Design Document

|  |  |
| --- | --- |
| **Project name** | Trading Journal |
| **Project manager** | Christopher Berger |
| **Created on** | 07.02.2022 |
| **Last modified on** | 11.03.2022 |
| **Status** | Finished |
| **Current version** | 1.3 |

Change History

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Date** | **Version** | **Amended chapters** | **Type of change** | **Author** | **Status** |
| 07.02.2022 | 1.0 | All | Creation | Christopher Berger | In Progress |
| 10.02.2022 | 1.1 | 3,5 | Addition | Christopher Berger | In Progress |
| 10.03.2022 | 1.2 | 1,5,6 | Addition | Christopher Berger | In Progress |
| 11.03.2022 | 1.3 | 4,5,7 | Addition | Christopher Berger | Finished |
|  |  |  |  |  |  |
|  |  |  |  |  |  |

Content

[1 Architecture 3](#_Toc100297014)

[1.1 Application Layers 3](#_Toc100297015)

[1.2 Design Patterns 3](#_Toc100297016)

[1.2.1 CQS 4](#_Toc100297017)

[1.2.2 Mediator 4](#_Toc100297018)

[1.2.3 Validation (Fluent Validation) 4](#_Toc100297019)

[2 Development 5](#_Toc100297020)

[3 External systems 5](#_Toc100297021)

[3.1 Trading 5](#_Toc100297022)

[4 Network Diagram 6](#_Toc100297023)

[5 Classes 7](#_Toc100297024)

[5.1 UML Class Diagram (Domain Layer) 7](#_Toc100297025)

[5.2 UML Class Diagram (Application Layer) 8](#_Toc100297026)

[5.2.1 Services 8](#_Toc100297027)

[5.2.2 Controller 8](#_Toc100297028)

[5.2.3 Validation (Fluent Validation) 9](#_Toc100297029)

[5.2.4 Mediator - CQS 9](#_Toc100297030)

[5.3 Feature Description Example - AddTradingAccount 10](#_Toc100297031)

[5.3.1 Structure of the input mask 10](#_Toc100297032)

[6 Data 11](#_Toc100297033)

[6.1 Database 11](#_Toc100297034)

[7 Configuration file 11](#_Toc100297035)

# Architecture

The program runs on the following environment at the customer's site:

* IIS Web Server
* Microsoft SQL-Server

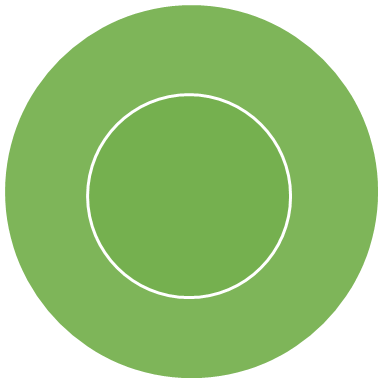
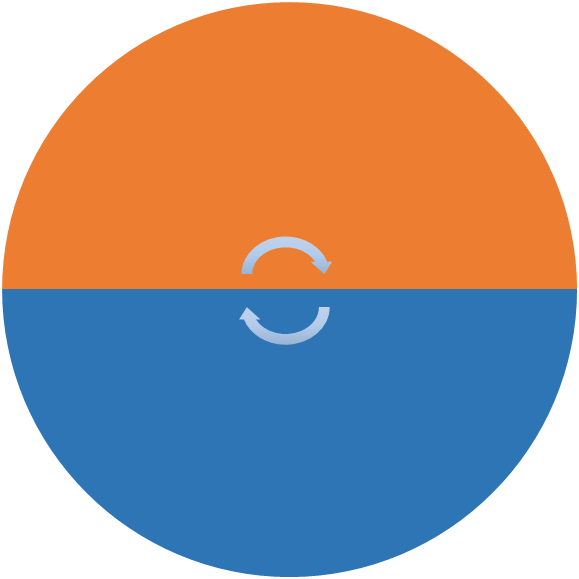
The application is implemented based on the **Clean Architecture** template by Jason Taylor.

## Application Layers

* The **domain** contains enterprise-wide logic and types
* The **application** contains business logic and types, as well as interfaces of the types from the infrastructure layer
* **Infrastructure** includes all external concerns
* **Presentation** and **infrastructure** depend only on the application layer
* **Infrastructure** and **presentation components** can be replaced with minimal effort

Domain

Application



Presentation

Infrastructure

Further details can be found in the source.

Source: <https://github.com/jasontaylordev/CleanArchitecture>

## Design Patterns

In addition to templateing the structure of the application, the design patterns below are applied to interact with the data from the domain.

### CQS

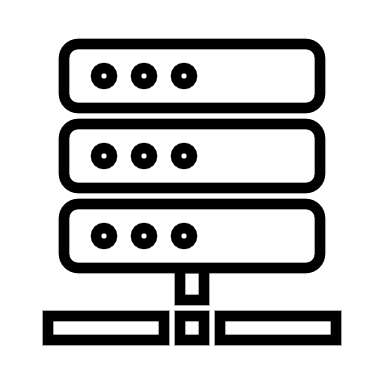
CQS stands for **Command and Query Segregation**, a pattern that separates read and update operations. Implementing CQS in can maximize their performance, scalability, and security. The flexibility created by migrating to CQS allows a system to evolve better over time and prevents update commands from causing domain-level merge conflicts.

Database

Web-Server

Read operation

**Query Model**



**Command Model**

Write operation

More details at: <https://docs.microsoft.com/en-us/azure/architecture/patterns/cqrs>

### Mediator

The Mediator pattern encapsulates communication between objects in a Mediator object. Objects no longer communicate directly with each other but communicate via the mediator. This reduces the dependencies between communicating objects, thereby reducing coupling.

More details at: <https://en.wikipedia.org/wiki/Mediator_pattern>

### Validation (Fluent Validation)

Fluent Validation is used to validate the operations . This is a validation library for the . NET, which is used to create strongly typed validation rules for business objects. Fluent validation is a way to set up dedicated Validator objects that you would use if you wanted to handle validation logic separately from business logic.

# Development

The program is developed using the following development environment:

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

# External systems

## Trading

The user-related trading data (trades, executions) are imported via the API of the ByBit trading platform.

Rest API: <https://bybit-exchange.github.io/docs/inverse/#t-authentication>

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

For each account that the users have stored, a websocket connection with the platform is created.

As soon as a message is received in the system reporting on the execution of a trade. More detailed information matching the contract (e.B. ETHUSDT) is loaded via the Rest API. Executions are generated from this query and the corresponding trade is updated or created.

# Network Diagram

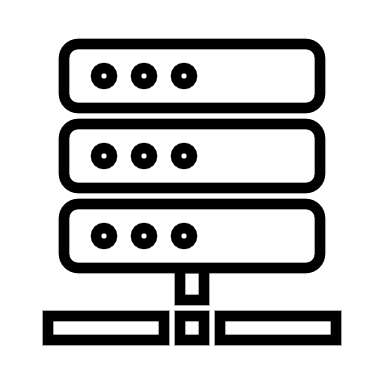
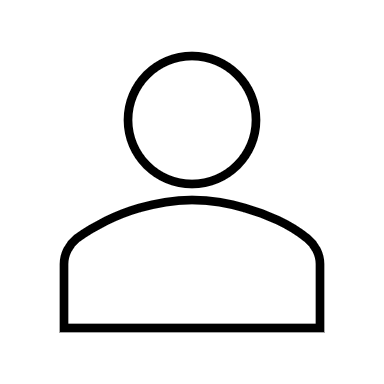
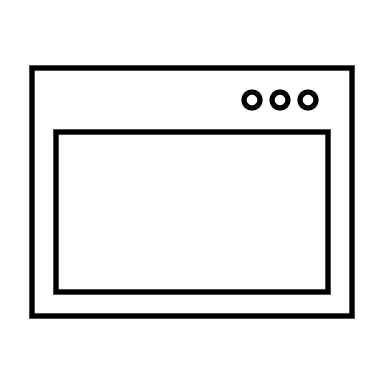
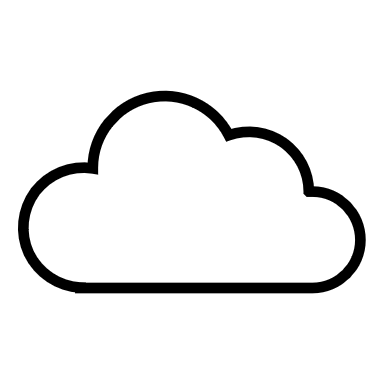
The **Blazor app**, its dependencies, and the . NET runtime are loaded into the user browser in parallel when the web page is first called. The app runs directly in the browser UI thread.

The **Blazor app** is provided by a ASP.NET core app (**web server**) which also serves as a backend server and accesses the database directly (**SQL server**).

In addition, the web server retrieves user-related data from the **API** of the **trading platform** .

SQL-Server

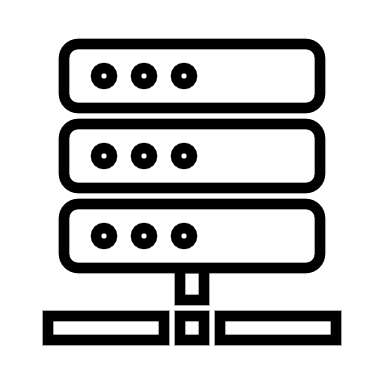
Web-Server



Trading Platform

User Browser

Blazor-App Client



**443 https**

**1433**

# Classes

The following diagram shows the classes, attributes, methods, and associations of the application.

The classes are located in the following namespace within the domain layer:

* TradingJournal.Domain.Entities

The database is accessed and created through EntityFrameworkCore. In addition, the configuration of the database model is controlled by configuration files, which are located in the infrastructure layer under the following namespace:

* TradingJournal.Infrastructure.Server.Persistence.Configurations

## UML Class Diagram (Domain Layer)

Diagram

Description automatically generated

## UML Class Diagram (Application Layer)

Interactions with the classes of the domain take place through the mediator. For tasks and queries, separate classes are implemented using the CQS design pattern. This must be done for each class of the domain.

Due to the high number of application knobs, we will only go into more detail about the individual types for interaction with the **TradingAccount** table in the examples below. However, the schema is the same for all classes. The interaction is always similar to the figure.

Diagram

Description automatically generated

### Services

The client implements services that are injected and used on the Blazor components via dependency injection.

The services run in the client's browser and send get, post and put requests to the various controllers on the server.

Return data is [deserialized by the](https://www.dict.cc/?s=deserialisieren) service and passed to the calling Blazor component.

Table

Description automatically generated

### Controller

The server implements controllers that do not contain any corporate logic. The controllers only pass on the requests to the mediator. This is done by means of the types of tasks and requests, which are explained in more detail in points 5.2.1 and 5.2.2.

If the task or request has a return value , it is returned directly by the mediator's call. The implementation of a controller usually amounts to 1 to 2 lines.

Diagram

Description automatically generated

### Validation (Fluent Validation)

Optionally, a so-called AbstractValidator can be implemented per operation. This is first executed by the mediator and contains the validation logic.

Diagram

Description automatically generated

### Mediator - CQS

The business logic is then executed. For this purpose, 2 classes per operation are implemented. The first implements the interface IRequest and contains only the transfer values in the form of properties. For the operation, a class is also created with the interface IRequestHandler. This contains the business logic and accesses the database directly.

#### Commands Tasks

Text

Description automatically generated with low confidence

#### Queries

Diagram

Description automatically generated

## Feature Description Example - AddTradingAccount

1. The user opens the /accounts page and clicks the **Add** button
2. He fills in the following fields and then clicks **Submit**:
   1. Name
   2. API-Key
   3. API-Secret
3. This is linked to the AddTradingAccount function of the TradingAccountService. The service sends the data in the form of the **CreateTradingAccountCommand** class via JSON to the **TradingAccountsController** controller and the **Create** method.
4. The controller passes the request to the injected mediator.
5. A validator has been implemented for the command, before the Execution of the CommandHandler all validation steps are performed, should one of the steps fail, it will be aborted at this point and an error message will be passed to the controller. The following tests are performed:
   1. Name is unique among all accounts of the same user
   2. The account only has read rights
   3. The account is successfully authenticated
6. The CommandHandler which contains the company logic creates the TradingAccount based on the transferred data and sets the user from the Http context as the owner.
7. Return value is the **ID** that the TradingAccount receives once it has been stored in the database.

### Structure of the input mask

Graphical user interface, application

Description automatically generated

# Data

## Database

The following diagram shows the database layout (DATABASE SQL Server):

Graphical user interface, diagram

Description automatically generated

# Configuration file

The following parameters can be adjusted via the **appsettings.json** file.

|  |  |  |
| --- | --- | --- |
| Parameter | Function | Type |
| ConnectionStrings:Default | Contains the connection string to the database. | String |
| SeedDatabaseWithSampleData | If "true" the database is seeded with demo data (Users, TradingAccounts, Trades, Executions) | Boolean |
| JavaWebTokenSettings:EncryptionKey | The key used to encrypt the Java Web Tokens | String |
| JavaWebTokenSettings:DaysToExpire | Sets the validity period of tokens cached in the browser (the shorter the more secure) | Integer |