



OSLO METROPOLITAN UNIVERSITY
STORBYUNIVERSITETET

ITP3200 - WEB APPLIKASJON

Aksje Handle Webapplikasjoner

Authors:

Awet Teklemariam, Bastien, Bernt Olsen, Okbamicheal Elias

November, 2022

Table of Contents

1	Prosjekt Beskrivelse	1
1.1	Produkt Mål	1
1.2	Brukte Teknologier/ressurser	1
1.3	Applikasjonsarkitektur	1
1.3.1	Filstruktur	1
1.3.2	Data fra Alpha Vantage API	2
1.3.3	Beskrivelse av EcbCurrencyInterface	3
1.3.4	Databasemodellen	3
1.3.5	Migration og seeding av databasen	5
1.3.6	Hosted service for rydding av databasen	5
2	Beskrivelse av applikasjonens funksjonalitet	5
2.1	Funksjonalitet knyttet til watchlist/favorittliste	5
2.2	Kjøp av Aksje	7
2.2.1	Kjøp av aksjer på klientsiden	7
2.2.2	Kjøp operasjonen på tjenersiden	8
2.3	Selge Aksje	10
2.3.1	Salg på klientsiden	10
2.3.2	Salg på tjenersiden	10
2.4	Søking	12
2.4.1	Søking på klient siden	12
2.4.2	Søking på tjener siden	13
2.5	Transaksjoner og bruker instillinger	15
2.5.1	Transaksjon Side	15
2.5.2	Bruker instillinger side	15

1 Prosjekt Beskrivelse

1.1 Produkt Mål

I grunlaget kommer funksjonelitetens krav til gjeldende produkt fra oppgavebeskrivelse av prosjektet. De sentrale funksjonelle kravene for produktet inkluderer:

- Lage en web applikasjon som skal tilby kjøp og salg av aksjer for brukere.
- Applikasjonen skal ha et frontend brukergrensesnitt med interaktive form i form av en web-side.
- Brukeren skal kunne bruke applikasjonen uten å logge inn.
- Den må støtte CRUD funksjonalitet.

1.2 Brukte Teknologier/ressurser

- Programmeringsspråk
 - i. C#
 - ii. Javascript
 - iii. HTML
 - iv. CSS
- Rammeverk og biblioteker
 - i. ASP.NET Core Web API 6.0
 - ii. Entity framework core
 - iii. JQuery 1.11.3
 - iv. Bootstrap 3
- IDE
 - i. Visual Studio Enterprise 2022
- Andre verktøy
 - 1. Database : SQLite, sqlite browser
 - 2. Versjonskontrol : GitHub, git

1.3 Applikasjonsarkitektur

1.3.1 Filstruktur

Applikasjonen består av klient side og tjener side. Hoved fokus av prosjektet frem til nå er tjener siden. Klient siden er en midlertidig løsning for å vise funksjonaliteten tjeneren tilbyr. Som figurene 1,2 og 3 viser er filene/klassene strukturert i en lagdeling. I tillegg til det har vi også en dependency injection og DAL(data layer access). I denne delen skal vi forklare kort hva hver mappe inneholder og hva som er hensikten bak inndelingen. Merk at det er Webapplikasjoner_oblig mappen som inneholder .net core web api prosjektet som skal kjøres.

I model mappen finner vi alle entity objektene som applikasjonen skal bruke til å mappe de tilsvarende tabellene i entity framework. Klassene benyttes for å representere dataen fra databasen som skal sendes til klienten.

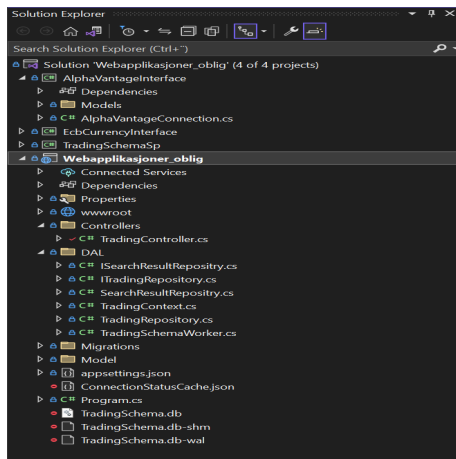


Figure 1: Fil struktur

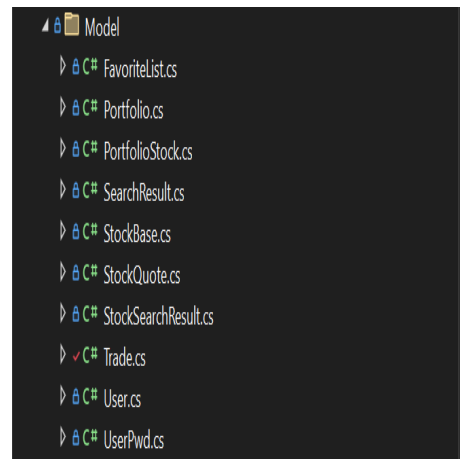


Figure 2: Model mappen

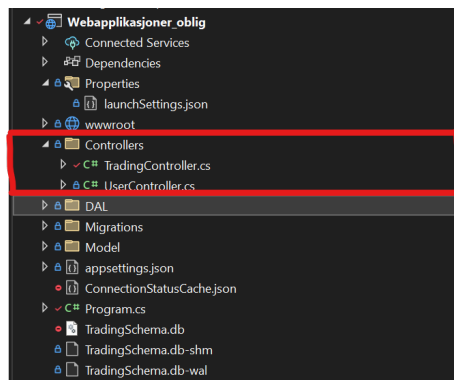


Figure 3: Kontroller mappen

Kontroller mappen inneholder klasser som håndterer request og response mellom klient siden og tjener siden (endepunktene for REST apiet). Her skal request fra klient siden håndteres ved enten hente data fra databasen eller gjøre et kall til API dependency. Det finnes en kontroller klasse: trading kontrolleren, som tar seg av kjøping, salg, favoritte liste og søking av aksjer.

repository klassene som håndterer operasjoner mot databasen. I tillegg til dette finnes Trading context klassen i denne mappen. Denne klassen initialiserer databasen med alle tabeller.

wwwroot mappen er der alle klient side kode er. Her har vi html, css og javascript koder som er brukt for å lage et enkelt midlertidig grensesnitt for applikasjonen.

1.3.2 Data fra Alpha Vantage API

Applikasjonen benytter data fra finansmarkedet hentet fra Alpha Vantage APIet. Dette er et RESTful API som i utgangspunktet er gratis, med en begrensning i antall kall mot apiet med samme api-nøkkel. Det er ikke mulig å gjøre mer en 500 http-forespørsler mot tjenesten per dag og 5 forespørsler per minutt. I applikasjonen har vi benyttet endepunktene "Quote Endpoint" og "Search endpoint". Quote endpoint returnerer verdier knyttet til aksjens verdi og volum for forrige dag. Det er "Price" verdien vi benytter når brukere selger og kjøper aksjer i applikasjonen vår. Videre har vi benyttet Search endpoint for å gi brukere muligheten til å søke på aksjer og velge aksjer man eventuelt vil se nærmere på med quote endpoint.

Vi har valgt å lage et C# klassebibliotek for koden som benyttes for å håndtere kommunikasjon med Alpha Vantage. Fordelen med dette er at det er enklere å manuelt teste koden gjennom et testprosjekt og generelt gjenbruke koden i andre prosjekter. Videre er dette et tiltak som kan føre til

høyere kohesjon og lav kobling i applikasjonen. Klassebiblioteket heter AlphaVantageInterface og består av klassen AlphaVantageConnection som inneholder metoder for å sende http forespørsler til Alpha Vantage, håndterer respons data og sørger for at begrensningene knyttet til apiet er håndtert.

AlphaVantageInterface biblioteket benyttes i applikasjonens TradingController klasse. Der opprettes det nye AlphaVantageConnection objekter hver gang det er nødvendig å hente data fra finansmarkedet. Api nøkkelen brukt i dette prosjektet er registrert på en student bruker og er lagt til i appsettings.json under AlphaVantageApi. Api nøkkelen er tilgjengelig gjennom et IConfiguration objekt som inneholder all nøkkel/verdi-par definert i appsettings filen. Objektet blir tilgjengelig i kontrolleren gjennom dependency injection ved at en IConfiguration variabelen spesifiseres som parameter i konstruktøren. Merk at api-nøkkelen må lagres på en annen måte før prosjektet eventuelt publiseres.

Grunnet begrensningene knyttet til http forespørsler, har vi valgt en økonomisk tilnærming til innhenting av data. Både quotes og søkeresultater blir mellomlagret i databasen. På denne måten unngås det at unødvendige forespørsler gjøres mot Alpha Vantage. Dette vil ikke fullstendig løse problemet knyttet til begrensning og det vil dukke opp en feilmelding når man for eksempel gjennomfører 6 forskjellige søk etter hverandre. Dersom applikasjonen skal støtte flere brukersesjoner samtidig i fremtiden, vil det være viktig å la brukere registrere hver sin api nøkkel, og lagre api nøkkelen i databasen.

1.3.3 Beskrivelse av EcbCurrencyInterface

Ettersom de ulike aksjene man kan kjøpe i applikasjonen kan ha ulike valutaer, har vi implementert et klasse bibliotek som henter vekselkursen mellom to valutaer. Til dette har vi benyttet rss feeden til European Central Bank (ECB). The rss news feeds: <https://www.ecb.europa.eu/home/html/rss.en.html> Responsen fra rss endepunktet er i XML format og vi har derfor benyttet XmlTextReader klassen for å hente ut vekselkursen.

Koden som er laget for å hente vekselkursen er lagt til som et eget klassebibliotek som igjen er referert til som dependency i Webapplikasjoner.oblig applikasjonen. Denne inndelingen kan bidra til å oppnå målet om høy kohesjon og lav kobling ved at ansvaret for å håndtere kommunikasjonen med ecb legges inn i en egen klasse. Videre er det enkelt å benytte koden i andre prosjekter (f.eks et test prosjekt) ved å legge til en referanse i.csproj filen.

1.3.4 Databasemodellen

I dette prosjektet har vi benyttet Sqlite databasen og Entity Framework core som ORM (object-relational mapper) for å aksessere databasen fra .net core web applikasjonen. Databasemodellen er definert i klassen TradingContext som arver fra DbContext klassen. Databasemodellen som er blitt konstruert til denne oppgaven er beskrevet i Er-diagrammet i 15. Modellen består av entitetene Users, Stocks, Trades, SearchResults og StockQuotes. Videre eksisterer det mange til mange forhold mellom Users og Stocks: En bruker kan eie flere aksjer og ha flere aksjer i sin favorittliste. Videre kan en aksje være i favorittlisten til flere brukere og være kjøpt av flere brukere. Videre er det et mange til mange forhold mellom SearchResults og Stocks ettersom flere aksjer kan være i et søkeresultat og en aksje kan være del av ulike søkeresultater.

For å representere disse mange til mange forholdene, opprettes hjelpetabeller (join tables). I utgangspunktet vil entity framework gjøre dette automatisk dersom et mange til mange forhold er definert implisitt gjennom navigation properties i DbSet klassene definert i TradingContext. EF vil da sette sammen navn på tabell og attributter ut fra de to entitetene som inngår i forholdet. I vår databasemodell er alle navnene til hjelpetabellene definert eksplisitt: FavoriteLists, StockOccurrences og StockOwnerships. Merk at StockOwnerships er en tabell som inneholder flere attributter enn bare primær- og fremmednøkklene (SpentValue og StockCounter). Derfor er denne tabellen definert som en egen entitet i TradingContext. For at EF skal opprette tabellen med både Symbol (aksje id) og UserId som både primær og fremmednøkkel er det viktig at StockId og UserId er

definert som attributter i tillegg til navigation property for både Stocks og Users tabellen.

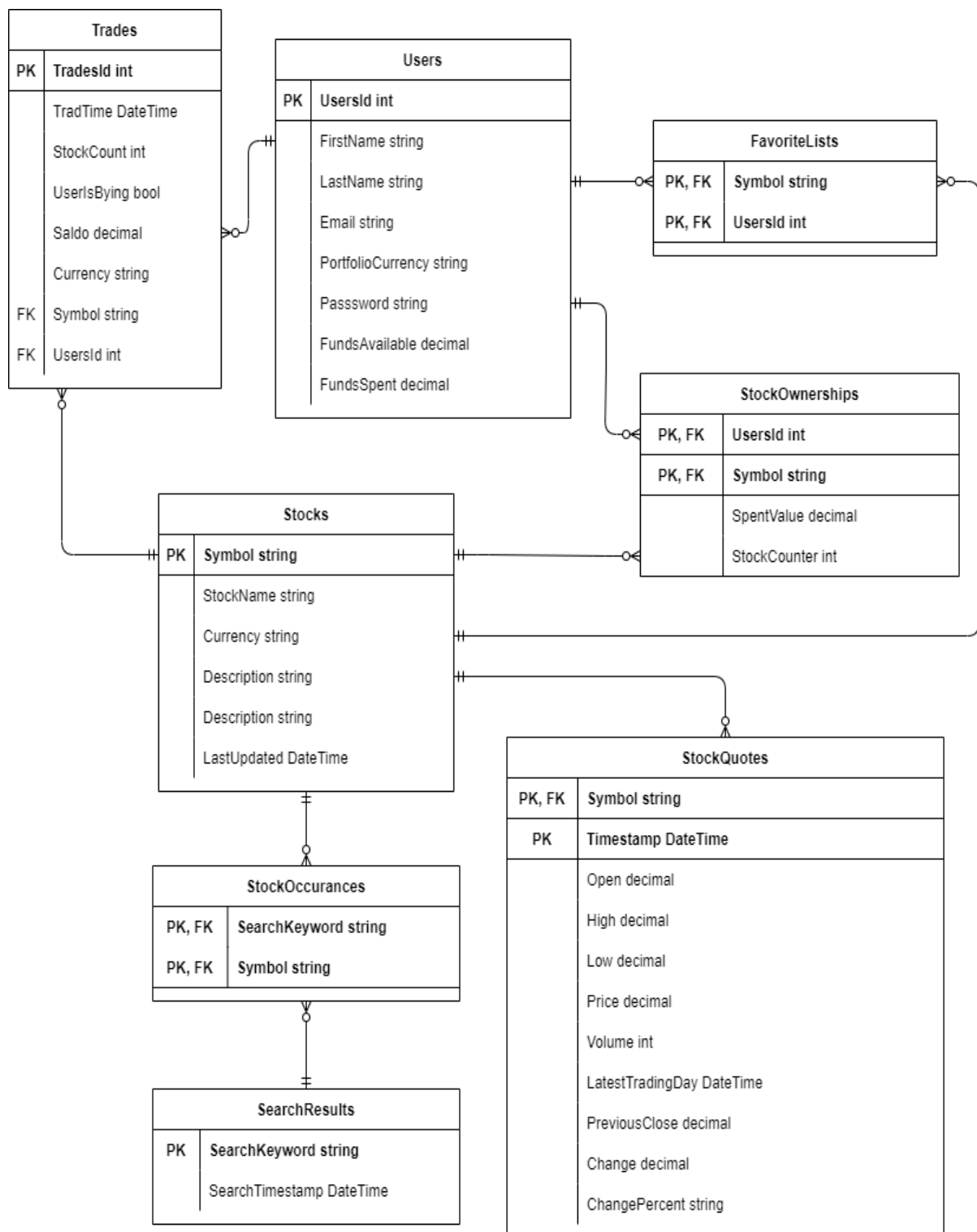


Figure 4: Er-diagram som beskriver databasemodellen som er definert i TradingContext.cs

1.3.5 Migration og seeding av databasen

Med Entity Framework finnes flere metoder for å bygge databasen ut fra kontekst klassen. I dette tilfelle har vi benyttet migrations funksjonaliteten i stedet for å benytte EnsureCreated og EnsureDeleted metodene. Ved hjelp av migrations kan eksisterende data i databasen bevares ved endringer av databasemodellen. Videre er det kun lagt til et user brukerobjekt som skal legges til i databasen når den opprettes første gang (seeding). Hver gang det foretas endringer i database modellen (f.eks legge til en attributt) kan det opprettes en ny migrasjon med "Add-Migration migrationName". Databasen kan dermed bygges på nytt baserende på den siste migrasjonen med "Update-Database". Resultatet er en database med en oppdatert struktur samtidig som data, som var lagret i den forrige versjonen, er tilgjengelig i den nye databasemodellen. Merk at seeding ikke forekommer etter første migrasjon. Dersom man ønsker å starte med en helt ny database kan man slette TradingSchema.db filen i prosjektmappen og utføre "update-Database" på nytt. Kommandoene knyttet til migrasjonen utføres i "package manager console".

1.3.6 Hosted service for rydding av databasen

Lagring av søkedata kan føre til at mye data i form av at aksje objekter og søkeresultat objekter er lagret i databasen uten at det blir aktivt brukt av brukere. I forbindelse med dette er det lagt til en metode i TradingRepository: CleanTradingSchemaAsync(). Denne metoden fjerner søkeresultater som ble lagret for over 24 timer siden. Videre fjernes alle Stocks objekter og tilhørende StockQuotes objekter som ikke har noen relasjon til en bruker (portfolio eller favorittliste) eller et søkeresultat.

For at denne metoden kjøres en gang daglig, er det implementert en hosted service, "TradingSchemaWorker", som starter når applikasjonen startes og kjører i bakgrunnen som en uavhengig tjeneste. TradingSchemaWorker klassen er basert på Worker malen i dotnet. Videre defineres det et Timer objekt i StartAsync metoden, som starter en syklus der CleanTradingSchemaAsync() kalles når klokken til maskinen som kjører programmet er 00:00:00. Merk at "CleanTradingSchemaAsync" metoden kjøres en gang ved oppstart av applikasjonen.

2 Beskrivelse av applikasjonens funksjonalitet

2.1 Funksjonalitet knyttet til watchlist/favorittliste

Legge til/fjerne aksje fra watchlist på klient siden Brukeren må gå inn i "Market" på menyen og søke opp en ønsket stock for å legge til favorittsiden med knappen "Add to Watchlist", og omvendt kan det fjernes en aksje fra favorittsiden med knappen "Remove from Watchlist". Eks. Se figur 5. På favorittsiden under Trading, kan man velge en aksje i favorittlisten ved å trykke på linjen i tabellen. Dette vil hente stock quote for aksjen (sist tilgjengelig data knyttet til aksjen) og vise det i vinduet til høyre. Om man ønsker å kjøpe den valgte aksjen kan man velge "Buy" knappen (se figur 6).

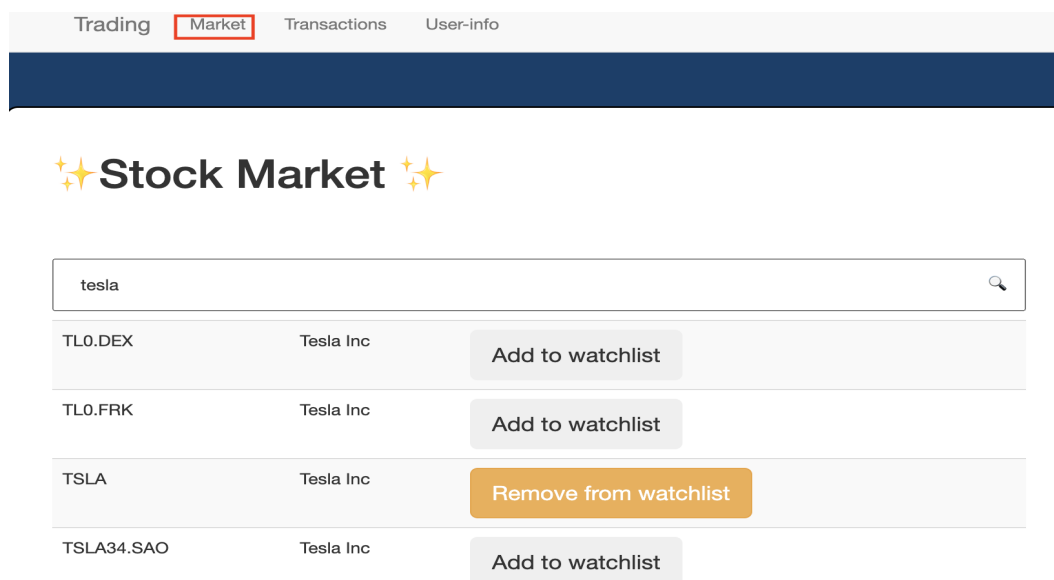


Figure 5: Aksje søk side

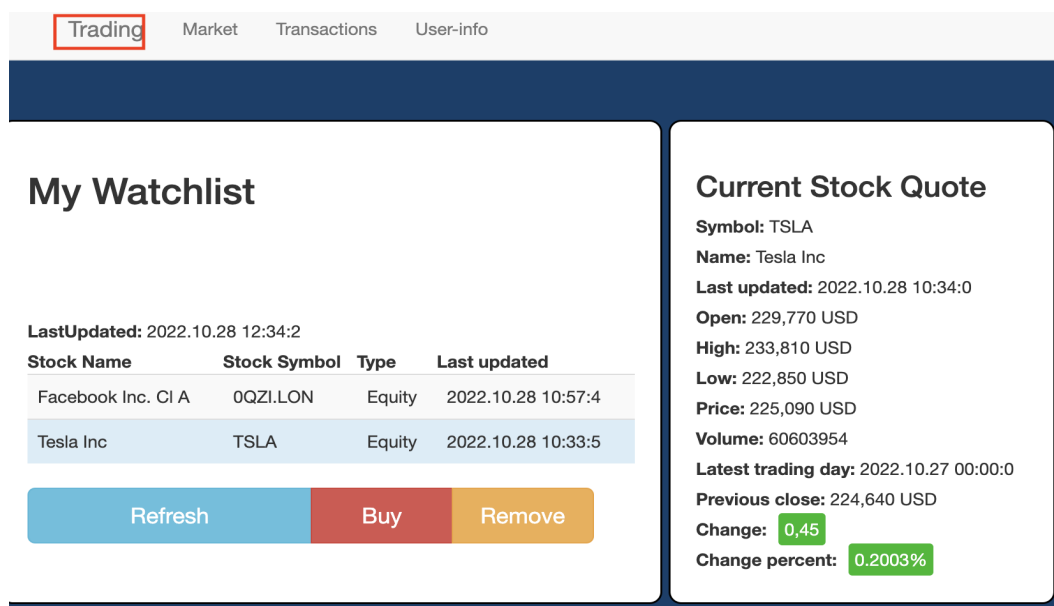


Figure 6: Aksje favoritt side

TradingRepository Klientsiden gjennomfører en get forespørsel til /trading/GetFavoriteList endepunktet med user id som parameter brukt for å identifisere favorittlisten til brukeren. Metoden "GetFavoriteList" definerer endepunktet på tjenersiden i TradingController klassen. Denne metoden kaller GetFavoriteListAsync metoden i TradingRepository klassen som igjen henter ut brukers favorittliste gjennom navigation property Favorites som er definert under Users entiteten i Kontekst klassen. Videre omgjøres denne database favorittlisten til et FavoriteList object som igjen returneres til klienten.


```

/**
 * This metode will obtain favorites for the user that matches the userId.
 * Parameters:
 * (int) userId: The user to find the favorite list for.
 * Return: FavoriteList object containing the favorite list of the given user
 */
2 references | 0 changes | 0 authors, 0 changes
public async Task<FavoriteList> GetFavoriteListAsync(int userId)
{
    // Find the user in the database
    Users oneUser = await _db.Users.SingleAsync(u => u.UsersId == userId);
    // Getting the favorites of a user
    List<Stocks>? favorites = oneUser.Favorites;
    // Declaring the new favorite list containing StockBase objects
    List<StockBase>? stockFavorite = new List<StockBase>();
    StockBase currentStockDetail;

    foreach (Stocks currentStock in favorites)
    {
        // Foreach favorite in the favorite list, create a new StockBase object and
        currentStockDetail = new StockBase
        {
            StockName = currentStock.StockName,
            Symbol = currentStock.Symbol,
            Type = currentStock.Type,
            LastUpdated = currentStock.LastUpdated
        };
        stockFavorite.Add(currentStockDetail);
    }
    // Create the favorite list if all stocks have been iterated over.
    var currentFavorite = new FavoriteList
    {
        LastUpdated = DateTime.Now,
        StockList = stockFavorite
    };
    return currentFavorite;
}

```

Figure 7: GetFavourittListAsync definert i Respository

```

/**
 * This method is used as an endpoint for finding the watchlist/favorites of a user.
 * The favorite list contains a list of all stocks that a user has marked as a favorite.
 * Parameters:
 * (int) userId: The user that is connected to the favorite list.
 * Return: A FavoriteList object.
 */
2 references
public async Task<FavoriteList> GetFavoriteList(int userId)
{
    // The favorite list is obtained from the repository
    return await _tradingRepo.GetFavoriteList(userId);
}

```

Figure 8: GetFavoriteList definert i TradingController

2.2 Kjøp av Aksje

2.2.1 Kjøp av aksjer på klientsiden

En bruker kan kjøpe aksjer via portfolio eller via en egen favorittliste. For å kunne bruke "buy" funksjon så må brukeren ha valgt en aksje ved å trykke på den. Når det er gjort og man trykker på "Buy" knappen, kommer det et nytt vindu der man kan velge antall av aksjer som man ønsker å kjøpe (se 9). JQuery sin "\$.Post()" utility metode benyttes for å sende http forespørselen til Rest endepunktet for kjøp operasjonen. Det sendes med 3 parametre (userId, aksje Symbol (primærnøkkel av en aksje) og antall aksjer som skal kjøpes). Når klienten har mottatt responsen kalles javascriptfunksjonen "updatePortfolioList" for å vise portfolio når den har blitt oppdatert.

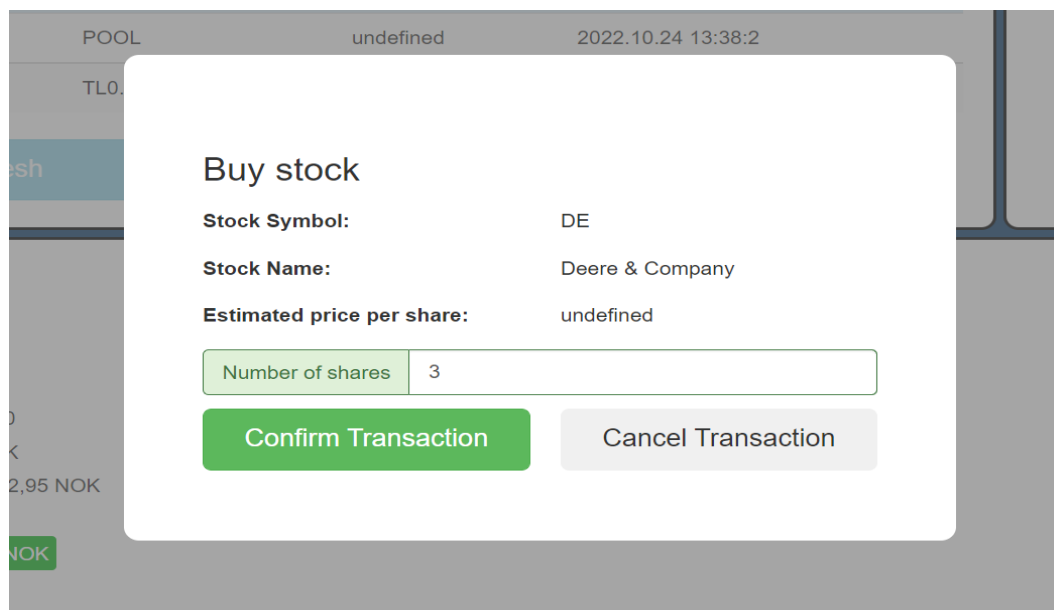


Figure 9: Kjøpe en aksje

2.2.2 Kjøp operasjonen på tjenersiden

I TradingController.cs på server siden er funksjonen "BuyStock" definert. Det sjekkes først om antall er et positivt heltall, ellers får brukeren en feilmelding. Videre kontrolleres det at informasjon om aksje (pris, osv.) er oppdatert. Hvis ikke hentes oppdatert informasjon fra AlphaVantage API. Funksjon sjekker også at valuta for aksjen er den samme som brukeren har registrert. Hvis ikke kalles en funksjon fra EcbCurrencyHandler for å få tilbake vekslingskurs for å beregne hvor mye transaksjon vil koste med brukerens valuta. Den totale saldoen beregnes ut fra "price" Quote data fra Alpha Vantage. Funksjon sjekker videre at transaksjonen vil koste mindre enn det brukeren har tilgjengelig på kontoen sin. Hvis ikke det er tilfelle får brukeren beskjed om at det ikke er nok penger på konto.

Etter dette kalles funksjonen "buyStockTransaksjon" med 4 parametre (brukeren, aksje, saldo, antall) fra TradingRepository.cs. Denne metoden foretar alle nødvendige endringer på databasen for å gjennomføre kjøpet. Det blir blant annet gjort endringer på StockOwnership objektet for brukeren og aksjen, slik at antall registrerte askjer økes og antall penger investert økes. Dersom det ikke er noen eksisterende eierskap, opprettes et nytt eierskap. Det oppdateres også penger tilgjengelig på konto. En transaksjon blir også lagret i database sånn at brukeren kan se det (fra transaksjon i navbar). Merk at alle endringene som foretas mot databasen vil bli utført som en transaksjon av Entity Framework når SaveChanges() blir kalt. Til slutt returneres et oppdatert portfolio objekt til klienten slik at endringene vises i frontend.

Dette gjøres av funksjonen GetPortfolio i TradingController.cs. Målet med denne funksjon er å beregne forskjellige nyttige informasjoner knyttet til brukerens aksjeeierskap. Dette gjør at vi kan vise informasjon om aksjeportefølje generelt som:

- Verdien til aksjeportefølje
- Kostnaden til aksjeportefølje
- Tilgjengelige midler brukeren har
- Hvor mye brukeren har vunnet eller tapt inntil nå. Vi bruker denne funksjon også for å kunne vise noen informasjon om hver aksje:
 - i. Vanlig informasjon om en aksje

- ii. Hvor mye man har tjent eller tapt med en aksje
- iii. Hvor høy prosent hver aksje representerer i forhold til hele aksjeportefølje.

For å kunne vise disse informasjoner så må vi beregne en del ting. For hver aksje hentes oppdaterte verdier for aksjen (stock quotes). Disse kan ha forskjellige valuta. Så vi henter vekslingskurs fra EcbCurrencyHandler hver gang det trengs sånn at vi kan ha verdien til hver aksje i brukerens valuta. Vi henter fra database total kostnad av en aksje. Så trekker vi fra den til dagens totalverdi av denne aksjen som brukeren har. Har den kostet mer enn den er verdt vil tallet være negativ. Vi gjør dette med hver aksje som er i portfolio til brukeren. Vi legger også til kostnaden av hver aksje til total kostnad av hele portfolio. vi deler totalverdi til en aksje med verdien til hele portfolio og ganger med 100 for å se hvor mye hver aksje representerer i forhold til portfolio i prosent. Vi trekker fra total verdi brukt til total verdi av portfolio for å kunne vise brukeren hvor mye som er tjent eller tapt. Denne metoden fungerer også som endepunkt for klienter som ønsker å hente aksjeporteføljet til en bruker

Se figure 10 for et sekvensdiagram som er en forenklet oversikt over kjøp prosessen. Merk at kun de mest sentral alternative flytene er lagt til.

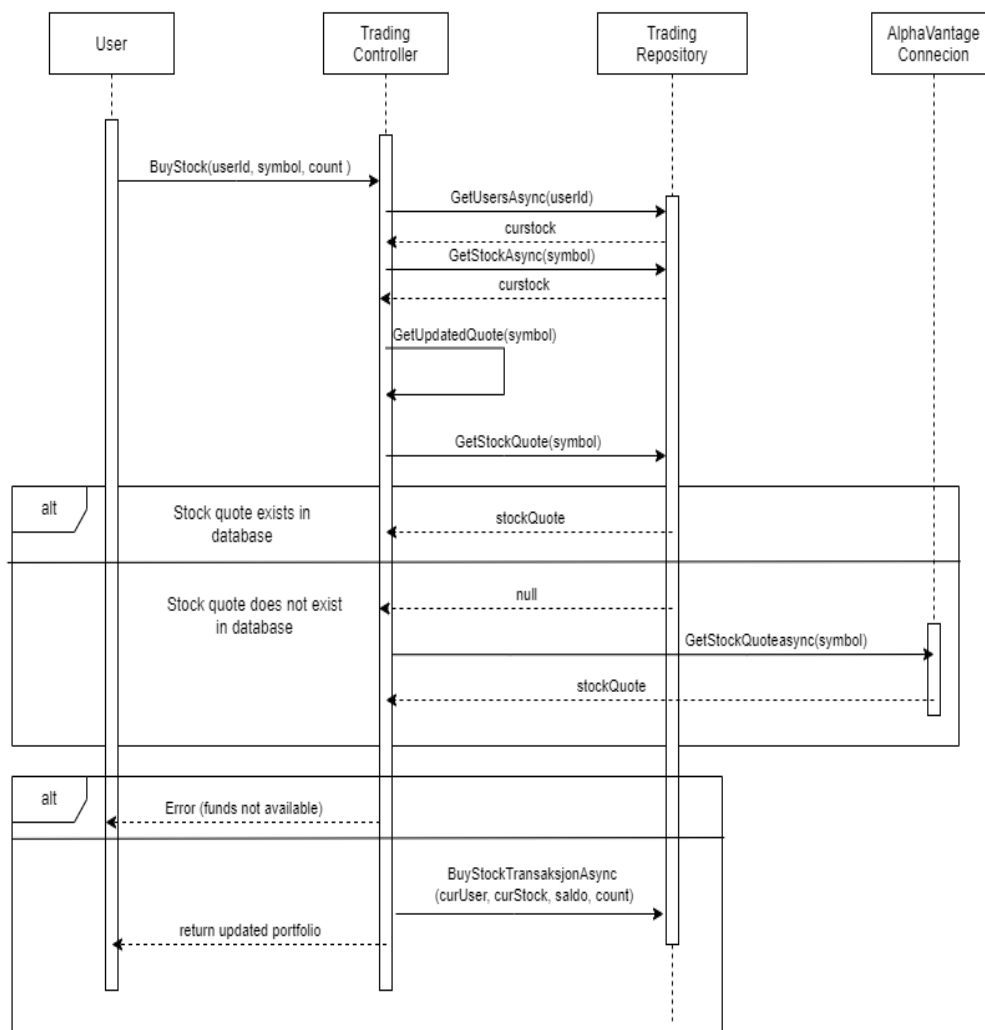


Figure 10: Sekvensdiagram som beskriver kjøp av en aksje.

2.3 Selge Aksje

2.3.1 Salg på klientsiden

I trading system skal brukeren finne ut Portfolioen. Etter brukeren er inne i systemet skal det velges kun et Stock og trykke sell knappen. Brukeren får mulighet til å oppgi antall aksjer han/hun ønsker å selge. Etter transaksjonen er akseptert vil transaksjonen lagres og kan ses under Transaction på menyen.

Quantity	Estimated price	Portfolio partition %	Estimated total value
2	4 095,52 NOK	78,08%	8 191,04 NOK

Figure 11: Aksje salg side

2.3.2 Salg på tjenersiden

En HTTPRequest sendes til Trading kontrolleren på serversiden (endepunkt SellStock). Funksjonen SellStock i TradingController henter oppdatert Stock quote ved å kalle "GetUpdatedQuoteAsync". Videre hentes brukerobjektet med "GetUsersAsync(userId)" fra DAL-TradingRepository. Den totale penge verdien brukeren får blir så beregnet og transaksjonsmetoden, "SellStockTransactionAsync" i repository, kalles for å fullføre salget.

```

/**
 * This method is the endpoint used to execute a sell operation for a specific user.
 * Parameters:
 * (int) userId: The user that is selling.
 * (string) symbol: The identity of the stock that should be sold.
 * (int) count: The amount of shares that should be sold of the specified stock
 */
namespace Webapplikasjonjerner.oblig.Controllers
{
    public async Task<Portfolio> SellStock(int userId, string symbol, int count)
    {
        // Check if the stock exists in the database
        Stocks? curStock = await _tradingRepo.GetStockAsync(symbol);
        if (curStock is null)
        {
            throw new NullReferenceException("The stock was not found in the database");
        }

        // Validate stock counter. It must be an positive integer greather than or equal to 1
        if (count < 1) {
            throw new ArgumentException("The provided count value is invalid");
        }

        // Get the updated quote for the stock
        StockQuotes curQuote = await GetUpdatedQuote(symbol);

        // Get user
        Users? identifiedUser = await _tradingRepo.GetUsersAsync(userId);
        // Verifying that a user was found
        if (identifiedUser is null)
        {
            throw new ArgumentException("The provided userId did not match any user in the database!");
        }

        // Finding the total that needs to be added to the users funds
        decimal exchangeRate = 1;
        if (identifiedUser.PortfolioCurrency != curStock.Currency)
        {
            // Get the exchange rate from Ecb if the user currency differs from the stock currency
            exchangeRate = await EcbCurrencyHandler.GetExchangeRateAsync(curStock.Currency, identifiedUser.PortfolioCurrency);
        }

        // Calculating the total saldo with the amount of stocks and correct currency
        decimal saldo = (decimal) curQuote.Price * count * exchangeRate;

        // Execute the sell transaction against the database
        await _tradingRepo.SellStockTransactionAsync(userId, symbol, saldo, count);
        // Return an updated portfolio object
        return await GetPortfolio(userId);
    }
}

```

Figure 12: Selge aksje metode

```

/**
 * This method executes all operations against the database that are needed to fullfill
 * a sell operation in the application. The operations are applied to the database as one transaction.
 * Parameters:
 * (int) curUser: The user that executes the sell operation
 * (string) symbol: The stock that should be sold
 * (decimal) saldo: The monetary value that the user receives from the transaction
 * (int) count: The amount of shares that should be sold
 */
2 references
public async Task SellStockTransactionAsync(int userId, string symbol, decimal saldo, int count)
{
    // Get the User object from the database - InvalidOperationException if it does not exist
    Users? curUser = await _db.Users.SingleAsync<Users>(t => t.UserId == userId);
    // Add the saldo to the buying power of the user
    curUser.FundsAvailable += saldo;

    // Remove the stock or a shares from the portfolio
    StockOwnerships? curOwnership = await _db.StockOwnerships.SingleAsync<StockOwnerships>(t => (t.StocksId == symbol) && (t.UserId == userId));
    if (curOwnership.StockCounter < count) {
        // Check that the user has enough shares of the stock to complete the transaction
        throw new ArgumentException("The specified amount of stocks to sell exceeds the amount of shares owned!");
    }

    // Subtract from the stock counter of the ownership
    curOwnership.StockCounter -= count;
    curOwnership.SpentValue += saldo;

    if (curOwnership.StockCounter <= 0)
    {
        // The user has no more ownership of this stock type
        curUser.Portfolio.Remove(curOwnership);
    }

    // Create a new trade object
    Trades tradeLog = new Trades {
        StockCount = count,
        TradeTime = DateTime.Now,
        UserIsBuying = false,
        Saldo = saldo,
        Currency = curUser.PortfolioCurrency,
        Stock = curOwnership.Stock,
        User = curOwnership.User
    };

    // Adding the new Trade object
    _db.Trades.Add(tradeLog);
    // Applying changes to database
    await _db.SaveChangesAsync();
}

```

Figure 13: SellStockTransaction metoden i TradingRepository.cs

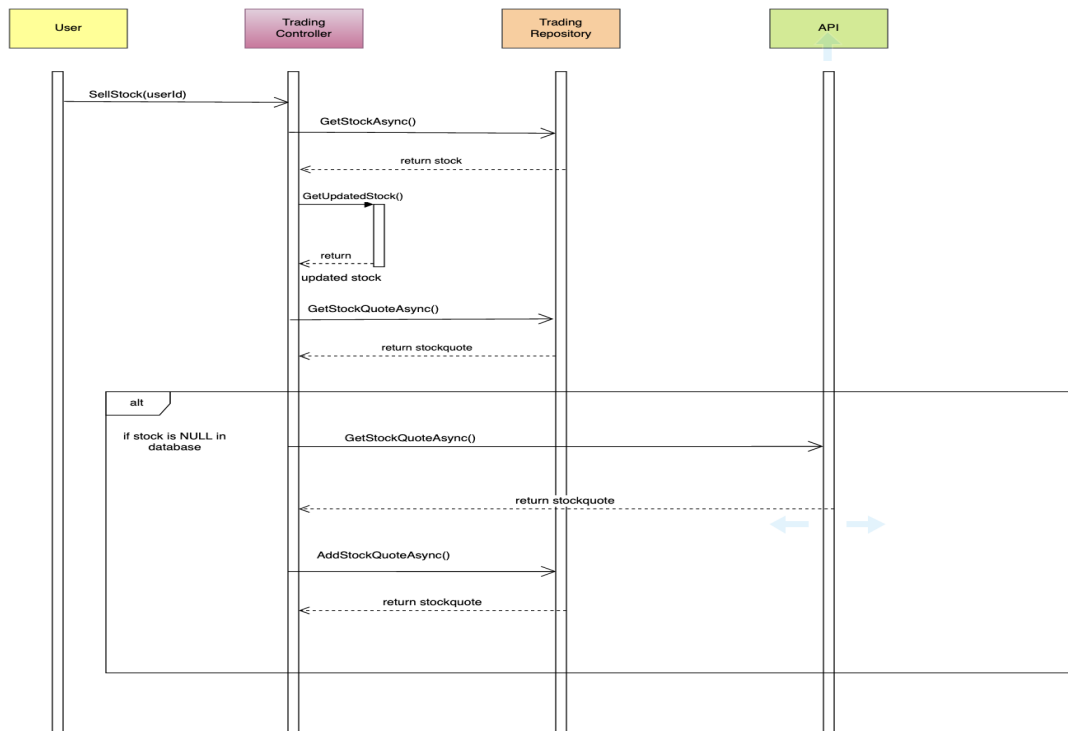


Figure 14: Sekvensdiagram for salg av aksje

2.4 Søking

2.4.1 Søking på klient siden

Bruker grensesnittet inkluderer en navigasjon knapp som heter "Market". Denne knappen vil åpne siden for søking av aksjer. Et skjermbilde er lagt til i figur 15:

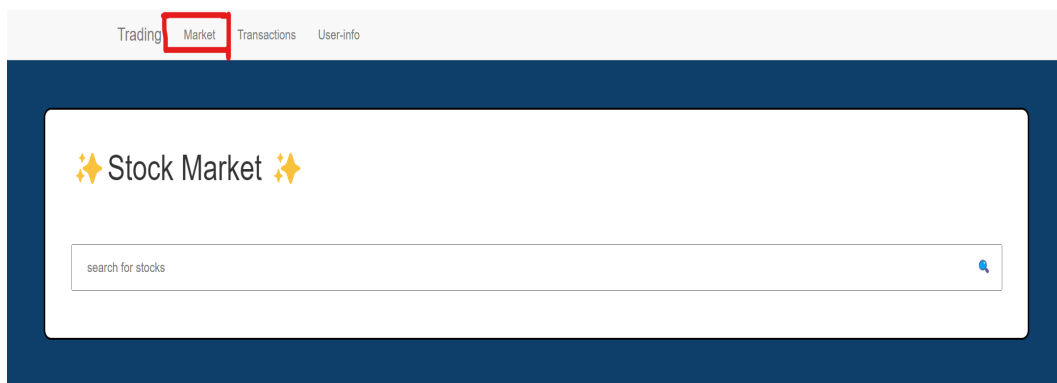



Figure 15: Aksje søking side

Etter at brukeren tastet inn et nøkkelord og trykker på  endres elementets inputverdi. Da blir JQuery sin change funksjon utløst. Denne funksjonen gjør klar en url med event target sin value som parameter verdi og bruker sin id. For øyeblikket er det bare en bruker så int verdi 1 er gitt som id til brukeren. Url en til er da passert som parameter til JQuery sin get utility metode. Responsen fra get metoden er sjekket for verdi. Hvis den er tom, blir passelig feilmelding vist. Hvis responsen kommer med data er dette en et objekt med blandt annet property StockList som

inneholder alle aksjene. Listen gås gjennom ved bruk av en loop og blir skrevet ut i tabell format. Alt dette behandles i `tradeSearch.js` filen.

2.4.2 Søkning på tjener siden

På tjenersiden går http forespørselen til trading kontrolleren og metoden `GetUserSearchResult(string keyword, int userId)`. Denne funksjonen henter brukeren som matcher id-en. Dette gjøres ved å kalle `GetUsersAsync(userId)` fra DAL's `tradingRepository`.

```
/**
 * This method executes the search operation used to find stocks with a given keyword.
 * It is used as the endpoint for stock search on the client side. The endpoint returns a SearchResult
 * object containing a list of StockSearchResult objects containing the IsFavorite flag, which indicates
 * if the stock is in the watchlist of the user with the specified userId
 * Parameters:
 * (string) keyword: The search keyword used to match the available stocks.
 * (int) userId: The userId of a user, used to identify if a stock is in the watchlist or not.
 * Return: The method returns a SearchResult object
 */
0 references
public async Task<Model.SearchResult> GetUserSearchResult(string keyword, int userId)
{
    // Checking that the keyword is not an empty string
    if (keyword == "")
    {
        // Return an empty SearchResult object
        return new Model.SearchResult();
    }
    // Obtaining the Users object (entity)
    Users curUser = await tradingRepo.GetUsersAsync(userId);
    if (curUser is null)
    {
        throw new ArgumentException("The specified user was not recognized");
    }
    // Obtaining the watchlist of the user, adding an empty list if the favorites property is null
    List<Stocks> favoriteStockList = (curUser.Favorites is null ? new List<Stocks>() : curUser.Favorites);
    // Executing the search and making sure that the search result is stored in the database
    Model.SearchResult result = await SaveSearchResult(keyword);

    // Going through the stocks in the search results
    foreach (StockSearchResult curStock in result.StockList)
    {
        // For each search result stock, check if it is in the watchlist
        foreach (Stocks favStock in favoriteStockList)
        {
            if (curStock.Symbol == favStock.Symbol)
            {
                // Setting the IsFavorite flag on the stock if it is in the watchlist
                curStock.IsFavorite = true;
            }
        }
    }
    return result;
}
```

Figure 16: Bruker Søk resultat

Metoden `SaveSearchResult(keyword)` er definert i kontrolleren og returnerer et Task med resultat type av `SearchResult`.

```

/**
 * A function to add a search result using a received word from client
 * and checking if record matching exist in database.
 * If it exist already it checks its last update, if last update is greater than 24
 * then removes existing record and add new one by creating a new object of search result and passing it to saveSearchResult in repository.
 * If there is no such record, it fetches data from api using the keyword and save it by passing it to saveSearchResult function
 */
2 references
public async Task<Model.SearchResult> SaveSearchResult(string keyword)
{
    // The search results are stored with keyword name as uppercase uppercase. Thus we operate with keyword in uppercase
    // to implement a non-casesensitive search feature
    keyword = keyword.ToUpper();
    // /trading/saveSearchResult?keyword=Equinor

    // Try and find a search result with given keyword in searchresults table
    Model.SearchResult res = await _searchResultRepository.GetAsync(keyword);

    // If there is no search result stored in the database, then go a head and fetch it from
    // Alpha vantage api
    if (res is null)
    {
        return await createNewSearchResult(keyword);
    }

    // If there exist a search result match then check when it was added.
    double timeSinceLastUpdate = (DateTime.Now - res.SearchTime).TotalHours;
    if (timeSinceLastUpdate >= _quoteCacheTime)
    {
        _searchResultRepository.DeleteSearchResult(keyword);
        return await createNewSearchResult(keyword);
    }
    return res;
}

```

Figure 17: Lagre Søk result

`GetOneKeywordAsync(keyword)` metoden fra Search Result Repository mappen aksesserer databasen og hente et søk resultat fra tabellen `SearchResults`. Hvis det ikke er noe data som tilsvarer nøkkel ordet så returnerer metoden null.

I dette tilfelle gjøres det et kall til metoden `CreateNewSearchResult(keyword)`. Denne metoden lager et nytt søk resultat objekt ved å gjøre et kall til AlphaVantage API. En dependency som heter `AlphaVantageInterface` fungerer som et grensesnitt til AlphaVantage API-en og metoden bruker denne dependency til å kommunisere med AlphaVantage API. Aksjen som hentes fra API lagres i databasen. Hvis det gjøres et søk igjen for denne aksjen så kan den hentes fra databasen. Deretter returnere funksjonen søk resultatet. I det tilfellet søket returnerer et verdi og ikke et null, sjekkes det når dataen ble lagt til databasen. Hvis det er over 24 timer siden dataen ble lagt til databasen slettes den og gjør et nytt kall til API-en.

Nede følger det et Sekvensdiagram som viser hvordan et søk funksjonalitet oppnås i systemet.

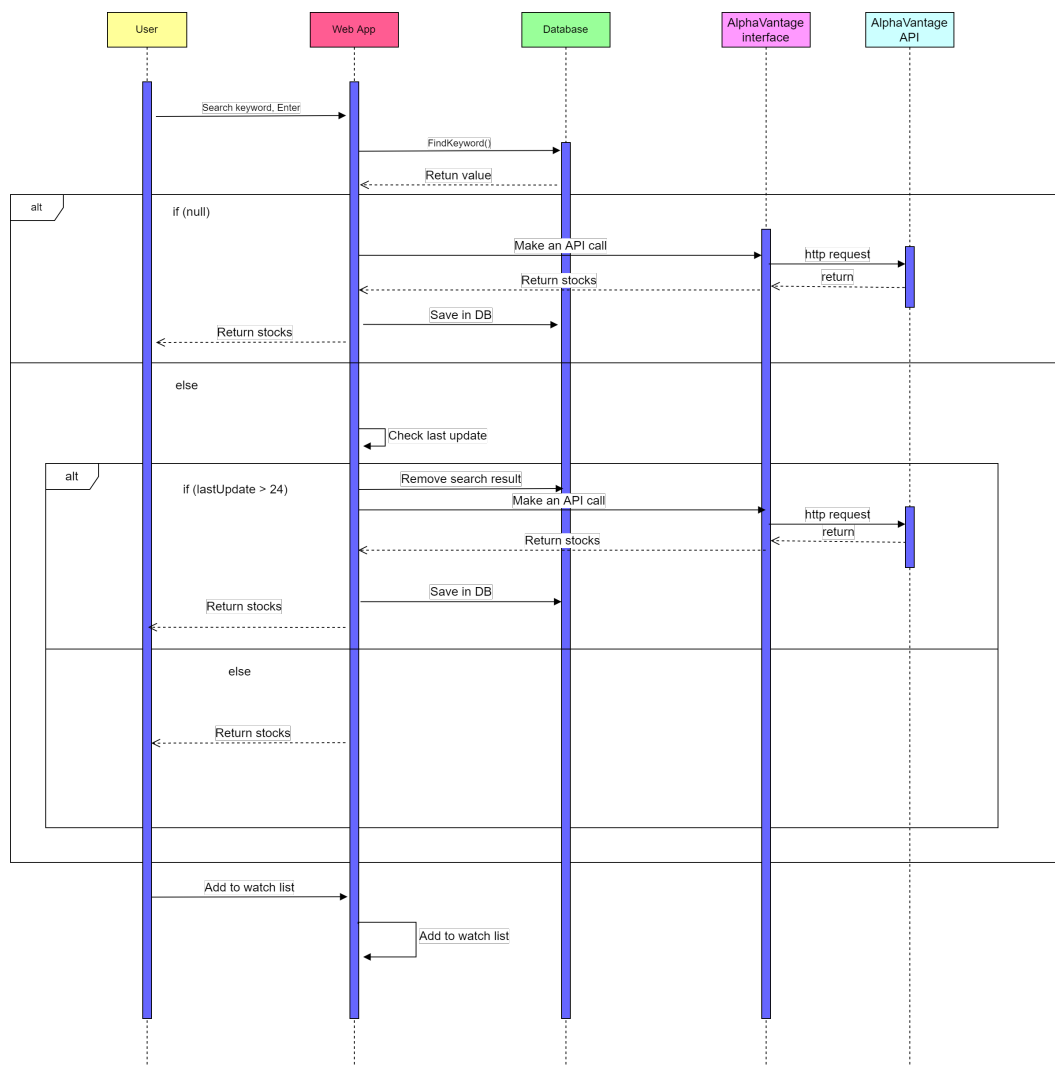


Figure 18: Er-diagram Søk resultat

2.5 Transaksjoner og bruker instillinger

2.5.1 Transaksjon Side

Transaksjonsiden viser de forskjellige transaksjonene som brukeren har gjort tidligere. "Update" og "clear all" knappene er de 2 eneste knappene her. "Clear all" sletter visning av alle transaksjoner og etter at man har trykket på den så kan man ikke lenger se gamle transaksjoner. Oppdatering av transaksjonslisten skjer automatisk ved innlasting av siden, men i noen tilfeller vil "Update" være nyttig for å hente den oppdaterte listen manuelt.

2.5.2 Bruker instillinger side

Her kan brukeren modifisere navn, e-post og valuta. Man skal trykke på update knappen for å lagre endringene i database. Vi har ikke håndtert autentisering og dermed er det ikke mulig å oppdatere passordet til brukeren i denne versjonen av applikasjonen.

Valg av valuta: Det forandrer valuta som er brukt i portfolio for å vise diverse pengeverdier for brukeren. Det vises også hvor mye penger brukeren har brukt og hvor mye som er igjen på konto. reset knap: starter en operasjon der portfolio, favorittlist og transaksjoner knyttet til brukeren slettes. Man starter på nytt med 1 000 000 nok etter det.