-06	Erfolgreich	Ein leeres User-ID-Feld führte ebenfalls zu einer NumberFormatException, die im GUI-Code korrekt behandelt wurde. Es wurde eine verständliche Fehlermeldung ausgegeben, ohne Controller-Aufruf, JSON-Ausgabe oder Stacktrace.

ST-07	Erfolgreich	Eine negative User-ID wurde im UserService erkannt und führte zu einer IllegalArgumentException. Der UserController fing diese korrekt ab und gab ein ErrorDto-JSON mit dem Präfix „Invalid request:" sowie einem Timestamp aus. Es wurde kein UserDto ausgegeben.
ST-08	Erfolgreich	Eine negative User-ID in den PortfolioMetrics wurde im PortfolioMetricsService korrekt validiert. Der Controller gab ein ErrorDto-JSON mit der Exception-Message und Timestamp aus. Es wurde kein PortfolioMetricsDto erzeugt und kein Stacktrace angezeigt.
ST-09	Erfolgreich	Ein gezielt provozierter Datenbankfehler bei der User-Abfrage führte zu einer RuntimeException im DAO. Diese wurde im UserController korrekt abgefangen und als ErrorDto-JSON mit passender Fehlermeldung und Timestamp ausgegeben. Es erschien kein Stacktrace.
ST-10	Erfolgreich	Ein DB-/DAO-Fehler bei den PortfolioMetrics wurde erfolgreich simuliert. Die RuntimeException aus dem DAO wurde im PortfolioMetricsController im catch-Block abgefangen. Es wurde ein ErrorDto-JSON mit message und timestamp ausgegeben, ohne PortfolioMetricsDto und ohne Stacktrace.

Unterschrift: ....