Entwurf

Projektbezeichnung	Trading Journal
Projektleiter	Christopher Berger
Erstellt am	07.02.2022
Letzte Änderung am	11.03.2022
Status	Fertiggestellt
Aktuelle Version	1.3

Änderungsverlauf

Datum	Version	Geänderte Kapitel	Art der Änderung	Autor	Status
07.02.2022	1.0	Alle	Erstellung	Christopher Berger	In Bearbeitung
10.02.2022	1.1	3,5	Ergänzung	Christopher Berger	In Bearbeitung
10.03.2022	1.2	1,5,6	Ergänzung	Christopher Berger	In Bearbeitung
11.03.2022	1.3	4,5,7	Ergänzung	Christopher Berger	Fertiggestellt

INHALT

1 ARCHITEKTUR

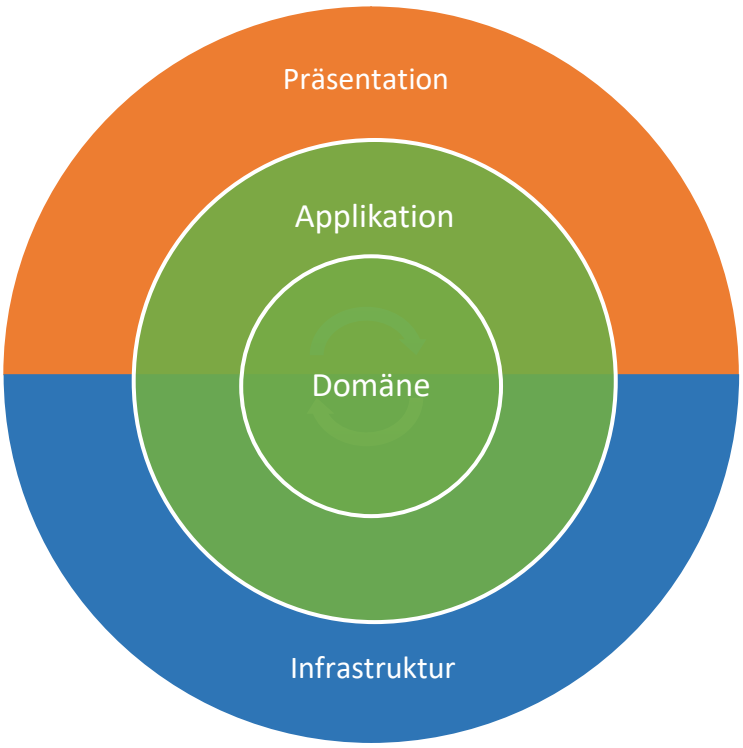
Das Programm läuft auf folgender Umgebung beim Kunden:

- IIS Web Server
- Microsoft SQL-Server

Die Implementierung der Applikation erfolgt auf Basis der **Clean Architecture** Vorlage von Jason Taylor.

1.1 KERNPUNKTE

- Die **Domäne** enthält unternehmensweite Logik und Typen
- Die **Anwendung** enthält Geschäftslogik und -typen, sowie Interfaces der Typen aus der Infrastrukturschicht
- **Infrastruktur** beinhaltet alle externen Belange
- **Präsentation** und **Infrastruktur** hängen nur von der Anwendungsschicht ab
- **Infrastruktur**- und **Präsentations**komponenten können mit minimalem Aufwand ausgetauscht werden



Weitere Details können der Quelle entnommen werden.

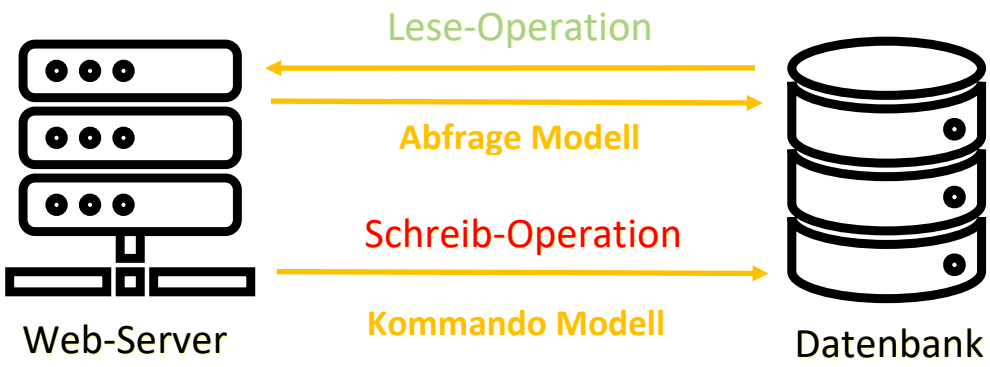
Quelle: https://github.com/jasontaylordev/CleanArchitecture

1.2 Design Muster

Zusätzlich zur Vorlage der Struktur der Applikation werden untenstehende Design Muster angewendet, um mit den Daten aus der Domäne zu interagieren.

1.2.1 CQS

CQS steht für **Command and Query Segregation**, ein Muster, das Lese- und Aktualisierungsvorgänge trennt. Die Implementierung von CQS in kann deren Leistung, Skalierbarkeit und Sicherheit maximieren. Die durch die Migration zu CQS geschaffene Flexibilität ermöglicht es einem System, sich im Laufe der Zeit besser zu entwickeln, und verhindert, dass Aktualisierungsbefehle Zusammenführungskonflikte auf Domänenebene verursachen.



Mehr Details unter: https://docs.microsoft.com/en-us/azure/architecture/patterns/cqrs

1.2.2 Mediator

Beim Mediator-Muster wird die Kommunikation zwischen Objekten in einem Mediator-Objekt eingekapselt. Objekte kommunizieren nicht mehr direkt miteinander, sondern kommunizieren über den Mediator. Dies reduziert die Abhängigkeiten zwischen kommunizierenden Objekten, wodurch die Kopplung reduziert wird.

Mehr Details unter: https://en.wikipedia.org/wiki/Mediator_pattern

1.2.3 Validation (Fluent Validation)

Für die Validierung der Operationen wird Fluent Validation verwendet. Hierbei handelt es sich um eine Validierungsbibliothek für . NET, das zum Erstellen stark typisierter Validierungsregeln für Geschäftsobjekte verwendet wird. Fluent-Validierung ist eine Möglichkeit, dedizierte Validator-Objekte einzurichten, die Sie verwenden würden, wenn Sie die Validierungslogik getrennt von der Geschäftslogik behandeln möchten.

2 ENTWICKLUNGSUMGEBUNG

Das Programm wird mit Hilfe der folgenden Entwicklungsumgebung entwickelt:

Name	Type	Version
Microsoft Visual Studio Community 2022 (64-bit)	IDE	17.0.2
ASP.NET Core	RTE	6.0.2
Blazor WASM Hosted (Client/Server)	Projekt Template	-
Microsoft.EntityFrameworkCore	Bibliothek	6.0.1
Microsoft.EntityFrameworkCore.Tools	Bibliothek	6.0.1
Microsoft.EntityFrameworkCore.SqlServer	Bibliothek	6.0.1
Microsoft.AspNetCore.Components.WebAssembly	Bibliothek	6.0.1
Microsoft.AspNetCore.Authentication.JwtBearer	Bibliothek	6.0.1
Microsoft.AspNetCore.Components.Authorization	Bibliothek	6.0.1
System.IdentityModel.Tokens.Jwt	Bibliothek	6.15.1
FluentValidation.AspNetCore	Bibliothek	10.3.6
FluentValidation.DependencyInjectionExtension	Bibliothek	10.3.6
MediatR.Extensions.Microsoft.DependencyInjection	Bibliothek	10.0.1
Blazored.LocalStorage	Bibliothek	4.1.5
MudBlazor	Bibliothek	6.0.6

3 EXTERNE SYSTEME

3.1 HANDELSPLATTFORM

Die userbezogenen Handelsdaten (Trades, Executions) werden über die API der Handelsplattform ByBit importiert.

API Dok.: https://bybit-exchange.github.io/docs

Web Socket Dok.: https://bybit-exchange.github.io/docs/inverse/#t-websocket

Für jeden Account, welchen die User hinterlegt haben, wird eine Websocket Verbindung mit der Plattform erstellt.

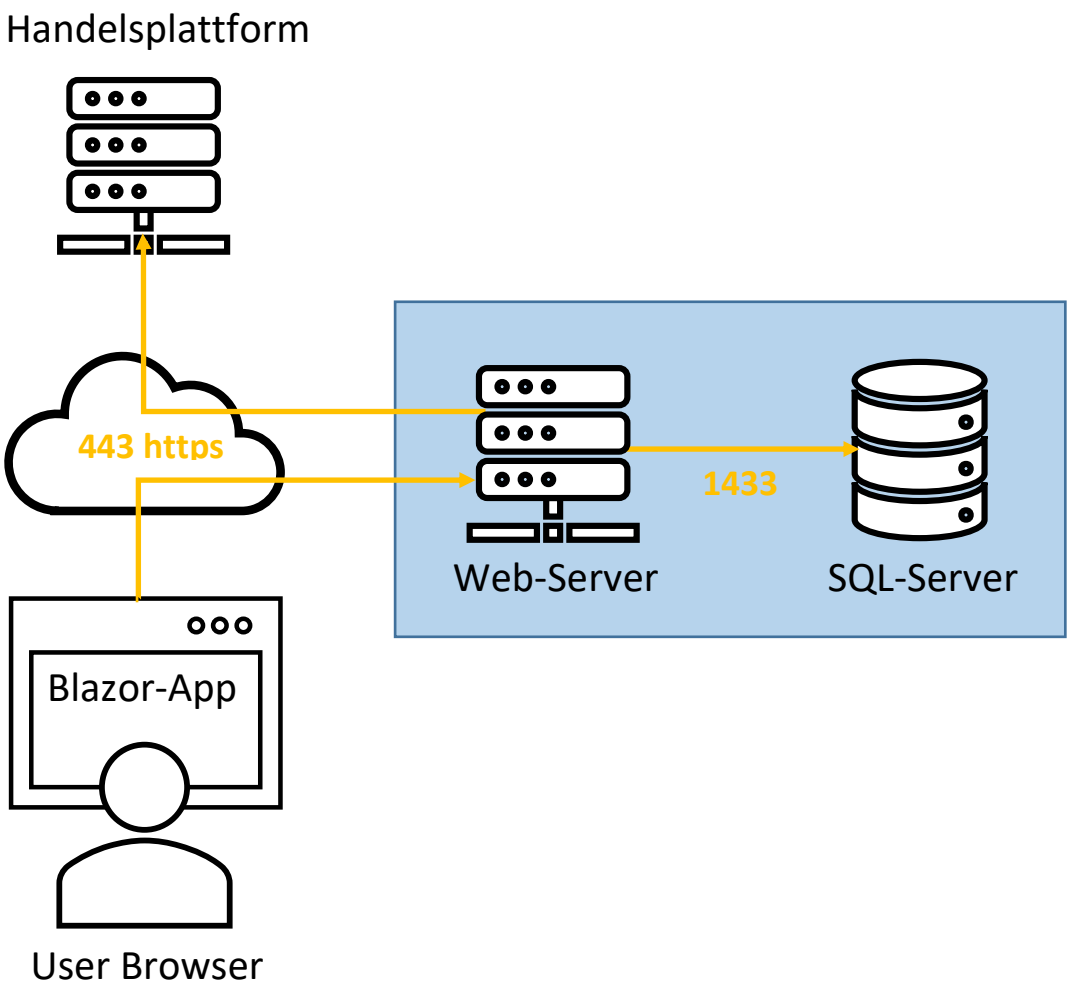
Sobald eine Message im System eingeht, in der über die Ausführung eines Trades berichtet wird. Werden genauere Infos passend zum Vertrag (z.B. ETHUSDT) über die Rest API geladen. Aus dieser Abfrage werden Executions generiert und der dazugehörige Trade wird aktualisiert oder erstellt.

4 NETZWERK DIAGRAMM

Die **Blazor-App**, ihre Abhängigkeiten und die .NET-Laufzeit werden parallel beim ersten Aufruf der Webseite in den **User Browser** geladen. Die App wird direkt im Browser-UI-Thread ausgeführt.

Die **Blazor-App** wird von einer ASP.NET Core-App (**Web-Server**) bereitgestellt welche auch als Backend-Server dient und direkt auf die Datenbank zugreift (**SQL-Server**).

Zusätzlich ruft der Web-Server userbezogene Daten von der **API** der **Handelsplattform** ab.



5 KLASSEN

Das folgende Diagramm stellt die Klassen, Attribute, Methoden und Assoziationen der Applikation dar.

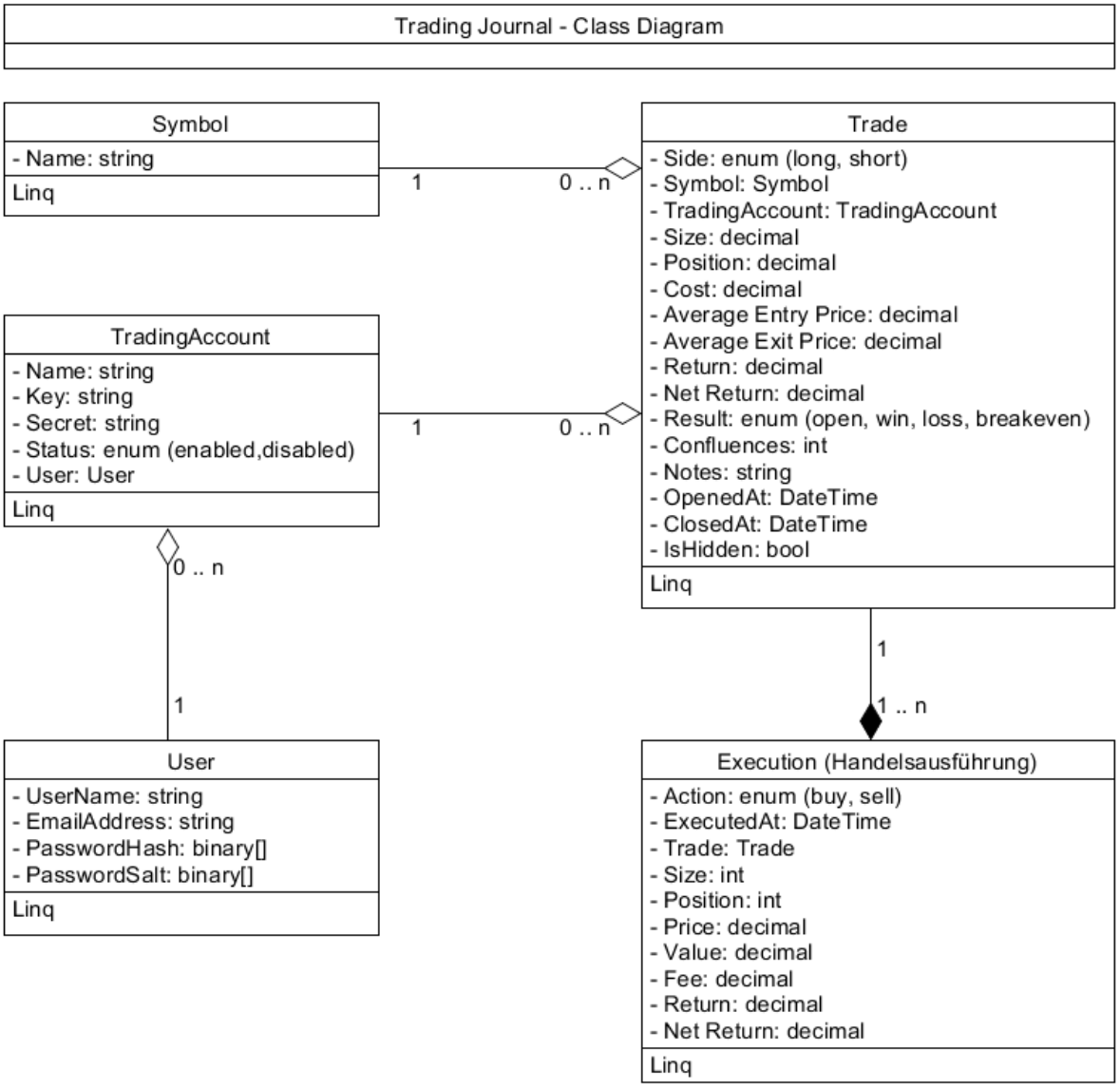
Die Klassen befinden sich in folgendem Namespace innerhalb der Domain Schicht:

- TradingJournal.Domain.Entities

Der Zugriff und Erstellung der Datenbank findet über EntityFrameworkCore statt. Zusätzlich wird die Konfiguration des Datenbankmodells über Konfigurationsdateien gesteuert, diese befinden sich in der Infrastrukturschicht unter folgendem Namespace:

- TradingJournal.Infrastructure.Server.Persistence.Configurations

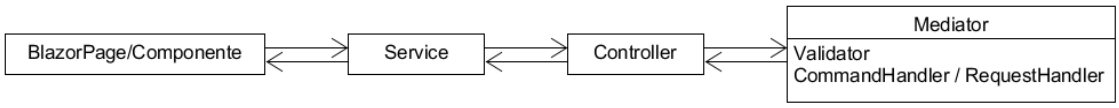
5.1 UML-KLASSENDIAGRAMM (DOMÄNEN-SCHICHT)



5.2 UML-Klassendiagramm (Applikations-Schicht)

Interaktionen mit den Klassen der Domäne findet über den Mediator statt. Für Aufgaben und Abfragen werden eigene Klassen anhand des CQS Design Musters implementiert. Dies muss für jede Klasse der Domäne erfolgen.

Aufgrund der hohen Anzahl an Applikations-Klassen gehen wir in den Beispielen unten, nur auf die einzelnen Typen zur Interaktion mit der Tabelle **TradingAccount** genauer ein. Das Schema ist jedoch für alle Klassen ident. Die Interaktion erfolgt immer ähnlich der Abbildung.

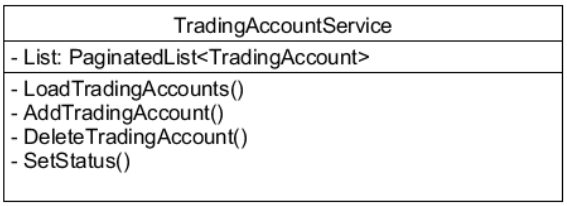


5.2.1 Services

Der Client implementiert Services welche über Dependency Injection (Einbringen von Abhängigkeiten) auf den Blazor Komponenten injiziert und verwendet werden.

Die Services laufen im Browser des Klienten und senden Get, Post und Put Anfragen an die verschiedenen Controller am Server.

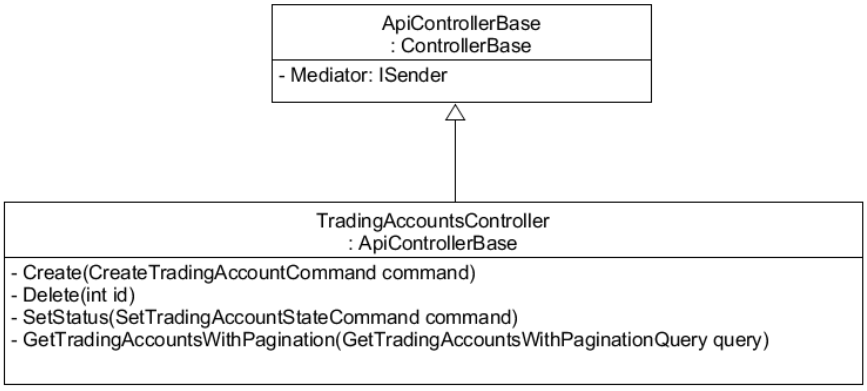
Rückgabedaten werden vom Service deserialisiert und an die aufrufende Blazor Komponente übergeben.



5.2.2 Controller

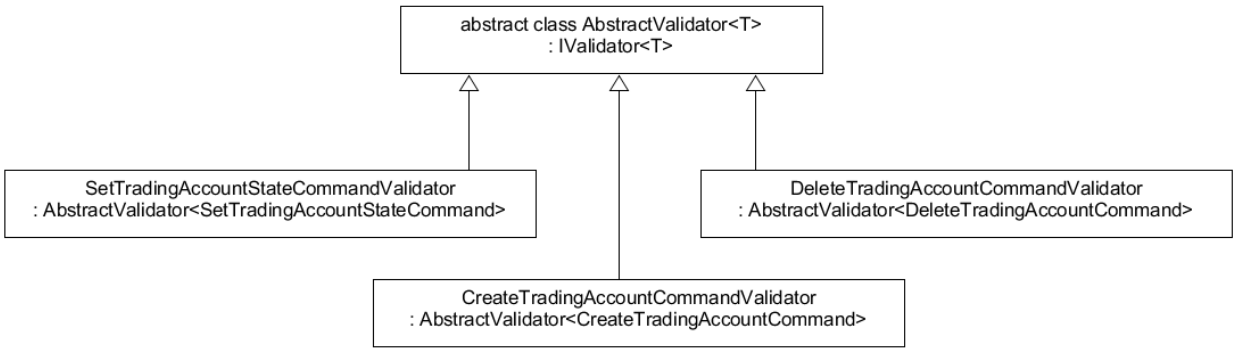
Der Server implementiert Controller welche keinerlei Unternehmens-Logik beinhalten. Die Controller geben die Anfragen lediglich an den Mediator weiter. Dies geschieht mittels der Typen der Aufgaben und Anfragen, welche unter Punkt 5.2.1 und 5.2.2 näher erläutert sind.

Sollte die Aufgabe oder Anfrage einen Rückgabewert haben, wird dieser direkt vom Aufruf des Mediators zurückgegeben. Die Implementierung eines Controllers beläuft sich meistens auf 1 bis 2 Zeilen.



5.2.3 Validation (Fluent Validation)

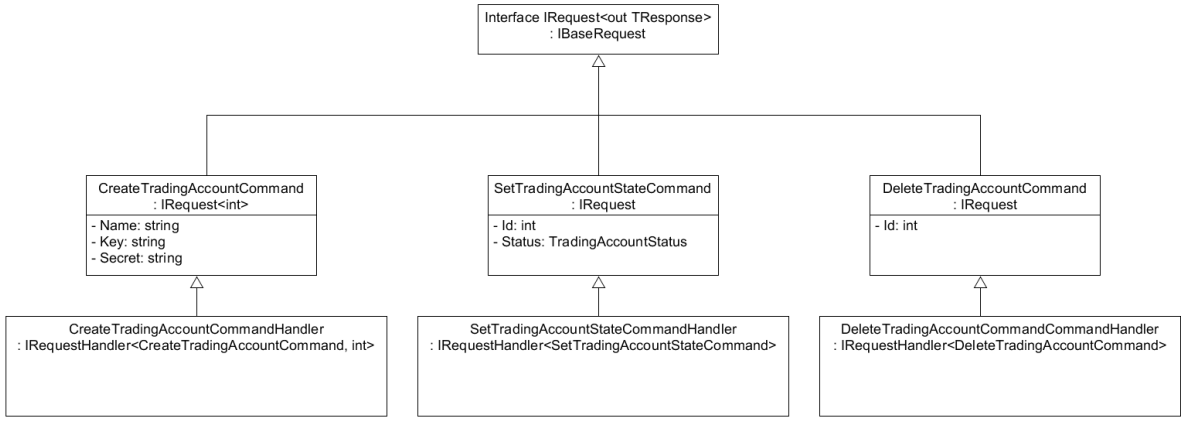
Optional kann pro Operation ein sogenannter AbstractValidator implementiert werden. Dieser wird als erstes vom Mediator ausgeführt und enthält die Validierungslogik.



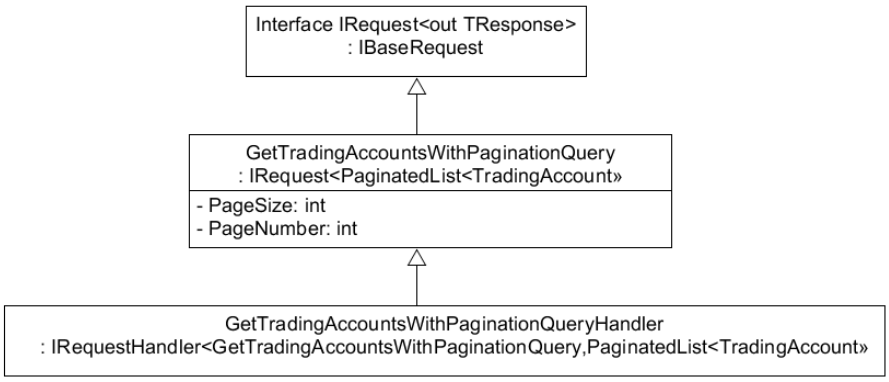
5.2.4 Mediator - CQS

Anschließend wird die Geschäftslogik ausgeführt. Hierfür werden jeweils 2 Klassen pro Operation implementiert. Die erste implementiert das Interface IRequest und enthält lediglich die Übergabewerte in Form von Properties. Für die Operation wird zusätzlich eine Klasse mit dem Interface IRequestHandler erstellt. Diese enthält die Geschäftslogik und greift direkt auf die Datenbank zu.

5.2.4.1 Commands (Aufgaben)



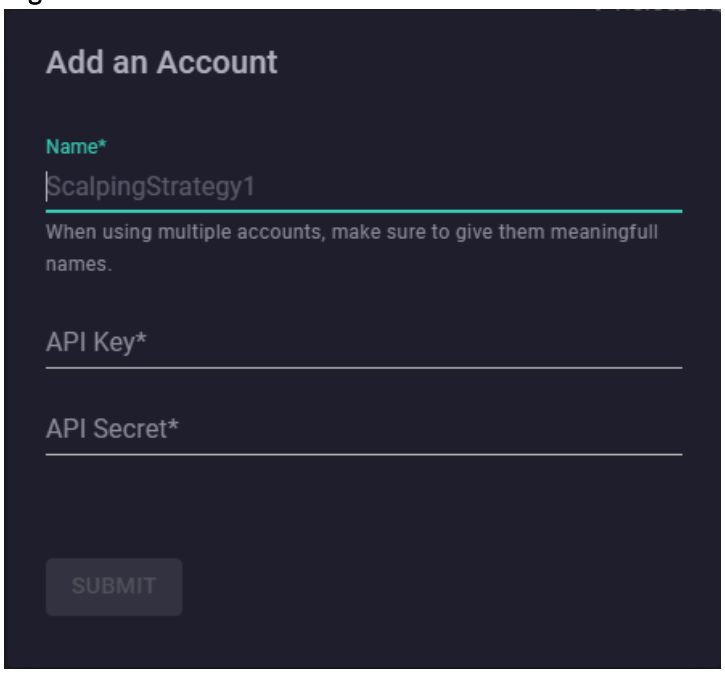
5.2.4.2 Queries (Abfragen)



5.3 FUNKTIONSBESCHREIBUNG BEISPIEL – ADDTRADINGACCOUNT

1. Der User öffnet die Seite /accounts und klickt auf die **Add** Schaltfläche
2. Er befüllt die folgenden Felder und klickt anschließend auf **Submit**:
    a. Name
    b. API-Key
    c. API-Secret
3. Diese ist mit der Funktion AddTradingAccount des TradingAccountServices verknüpft. Der Service sendet die Daten in Form der **CreateTradingAccountCommand** Klasse via JSON an den Controller **TradingAccountsController** und des Methode **Create**.
4. Der Controller gibt den Request an den injizierten Mediator weiter.
5. Für das Kommando wurde ein Validator implementiert, bevor der Ausführung des CommandHandlers werden alle Validierungsschritte durchgeführt, sollte einer der Schritte fehlschlagen wird an dieser Stelle abgebrochen und eine Fehlermeldung an den Controller übergeben. Folgende Tests werden durchgeführt:
    a. Name ist einzigartig unter allen Accounts desselben Users
    b. Der Account verfügt lediglich über Lese-Rechte
    c. Der Account wird erfolgreich authentifiziert
6. Der CommandHandler welcher die Unternehmenslogik enthält erstellt den TradingAccount anhand der übergebenen Daten und setzt als Besitzer den User aus dem Http-Kontext.
7. Rückgabewert ist die **Id** welche der TradingAccount erhält, sobald er in der Datenbank gespeichert wurde.

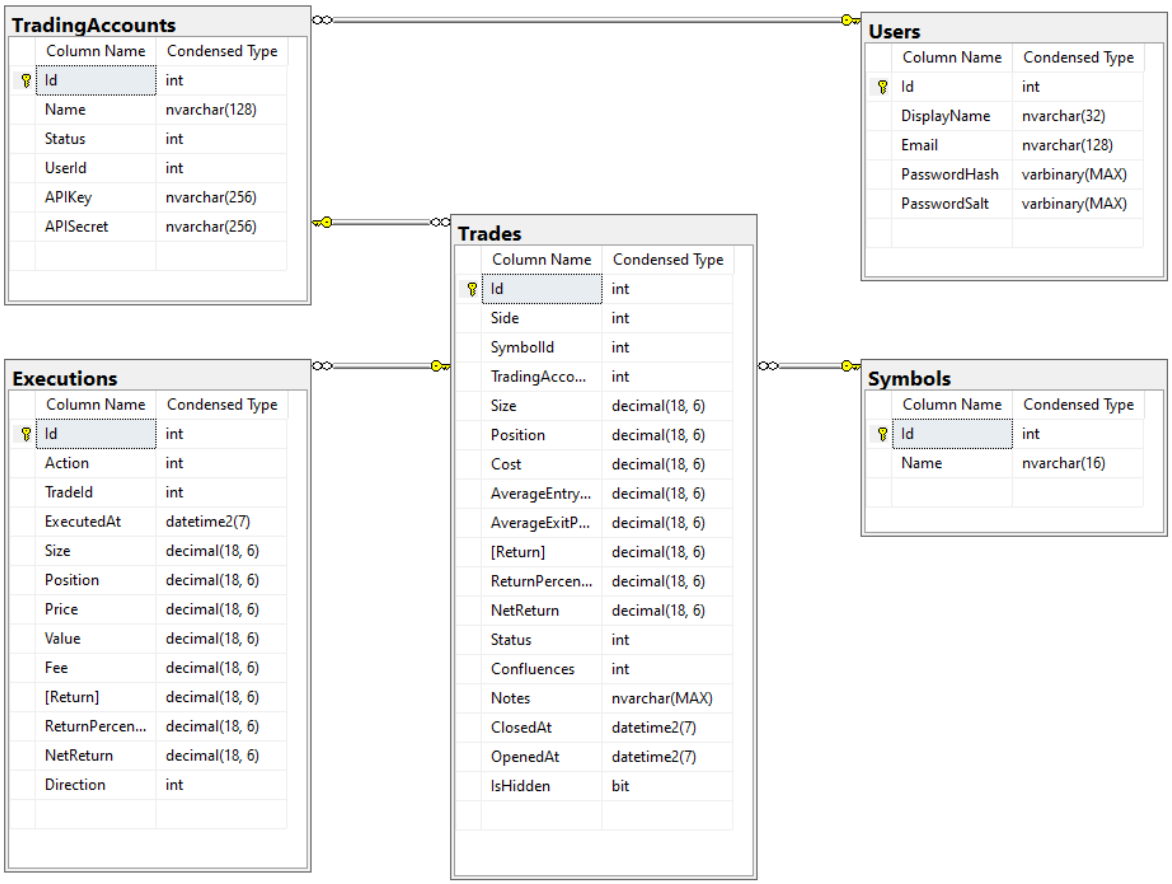
5.3.1 Aufbau der Eingabemaske



6 DATENHALTUNG

6.1 DATENBANK

Das folgende Diagramm stellt das Datenbanklayout dar (Datenbank SQL-Server):



7 KONFIGURATIONSDATEI

Die folgenden Parameter lassen sich über die appsettings.json Datei anpassen.

Parameter	Funktion	Typ
ConnectionStrings:Default	Enthält die Verbindungszeichenfolge zur Datenbank	String
SeedDatabaseWithSampleData	Falls „true“ wird die Datenbank mit Demodaten befüllt (User, TradingAccounts, Trades, Executions)	Boolean
JavaWebTokenSettings:EncryptionKey	Der Schlüssel der für die Verschlüsselung der Java Web Token verwendet wird	String
JavaWebTokenSettings:DaysToExpire	Legt die Gültigkeitsdauer der im Browser gecacheten Tokens fest (je kürzer desto sicherer)	Integer